Master thesis and internship[BR]- Master's thesis : Topology Optimization of Fixture Clamping Locations for Machining Processes[BR]- Integration internship

Auteur : Adler, Joé
Promoteur(s) : Salles, Loïc
Faculté : Faculté des Sciences appliquées
Diplôme : Master en ingénieur civil en aérospatiale, à finalité spécialisée en "aerospace engineering"
Année académique : 2024-2025
URI/URL : http://hdl.handle.net/2268.2/23334

Université de Liège
Faculté des Sciences Appliquées

Topology Optimization of Fixture Clamping Locations for Machining Processes

Final work

Final thesis completed in order to obtain the
Master's degree in 'Civil Engineering in aerospace engineering'
by Adler Joé.

Author:
ADLER Joé

Academic advisor:
SALLES Loïc

Jury:
SALLES Loïc, BRUYNEEL Michaël, DUYSINX Pierre

Academic year 2024-2025

Acknowledgements

I would like to express my deepest gratitude to all those who have supported me throughout the journey of writing this thesis.

First and foremost, I would like to thank my supervisor, Loïc Salles, for his guidance and open-mindedness. His support and constructive feedback were essential in shaping the direction of this work.

I would also like to extend my sincere thanks to Pierre Duysinx for his insightful advice and for providing me with a clear direction at a crucial stage in my thesis. His input was vital in giving this work a solid foundation and purpose.

A special thank you goes to my classmates, whose camaraderie and teamwork made this experience both enjoyable and enriching. The collaborative atmosphere we shared made the challenges of this journey much easier to navigate.

Finally, I would like to thank my parents and grandparents for their continuous support and encouragement throughout my studies. Their understanding and help provided me with the motivation to complete this work.

Abstract

This work addresses the optimization of fixture clamping locations to minimize workpiece deformation under prescribed external machining forces. The approach employs finite element analysis (FEA), introducing bar elements on the exterior surface of a 3D tetrahedral workpiece mesh. The design variables are the densities of these bar elements, formulated using the SIMP homogenization method. A linear static analysis is used to evaluate the objective function. The algorithm, implemented in Python, includes finite element assembly and sensitivity analysis. Optimization is performed using a prewritten Python module that implements the Method of Moving Asymptotes (MMA). To improve computational efficiency, the Guyan-Irons method is successfully applied, reducing the system by keeping only relevant degrees of freedom (DOFs), thus significantly reducing computation time. The framework is demonstrated on two test cases: a simple academic example involving a cylindrical workpiece under point loading, and a more realistic turbine impeller subjected to a machining tool path. The effect of physical and numerical parameters on the optimal solution is investigated to reduce intermediate density values. Since tetrahedral elements are not ideal for high-accuracy simulations, this work is intended as a proof-of-concept.

Table of Contents

1 Introduction

This section introduces the present work, which focuses on the optimization of fixture clamping locations in machining processes. It begins by discussing the role and importance of machining fixtures to establish the context. The motivation for optimizing clamping locations is then presented. A brief overview of optimization problems is provided to introduce key concepts. Finally, the structure of the thesis is outlined to guide the reader through the subsequent chapters.

1.1 Machining Fixtures

In subtractive manufacturing, *machining* refers to the process of transforming a raw metal part, known as the *workpiece*, into a final geometry that meets precise tolerance requirements. These tolerances are critical for the part's functionality in a wide range of applications, such as automotive components and aerospace systems. Machining can also involve modifying an existing part that requires adjustments. The process typically consists of a carefully planned sequence of steps. Often, it is not possible to achieve the final shape in a single CNC pass; instead, the part must be machined in several intermediate stages to reach the desired specifications.

One of the most common machining operations is milling. In this process, a rotating cutting tool moves at high speed and comes into contact with the workpiece to gradually remove material in the form of small chips. The tool's movement, combined with the controlled motion of the workpiece, often along multiple axes, is guided by instructions programmed into the CNC system. These instructions specify exactly where and how much material to remove to shape the part according to design requirements. Milling can be used for a wide variety of tasks, such as contouring, slotting, drilling, and finishing surfaces. Its flexibility makes it essential in industries ranging from precision engineering to heavy manufacturing. Turning is another fundamental machining operation, commonly performed on a lathe. In this process, the workpiece rotates at high speed while a stationary cutting tool is brought into contact with its surface to remove material. Turning is typically used to create cylindrical or conical shapes and is ideal for producing parts like shafts, pins, and bushings. The cutting tool can move along different directions, usually parallel or perpendicular to the workpiece axis, to achieve the desired dimensions and surface finish. Like milling, turning is guided by CNC programming for precision and repeatability.

Before a part can be machined, it must be attached to the machine table, a task accomplished using devices known as *fixtures*. Fixtures are essential for accurately positioning the workpiece on the CNC machine for operations such as milling and turning. They serve two primary functions: locating the workpiece in the correct position, and clamping it firmly in place during machining. The first function is handled by *locating elements* (or *locators*), which ensure the part is aligned as required. The second is managed by *clamping elements* (or *clamps*), which apply the necessary force to hold the part steady throughout the machining process, ensuring both precision and safety.

Accurate positioning of the workpiece is a critical aspect of machining, and this is primarily achieved through the use of locating elements. These components are designed to restrict

the workpiece's movement by eliminating its six degrees of freedom (3 in translation and 3 in rotation) to ensure consistent and precise alignment relative to the cutting tool. Locators are typically fixed elements that provide repeatable references for the part while also supporting it during machining. Depending on the geometry and design of the workpiece, various types of locators may be used, such as plane locators for flat surfaces, concentric locators that fit into pre-machined holes, and radial locators to prevent unwanted rotation. These elements are often complemented by supports like rest buttons or adjustable pins, which not only stabilize the part but also absorb cutting forces. Effective placement of locators is crucial, especially to avoid interference from machining debris, which can compromise accuracy if it builds up between the locator and the part (Pereira 2023).

Clamping elements play a critical role in securing the workpiece during machining by applying a force that presses it firmly against the locators. The clamping force should always be directed toward the locators to maintain accurate positioning, especially under the influence of machining forces. Poorly designed clamp placement can lead to elastic deformation, resulting in machining errors once the part is released. Different clamping devices exist to accommodate various part geometries, such as strap clamps, cam clamps, and swing clamps, each offering specific advantages in terms of speed, force application, and accessibility. Optimizing the clamp force and layout, often with the help of methods like finite element analysis and genetic algorithms, can significantly reduce part deformation and improve overall fixture performance, surpassing traditional trial-and-error approaches (Pereira 2023).

1.2 Context and Motivation

The initial objective of this work was to explore the potential of 3D printing machining fixtures generated through topology optimization. In precision mechanics, one of the most time-consuming tasks is the development of machining fixtures for various workpieces. Each fixture must be custom-designed to withstand cutting forces, including both elastic deformation and vibrations. Furthermore, machining a raw metal piece into a finished product often requires multiple CNC operations. Engineers in the technical bureau must plan these steps meticulously to ensure that all necessary areas are machined. A fixture designed for one operation may obstruct access to other critical areas, complicating the process. Additionally, the design phase often involves several iterations before achieving the required tolerances.

Given these challenges, automating and optimizing fixture design could significantly reduce machining costs by minimizing trial and error, especially if the fixture could be 3D printed. This forms the central theme of this work. However, as the project evolved, it has been decided to narrow the scope by focusing on optimizing the design of machining fixtures, with the long-term goal of automating their generation through topology optimization and manufacturing them via additive manufacturing.

The next challenge was to determine what aspects of the fixture should be optimized. While there is extensive research on the subject, much of which is discussed in the literature review, it was decided that, before addressing the fixture design itself, it was crucial to first understand the optimal way to attach the workpiece to the fixture. Specifically, this work focuses on identifying the best clamping locations. To optimize these clamping points, a Python program

has been developed, utilizing a finite element discretization of the workpiece along with bar elements to model the clamping elements along its surface. This has been formulated as a mathematical optimization problem, where the design variables correspond to the densities of the bar elements, with the objective of minimizing static deformation in the workpiece during a given machining operation.

The primary goal, illustrated in Figure 1, of this thesis is to develop a Python program that takes as input a mesh of a given workpiece and a specified machining operation (typically milling), and outputs the optimal clamping locations. This approach provides valuable insights into the best positioning of fixture clamps and lays the groundwork for potential future research that could extend this optimization to other aspects of fixture design, such as the generation of optimized fixture topology and beyond.

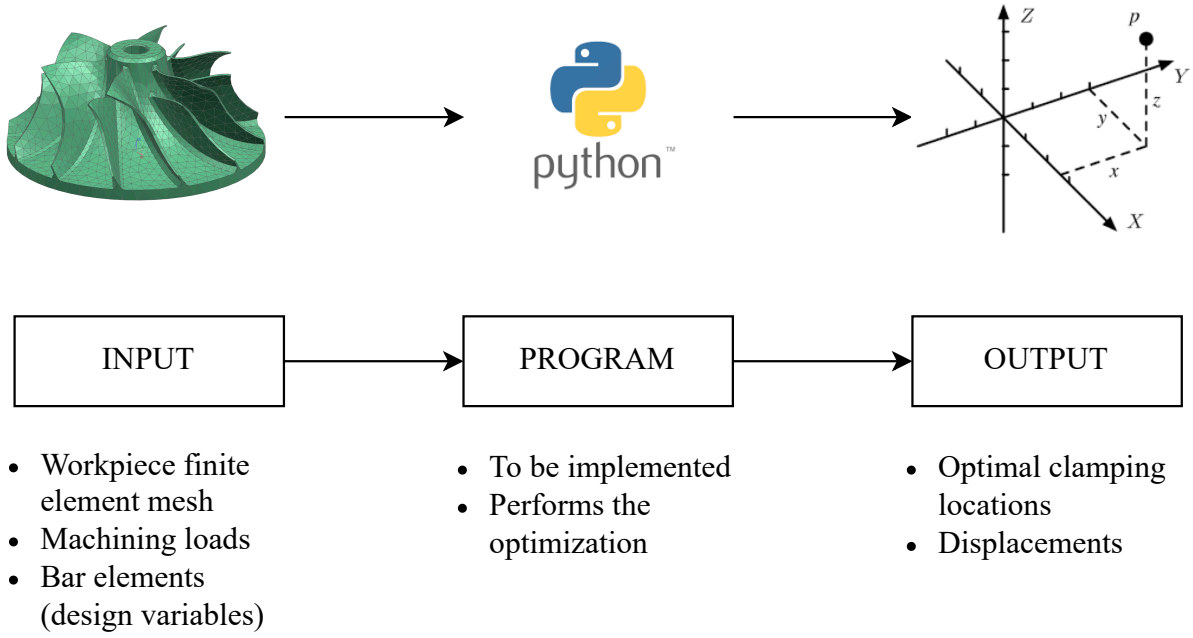


Figure 1: Schematic illustrating the general idea of the code, highlighting key inputs and outputs.

To keep the scope of this work manageable, the methodology has been implemented for workpiece meshes composed of 3D linear tetrahedral elements (T4). While it is acknowledged that these elements are not the most accurate, they offer significant advantages in terms of simplicity and geometric flexibility. This makes them especially convenient for manual mesh generation in software such as Siemens NX, where automated meshing options with T4 elements are generally available. Hence, the present work should be viewed as a proof-of-concept. Future extensions of the code could incorporate more advanced element types, such as second-order tetrahedral or hexahedral elements, to improve accuracy.

1.3 Optimization Problems

In many fields or applications, it is often necessary to determine a combination of variables that minimizes or maximizes a given output metric. In finance, for example, one could be concerned with establishing a portfolio that minimize the risk to reward ratio based on probabilistic measures. Production engineers, on the other hand, might want to find the best possible production parameters that minimize cost, while keeping production rates high. In mechanical engineering, one could ask which bar cross-sectional areas would minimize the global mass of a given truss structure, while ensuring that the maximum equivalent stress remains below a failure criterion.

These types of problems are referred to as *optimization problems* and can be expressed in terms of a so-called *objective function* $c_0 : \mathbb{R}^n \to \mathbb{R}$ that needs to be minimized with respect to *design variables* $\mathbf{x} = [x_1\ x_2\ ...\ x_n]^T \in \mathbb{R}^n$, generally submitted to some constraints. We write:

$$\begin{aligned} &\min_{\mathbf{x}\in\mathbb{R}^n} c_0(\mathbf{x}) \\ &\text{s.t.} \quad c_j(\mathbf{x}) \leq 0 \quad (j = 1, ..., m) \\ &\qquad \mathbf{x} \in \mathcal{S} \subset \mathbb{R}^n \end{aligned} \tag{1}$$

The objective (or cost) function $c_0(\mathbf{x})$ represents the quantity to be minimized, while the functions $c_j(\mathbf{x})$, for $j = 1, \ldots, m$, represent the constraint functions that ensure the feasibility of the design. Depending on the presence and type of these constraints (equality or inequality), the problem is classified as unconstrained or constrained. However, in general, any optimization problem can be reformulated into the standard form shown in Eq. 1 (Bruyneel and Tossings 2024).

KKT conditions

Since it will be relevant later in this thesis, the following text briefly outlines the necessary optimality conditions for general constrained problems, known as the Karush-Kuhn-Tucker (KKT) conditions. A candidate design point $\mathbf{x}^*$ associated with the problem in Eq. 1 must satisfy these conditions in order to be a potential optimal solution. However, because the KKT conditions are necessary but not sufficient, satisfying them does not guarantee optimality. Nonetheless, they are particularly useful for identifying points that can potentially be optimal solutions and are therefore often used as a convergence check in optimization algorithms (Bruyneel and Tossings 2024).

The condition is stated as follows. For the optimization problem given in Eq. 1, we associate the Lagrangian function:

$$L(\mathbf{x}, \lambda_j) = c_0(\mathbf{x}) + \sum_{j=1}^{m} \lambda_j c_j(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^n,\ \lambda_j \geq 0. \tag{2}$$

Assuming that c_0 and c_j are continuously differentiable (i.e., $\mathcal{C}^1$) at the optimal solution $\mathbf{x}^*$ and that the gradients of the active constraints at $\mathbf{x}^*$ are linearly independent, the KKT conditions

state that there exist $\lambda_1^*, \ldots, \lambda_m^*$ such that:

$$\nabla c_0(\mathbf{x}^*) + \sum_{j=1}^{m} \lambda_j^* \nabla c_j(\mathbf{x}^*) = 0, \tag{3}$$

and

$$\begin{cases} c_j(\mathbf{x}^*) \leq 0, \\ \lambda_j^* \geq 0, \quad (j = 1, \ldots, m). \\ \lambda_j^* c_j(\mathbf{x}^*) = 0 \end{cases}$$

1.4 Outline

This work begins with a literature review to examine existing research and developments on the optimization of fixture systems in machining. It covers a range of topics and concepts that have shaped the framework for this study, situating it within the broader context of fixture optimization. While this review provides a comprehensive overview, it is not intended to be exhaustive; rather, it serves as a foundational introduction to the subject. The insights gained from this review have been instrumental in identifying the direction for the optimization approach taken in this study.

In the third section, titled 'Methodology', the overall approach of the work is presented. This section outlines the initial strategies, describes the optimization problem, and justifies the choices made throughout the process. Additionally, the theoretical background necessary for understanding the methodology is provided. This includes an introduction to the finite element formalism, including the formulation of elementary matrices and their validation, as well as the use of sequential convex programming for solving the optimization problem once the objective and constraint functions have been defined. The section also covers the sensitivity analysis performed. Finally, the implementation in Python is discussed, with an overview of the code structure and its key components.

In the fourth section, the program developed in this work is applied to several case studies, and the results are analyzed and discussed. Initially, the program is tested on a simple cylinder to demonstrate its functionality. Following this, a more complex example of a turbine impeller is examined. The impact of various numerical and physical parameters is explored, and the importance of employing reduction methods to enhance efficiency is discussed.

This work concludes with a summary of key findings, followed by a discussion on potential improvements and future extensions of the program. Appendices are also included, containing, among other things, the Python code for the various modules of the program.

2 Literature Review

This section provides a summary and discussion of various approaches to optimizing fixtures, both in the context of machining and the assembly of critical parts. Among the explored methods, we look at the potential of using topology optimization for fixture design. Additionally, we examine research focused on optimizing clamping layouts and forces, as well as the optimization of contact interfaces.

2.1 Topology Optimization for Machining Fixtures

This section discusses topology optimization (TO) in the context of machining fixtures, beginning with an overview of the concept, followed by an explanation of the principles of TO, and concluding with a review of related research work in this area.

2.1.1 Overview

Recent advances in additive manufacturing (AM) technologies have made it possible to create more complex parts that were previously unfeasible. This naturally paves the way for topology optimization, a structural optimization method, which enables the automatic generation of ideal structures without the need for traditional design constraints.

There are three main categories of structural optimization: optimal sizing, shape optimization and topology optimization. The first concerns the optimization of specific design variables that define the geometry such as the cross-section surface area, the density, etc. It is important to note that the actual shape and topology are unchanged during optimal sizing, meaning that the finite element (FE) mesh stays the same. Hence, considering the example of a bar truss where the cross-section surface areas are to be optimized, no new bars will be added during the process. On the other hand, shape optimization involves changing the shape of pre-existing features (e.g. holes). The main difficulty here is that, at each step of the optimization algorithm, the mesh has to be adapted to the modified shape (mesh morphing) which makes this approach difficult to use in practice. For topology optimization, the algorithm can add or substract material within a given design space to reach the design goals, starting from imposed external loads and boundary conditions. This approach can lead to new design concepts and offer better results, especially in the context of structures submitted to vibrations, where engineers cannot always rely on their experience and intuition (Bruyneel and Tossings 2024, Poncelet 2005).

Designing structures can be a complex task due to large numbers of constraints and conflicting objectives. TO presents an alternative for engineers in which all the goals and constraints can be taken into account to obtain an organic shape for their design. In the static case, optimization algorithms are used to minimize compliance $C = \mathbf{f}^T\mathbf{u}$, minimize the largest displacement, or minimize maximum constraints in the structure while reducing the total mass. In the dynamic case, the goal is generally to maximize eigenfrequencies to avoid resonance while reducing the mass. Oftentimes, optimization problem set the dynamic condition as a constraint on the first eigenfrequencies instead of maximizing the eigenfrequencies per se (Bruyneel and Tossings 2024).

There exists different tools to perform topology optimization such as Altair OptiStruct, Siemens NX TOPOL, etc. The latter is a program that was developed by the SAMTECH Company in collaboration with the University of Liège. It has been used in different applications such as for the design of fixtures for testing structures on electrodynamic shakers (dynamic) as well as for the design of a new engine pylon for Airbus (static) (Poncelet 2005).

2.1.2 Principle

The algorithm for TO is divided into a series of different steps. The process generally starts with a 3D model of the design space, i.e. the region in space in which a solution will be looked for, any point outside this domain will not intervene in the final design. This model is then discretized in a finite element mesh where each element is assigned a density $\rho_i = x_i\rho_0$, where ρ_0 is the density of the initial material and $0 \leq x_i \leq 1$ are the design variables taking values between 0 and 1 to designate the absence or presence of material for each finite element respectively. Since optimization algorithms (e.g. MMA or CONLIN) evolve around continuous variables, it is more convenient to allow x_i to take continuous values between 0 and 1 rather than imposing a binary choice. The user is free to choose the objective function that best fits his needs, but the most prominent choice is the minimization of the compliance $C = \mathbf{f}^T\mathbf{u}$ while setting an upper bound on the total mass of the structure (Bruyneel and Tossings 2024).

To force the algorithm towards either one of the two values 0 or 1, homogenization methods are used. The most commonly used is the SIMP method standing for *Simply Isotropic Material with Penalization*, where the density and the Young's modulus are expressed as follows:

$$\rho_i = x_i\rho_0 \qquad \text{and} \qquad E_i = x_i^p E_0. \tag{4}$$

The penalization parameter $p > 1$ can be tuned to modify the slope of the Young's modulus as a function of the design variable. To understand why x_i will be pushed towards its boundaries of 0 or 1, consider a current design where $x_i = 0.8$. From the perspective of the optimization algorithm, further increasing the variable x_i will increase the density on one hand, but it will also increase the stiffness by a larger amount on the other hand, thus pushing x_i towards 1. Depending on the slope of $E_i/E_0 = x_i^p$, i.e. on the penalization p, the relative increase will be weaker or stronger. The opposite is true close to 0, where there is no reason for the algorithm to add more material because the gain in stiffness is negligible. Parasitic solutions, such as checkerboard patterns, may appear during the optimization process. These represent unphysical solutions and can be avoided by using special filtering techniques (Bruyneel and Tossings 2024, Duysinx 2019).

2.1.3 Related research work

Research on TO in the context of fixture design is scarce. While there are numerous applications in a variety of different fields, its use in machining fixture is not widespread. Some related examples have been mentioned below.

Burri (2023) proposes an approach for static reinforcement and vibration reduction of structures by using TO. The method consists in optimizing a design domain Ω_2 around a given fixed

structure Ω_1, such that the resultant structure meets specific criteria regarding its deformation under a given load or eigenfrequencies. Different objective function have thus been proposed and the method has been applied on beams and CubeSat-like structures.

Calabrese et al. (2017) discuss and investigate the optimization of a fixture for a thin-walled low pressure turbine casing (LPT) using topology optimization. The authors have approached the problem by first considering a 2D FE model with shell elements, where axial and radial loads have been applied on the inner part of the casing. The study has then been extended to 3D solid elements for a quarter of the workpiece to reduce computation costs. The objective function considered is to minimize the overall compliance, i.e. to maximize the stiffness of the fixture. The possibility of optimizing specific design regions successively has been investigated, but did not lead to a manufacturable solution. Also, different types of loading cases have been considered with more or less nodes on which forces were applied. The largest reduction in mass and the most feasible outcome has been obtained by dividing the overall design space into 6 regions, such that each of these regions feature different mass fractions. To verify if the stiffness of the structure could further be augmented, 3 types of lattice-structures have been placed in void areas of the obtained fixture. However, the static analysis has not shown any change in the maximum displacement, thus confirming the results from TO. It is questionable whether the obtained fixture is suitable for real applications, since the workpiece-fixture contact region is a surface of revolution (hyperstaticity), and it has not been mentioned how exactly clamping forces would be applied.

L. Rakotondrainibe et al. (2020) have investigated the possibility of coupling TO of a structure with the optimization of its connections simultaneously. This approach enhances classical TO, where the connections are assumed fixed, by also considering the important effect of connection locations and number, which is one step closer to the optimization of assembled systems. Instead of using the SIMP method, the authors have considered the level-set method with Hadamard's boundary variation method. The considered test case is an alternator and an air conditioning compressor bounded together by a bracket; the latter has been optimized. At first, supports have been assumed rigid and the design variables that have been selected are: the center posisions $\mathbf{x}_i$ of the circular supports and the design volume Ω. The optimization problem consists of minimizing the volume $V(\Omega)$ of the bracket while imposing a constraint on the compliance $C(\Omega, \mathbf{x}_i)$, which writes as follows:

$$\begin{aligned} &\min_{(\Omega,\mathbf{x}_i)\in\mathcal{U}_{ad}} V(\Omega) \\ &s.t. \quad C(\Omega,\mathbf{x}_i) \leq (1+\eta)C_0 \ , \end{aligned} \tag{5}$$

where $\mathcal{U}_{ad}$ is a set of admissible solutions, η is a tunable parameter and C_0 is the compliance of the full volume bracket. A gradient-based algorithm has been used to solve the problem. Interested readers are referred to the original text for more details. Then, L. Rakotondrainibe et al. have introduced a model for an idealized bolt-like connection. This analytical model (shown in Figure 2) enables to take into account the transmission of efforts between two structures without having to implement a fine FE mesh of the bolts, as it is not of interest here. The head and thread of the bolt are replaced by two spherical domains ω_A and ω_B respectively, both of radius ρr_e, where ρ is a scale factor. These are separated by a length l and are linked by a relationship of the form $F = KL_s$, where F is a generalized force, K the rigidity matrix and

L_s a generalized lengthening of the bolt. In the present case, L_s is the difference between the average displacements along the axis of the spring in ω_A and ω_B. The energy functional of the system is expressed as follows:

$$E(\boldsymbol{\phi}) = \frac{1}{2}\int_{\Omega_p} \sigma(\boldsymbol{\phi}) : \varepsilon(\boldsymbol{\phi}) dV - \int_{\Gamma_N} \mathbf{g} \cdot \boldsymbol{\phi} dS + \frac{1}{2} K(\rho) L_s^2 \ , \tag{6}$$

where the displacement field follows from the principle of minimum potential energy within the space of admissible displacements $\mathcal{V}$:

$$E(\mathbf{u}) = \min_{\phi \in \mathcal{V}} E(\boldsymbol{\phi}) \ . \tag{7}$$

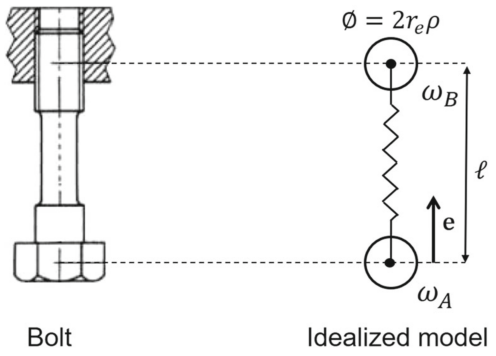


Figure 2: Idealized bolt-like connection (Rakotondrainibe 2020).

2.1.4 Discussion

The initial idea of this final year's work has been to investigate ways to generate topologically optimized fixtures for the machining of workpieces. As such, the goal would have been to input an arbitrary workpiece and the output of the process would be a fixture that could be printed using AM technologies. Different difficulties can be mentioned. Number 1, the entire structure cannot be printed as a single piece because we need to account for the locating and clamping of the workpiece, which is done with external elements such as pins, screws, clamps, etc. Number 2, there are many clamping techniques that the author might not know about due to lack of experience in the field, thus limiting the scope of possible solutions. Number 3, the seemingly infinite amount of workpieces and machining situations make it difficult to introduce any type of generality for the optimization of fixtures, let alone topology optimization. Number 4, for more simple or manageable geometries, TO does not seem like a promising option, since the fixture layout most often than not consists of a flat plate on which lays the workpiece plus some clamps for maintaining it in position. An example of this is shown in Figure 3, where the potential for optimization of the fixture itself is limited.

Number 5, the great variability of loading cases that occur during the machining process make it unclear as to how one should approach the definition of external forces. Number 6, if deformations and/or vibrations during machining are to be reduced, it would be best to keep the general stiffness of the fixture as large as possible, meaning that mass reduction under TO

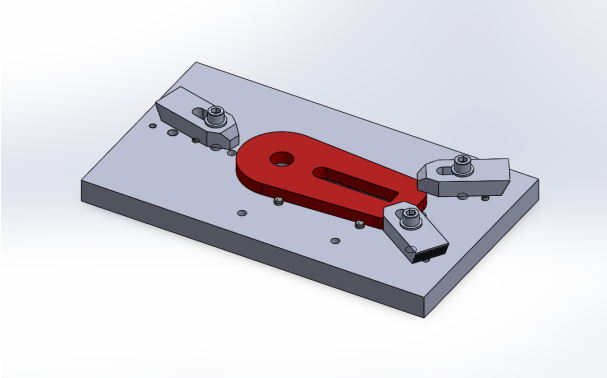


Figure 3: Example of a simple workpiece-fixture system.

makes little sense. Number 7, under TO, a design space and a loading case are required, but the design space itself is, most often than not, odly shaped with non-optimizable areas for the attachment of the clamping elements. Also, the clamping forces must be taken into account since they can be the dominating force. In addition, considering that the clamping concept, the design space and the loads acting on the fixture are all well defined, the solution yielded by TO will be valid for a specific loading case and could not be suited for other external force. Number 8, studying the dynamic behavior of the workpiece-fixture system requires to make assumptions regarding the interactions at the contact interfaces: Do we assume the whole system is a single solid body, or do we model the contact force and friction?

Hence, from all the above mentioned, it seems very challenging to introduce any type of generality for the proposal of fixture optimization using TO. Another alternative to optimizing the topology of the fixture is the optimization of clamping and locating locations, as well as their number. One could also consider optimizing the clamping forces, the contact interfaces, etc. These options have been investigated in the following sections.

2.2 Fixture Layout Optimization

This section focuses on the state-of-the-art in fixture layout optimization. While some approaches enable the simultaneous optimization of both clamping element locations and clamping force magnitudes, for clarity, these topics have been addressed separately here. We first explore layout optimization, followed by the optimization of force magnitudes in a subsequent section. The structure of this section is as follows: we begin with an overview, then delve into machine-learning approaches, discuss intelligent fixtures, and conclude with an exploration of soldering point modeling and optimization, highlighting their potential for improving clamping location optimization.

2.2.1 Overview

C. Liu et al. (2024) have presented a scientific review on optimization of fixture layouts for the assembly of thin-walled parts. These compliant parts are widely used in many fields such as aerospace, the auto industry, shipbuilding, etc. to build outer shells, like body in white (BIW). Optimizing fixture layout can help reduce manufacturing costs, reduce the deformation and enhance assembly precision and quality. The review discusses the current situation by

classifying, comparing and analyzing the literature. The topic of fixture layout optimization has been divided into 3 main branches: selection of optimization objectives, modeling methods and algorithms for optimization. The first step in layout optimization is the determination of the objective function, where two main types have been identified; these are in-plane variations and out-of-plane deformations. Despite the great variety of optimization objectives, researches often choose to minimize the total strain energy, expressed as

$$U = \frac{1}{2} \int_V \sigma^T \varepsilon dV. \tag{8}$$

Regarding the modeling methods, the authors mention two categories, which are: mechanism-based and data-based methods. Mechanism-based approaches include the Jacobian matrix method, the state-space method and the FEM; where FEM is the predominant choice due to its accuracy and availability in commercial software (e.g. ANSYS or ABAQUS). Since it is difficult to incorporate the FEA within optimization algorithms for evaluating the cost function, research generally focuses more on data-based methods. These allow for the prediction of mechanical behavior as a function of many parameters and for large scale problems, which would not be possible in mechanism-base modeling. C Liu et al. have emphasized the need for further research to improve modeling accuracy in data-based approaches. When it comes to the optimization algorithms, a distinction is made between two classes: nonlinear and heuristic algorithms. While nonlinear algorithms are best suited for small and medium sized problems, heuristic algorithms (e.g. genetic algorithms, particle swarm optimization) are an efficient alternative for large scale problems. The review ends with suggestions for future research questions such as "How to optimize the position and number of fixtures at the same time?".

2.2.2 Machine-learning approaches

Machine-learning, particularly artificial neural networks (ANNs), has become widely used in fixture layout optimization during machining. Instead of relying on FEA simulations to evaluate the objective function, e.g. the maximum displacement in the workpiece, researchers use ANNs trained on random clamping configurations. By performing FEM simulations on these configurations, they derive metrics like displacement, which are then used to train the ANN. The trained model can predict the maximum displacement for any given clamping setup and machining force. These approaches belong to data-based methods. Some application examples have been listed below.

The authors in (Vukelic 2016) proposed an integral system for optimal fixture design using what works best from previous research, which includes methods evolving around FEA, ANNs, genetic algorithms (GA), etc. The system uses a variety of input information regarding the workpiece geometry and characteristics as well as information about the machining context to output an optimal solution that is generated from known standardized elements. Different modules form a global system structure (see Figure 4): a module for input information, a module for optimization, a module for fixture elements selection and a module for output information. The article motivates this approach with the potential reduction in cost during the production process. It should be noted that the large volume of input information can make the system challenging for end users to navigate and utilize effectively.

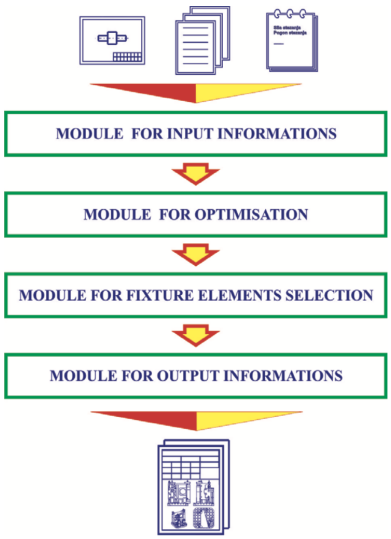


Figure 1 Global system structure

Figure 4: Global system structure (Vukelic 2016).

In (Feng 2021), Q. Feng et al. presented a new method for optimizing fixture layouts using machine-learning. The method consists in randomly generating clamping configurations, performing FE simulations in each case, and extracting relevant target variables to train the XGBoost model, a ML algorithm developed by T. Chen et al. (Chen 2016). These steps have been applied to a clamping situation from (Möhring 2016) (see Figure 5), assuming linear isotropy and frictionless contact between the workpiece and the clamping elements, for 100 FE simulations in Siemens NX. The variables that have been retained are maximum displacement and the first natural frequency; only an exemplary position has been considered for the cutting forces. It has been shown that the XGBoost model could predict the maximum deflection in the workpiece for any clamping configuration and it has led to a quasi-optimal layout.

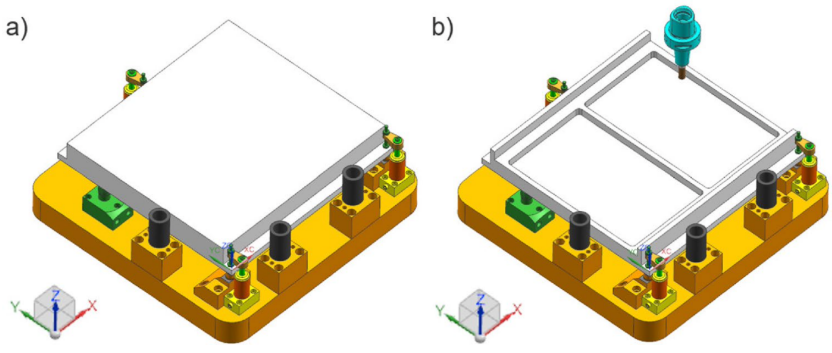


Figure 5: 3D models of the clamping system used in (Möhring 2016) and (Feng 2021).

The authors of (Quazani 2022) investigated the possibility of determining optimal positioning of locators and clamps through the use of machine-learning methods. A 2D rectangular sheet has been considered as workpiece with a 2-1 locating scheme (see Figure 6). The approach

consists in first generating random locating and clamping points, which have then been used to perform FE simulations in ANSYS. The maximum elastic deformation in the workpiece during the machining process has been retained to train machine-learning methods in MATLAB. Evolutionary algorithms have then been used to extract the ideal positioning. To compare and validate their method, the authors have referenced a previous work on this topic (Hoffman 2012). Note that the machining forces have been calculated prior to the FE simulations and that chip removal has been considered through the element death method.

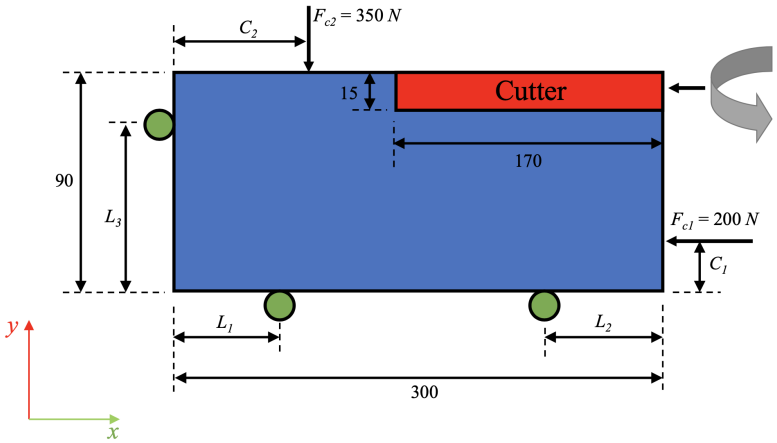


Figure 6: Two-dimensional 2-1 fixture locating scheme (Quazani 2022).

2.2.3 Intelligent fixtures

In machining, intelligent fixtures refer to workholding devices that integrate sensors, actuators, and digital controls to enhance precision and automation during the manufacturing process. These fixtures can adjust automatically to variations in the workpiece, monitor machining conditions, and provide real-time feedback to optimize performance, improve accuracy, and reduce human intervention.

A review on the state of intelligent fixture design using sensor and actuator systems has been presented by H.-C. Möhring et al. in (Möhring 2016). Different concepts resulting from the European research project INTEFIX ("Intelligent Fixtures for the manufacturing of low rigidity components") have been introduced. Among those, a prototype for applying counter excitations to limit regenerative chatter during machining of impeller blades, shown in Figure 7, has been discussed. This has been done by playing with the rotation degree of freedom around the central axis of the piece. Experiments have shown a reduction of surface waviness compared to conventional methods. Another example consists in using clamping pistons to compensate workpiece distortions from residual stresses in thin-walled aluminum parts. H.-C. Möhring et al. have highlighted the importance of considering the entire process-workpiece-fixture system in fixture design and optimization.

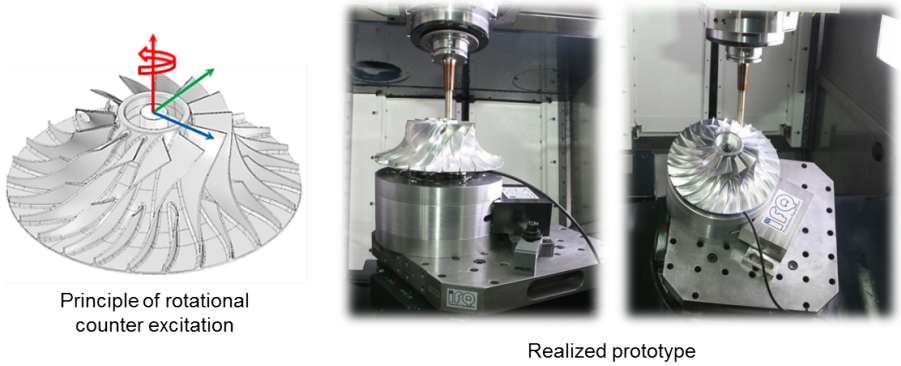


Figure 7: Intelligent impeller fixture prototype (Möhring 2016).

2.2.4 Soldering points modeling

Soldering points can be modeled in different ways. The figure below shows examples of steel sheets welded together using two soldering points on both edges (Beckers 1997).

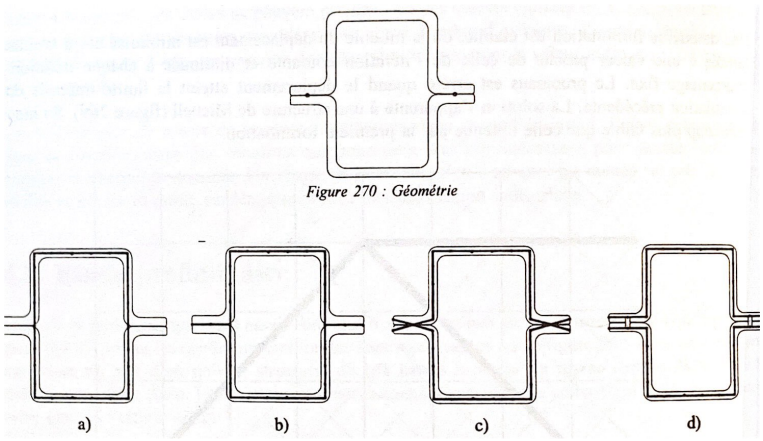


Figure 8: Modeling of soldering points (Beckers 1997).

As shown in Figure 8, the soldering points can be modeled in the following ways: (a) by neglecting the edges and attaching both sides at common nodes, (b) with the edges already fused together, (c) with the edges connected via a node at the actual soldering location, or (d) by introducing a bar (or beam) element between the two separate steel sheets. Note that the first two modeling methods lead to a geometry that is too rigid, but offer simplicity in modeling. The fourth approach was used in (Beckers 1997) to determine an optimal number of soldering points in the context of automobile manufacturing. The author's method involves introducing beam elements at each node susceptible to soldering and selecting the beam cross-sections as design variables. The optimization algorithm is then established by minimizing a metric for the overall displacement under a given loading case, while keeping mass as an upper-bound constraint.

Determining ideal locator and/or clamp contact locations is part of the overall optimization of fixtures. The method described above can be used to derive a locating layout under the

assumption that contact interfaces between the workpiece and fixtures can be modeled as soldering points. In this case, each node on the concerned surfaces can have bar or beam elements added, which correspond to welding points that will progressively be removed during optimization, leaving only the most useful ones.

2.3 Clamping Force Optimization

In this section, various approaches for optimizing the magnitude of clamping forces in fixture design during machining and assembly processes are explored. Optimizing clamping forces is essential for minimizing workpiece displacement and ensuring precision during machining. Several methods, including analytical techniques and multi-objective optimization approaches, are discussed. These methods focus on determining the optimal positions and magnitudes of clamping forces to achieve desired outcomes such as minimizing displacement, stress, and ensuring structural integrity. The section highlights both traditional analytical models and more advanced computational methods, including FEA and GAs, for efficient force optimization.

2.3.1 Analytical methods

In their paper on clamping force optimization, B. Li and S.N. Melkote (Li and Melkote 2001) have presented a new method for determining the optimimum clamping forces that minimize the workpiece locating error of a 3-2-1 fixture system (see Figure 9). Key assumptions include a rigid body modeling of the workpiece, quasi-static machining and clamping loads as well as an elastic contact modeling of the workpiece-fixture contact. Contact regions have been represented by lumped contact stiffnesses in x, y and z directions. The stiffness values have been obtained by linearizing the Hertz contact problem between a spherical tip (locator and clamp) and a half-plane (workpiece). The clamping forces have been assumed constant, which corresponds to hydraulic/pneumatic clamps. The authors have then been able to formulate the problem as a multi-objective constrained optimization problem by minimizing the total complementary energy and the L_2-norm of the resultant force.

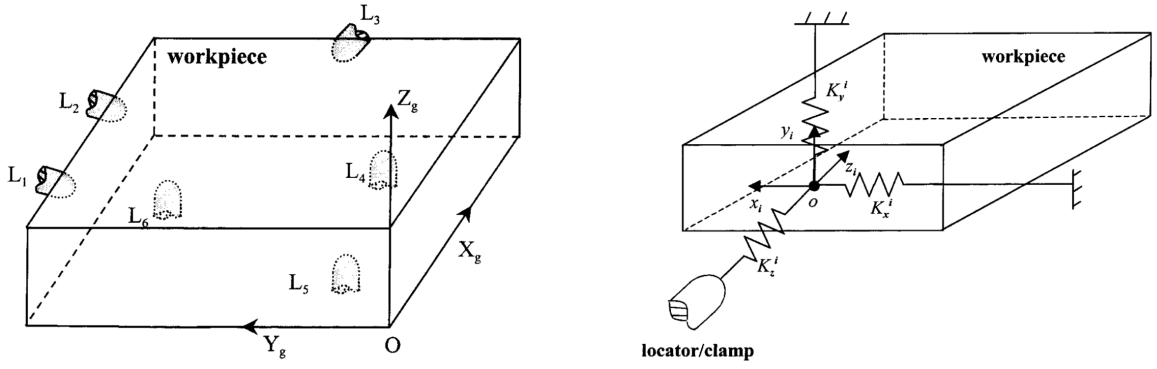


Figure 9: 3-2-1 fixture layout of a prismatic workpiece with elastic contact modeling (Li and Melote 2001).

Xiong et al. (2008) have proposed a general method to determine the optimal clamping force positions and magnitudes based on a frictional elastic contact model. At first, the authors have derived compatibility equations linking the displacement of the workpiece to the

elastic deformations at workpiece-fixture contact interfaces. Then, using Hertz contact theory, the relationship between the contact force and the local elastic deformation has been determined. The workpiece has been assumed as a rigid-body where only contact areas are elastically deformable. Xiong et al. stress the importance of considering the nonlinear nature of these contact interactions, unlike what has been done in (Li and Melkote 2001). The optimization problem consists of minimizing the norm of the elastic deformations at contact points, while imposing static equilibrium, conditions of compatible deformations, a threshold on the tangential contact forces (dry friction law) and a feasible clamping domain.

In (Selvakumar 2010), the optimal clamping forces for a 3-2-1 fixture layout of a simple prismatic workpiece have been determined analytically by balancing the forces and moments and by using Coulomb's friction law. The geometry has been taken from (Li and Melkote 2001) and is shown in Figure 9. The methodology proposed consists in computing the minimum required clamping forces so that the normal contact forces at the fixture elements remain below a static friction threshold. This approaches has been justified by the fact that the workpiece deformation is minimum when clamping forces are minimum. Different machining load cases have been considered and the retained clamping forces correspond to the maximum among these computed values. Other parameters such as number of locators and layout have not been considered.

2.3.2 Other methods

In (Cioata 2017), the locations and magnitudes of clamping forces are optimized using a multi-objective method to minimize the maximum total displacement on a selected edge and to minimize the maximum equivalent stress in the workpiece in ANSYS. The system is a 3-2-1 locating scheme of a 225x122x122 [mm] prismatic workpiece with a canal milling operation on the upper face. The design variables that have been selected are: the coordinates of the clamping force locations and their magnitudes. Constraints on the locator contact forces have also been imposed. These require that the workpiece material yield stress σ_y is not reached and that the minimum contact force remains above zero and that tangential loads remain below the static friction threshold ($f_{c,\min} \geq 0$ and $f_{c,\min} \cdot \mu \geq \sqrt{q_x^2 + q_y^2}$, where q_x, q_y are tangential contact forces and μ is a static friction coefficient).

In (Zhang 2021), Zhang et al. have proposed a new procedure for optimizing the clamping force layout and magnitudes for the assembly of composite structures. Gaps between mating surfaces of composite assemblies are inevitable due to limited manufacturing precision. Hence, clamping forces must be applied to force contact, which leads to internal stress distributions that can damage the composite materials. Coordinates of clamping positions and magnitudes of clamping forces have been chosen as design variables. Two objectives functions impose the maximum gap elimination rate and the minimum overall clamping force value. The former has been expressed using discrete point cloud models of the real surfaces measured via laser scanner. A criterion for the internal damage of the composite has also been considered as a constraint on the design variables. A genetic algorithm together with FEA simulations have been used to solve the optimization problem for the assembly of an upper panel on the skeleton of a wing box structure shown in Figure 10. Result accuracy has been verified by physical experiments.

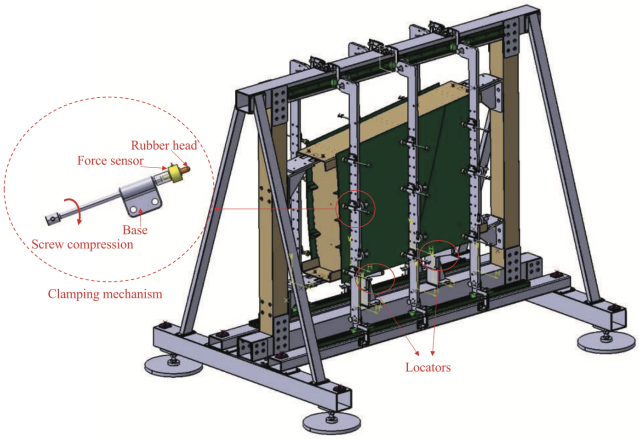


Figure 10: Wing box upper panel assembly (Zhang 2021).

2.4 Contact Interface Optimization

Conventional clamping elements generally present flat contact interfaces with the workpiece. To ensure that the workpiece remains in place with small displacements during machining, the clamping elements must withstand tangential loads resulting from external efforts. The larger the clamping force, the larger the tangential load capacity. However, in some cases, such as for thin-walled parts, the clamping forces cannot be too large or the deformation of the workpiece will be greater then can be tolerated. As a general rule, machining errors directly depend on the magnitude of the displacements at the workpiece-fixture contact interface. P.M. Todorovic et al. (Todorovic 2014) have thus proposed a specially designed clamping element with round cutting insert which has been shown to present superior clamping performance. As such, it has been shown experimentally that for a same clamping force, the tangential displacement at the contact interface is significantly reduced. Figure 11 shows the difference in performance between the flat conventional clamping element and the newly designed one.

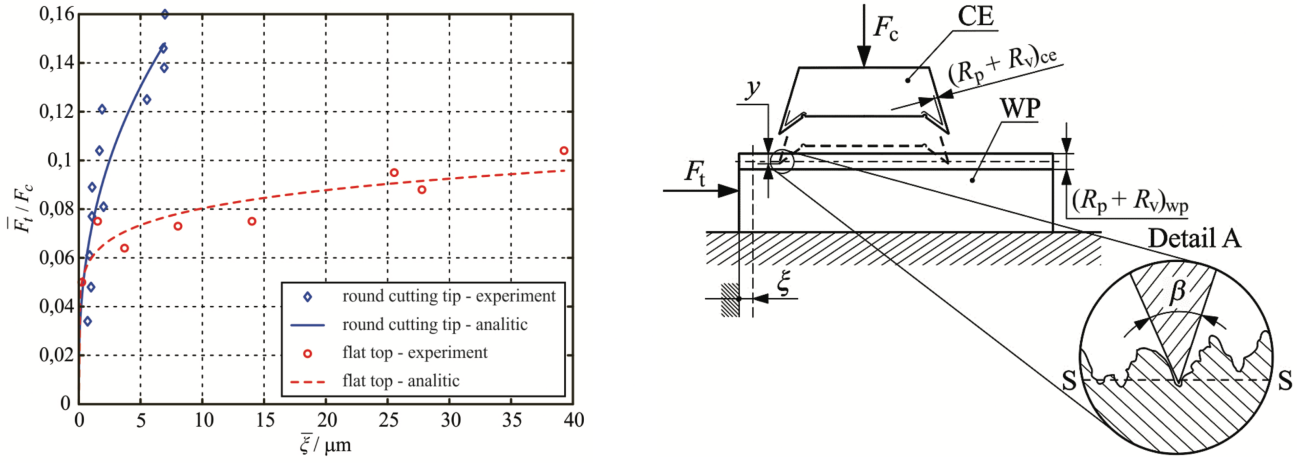


Figure 11: Comparison between the performance of a flat top clamping element and an element with round cutting tip (Todorovic 2014).

2.5 Solutions for Structural Optimization

There are numerous software options available for performing structural optimization, each distinguished by its capabilities in areas such as objective function control, optimization algorithm flexibility, load case management, mesh refinement, and constraint handling. Some solutions offer greater precision and adaptability than others, but the ideal choice depends on factors like budget, required functionality, and ease of implementation. A tool with a steep learning curve may be discouraging and demand significant time for engineers to develop the necessary expertise. Below, we will discuss different software options to provide an overview of potential solutions. It is important to note that different software can produce varying results, even when using the same input parameters (Benoist 2020).

One notable option is Siemens NX Topol, a program originally developed by the SAMCEF Company in collaboration with the University of Liège. It offers significant flexibility in terms of mesh generation, boundary conditions, and load case definition. However, its selection of objective functions is somewhat limited, allowing users to minimize compliance, maximize the first n resonance frequencies, or minimize the maximum constraint. Regarding optimization algorithms, users can choose between the Method of Moving Asymptotes (MMA) and CONLIN, as well as between the SIMP and RAMP material interpolation methods. Additionally, parameters such as the occupied volume fraction and density threshold are adjustable, providing further control over the optimization process. NX Topol has been successfully applied in various engineering scenarios, including the design of fixtures for testing structures on electrodynamic shakers (a dynamic example) and the development of a new engine pylon for Airbus (a static example) (Poncelet 2005).

Another alternative is Altair Inspire, a simulation-driven design software primarily used for structural, thermal, and motion analysis, as well as optimization. It supports various types of optimization, including different objective functions, mass targets, frequency constraints, and thickness constraints. Notably, Inspire serves as a more user-friendly and streamlined version of Altair OptiStruct, making it accessible to designers and engineers while still offering powerful optimization capabilities (Benoist 2020).

Another alternative is Abaqus Tosca, which offers more refined mesh control compared to Altair Inspire and allows for the combination of multiple objective functions. It provides greater flexibility in optimizing algorithms, including the ability to define parameterized convergence criteria. Additionally, users can choose between SIMP and RAMP for material interpolation, enabling further customization of the optimization process (Benoist 2020).

Since most of the aforementioned options impose certain limitations, whether in terms of the optimization methods supported, the definition of the objective function, or the flexibility of the framework, an alternative approach is to implement a custom structural optimization program in Python. This option offers the greatest degree of flexibility and customizability, allowing full control over the formulation of objective and constraint functions, the choice of optimization techniques, and the overall adaptability of the tool. Moreover, developing such a program presents an excellent opportunity to demonstrate technical competence and initiative, particularly in the context of a master's thesis.

3 Methodology

This section presents the methodology used to optimize the clamping locations. It begins by outlining the initial strategies considered, along with their respective limitations. The focus then shifts to the selected optimization approach, detailing the formulation of the problem, including the objective function, constraints, and relevant modeling aspects such as sensitivity analysis and the development of elementary stiffness and mass matrices. Key assumptions and limitations of the chosen method are also discussed. The section concludes with an overview of the implementation of the optimization algorithm in Python.

3.1 Exploring Initial Strategies in Clamping Optimization

The following text outlines an initial concept for optimizing the clamping of a workpiece subjected to machining loads. This strategy has however not been adopted for reasons outlined below.

3.1.1 Initial strategy

Given a finite element (FE) mesh of the workpiece and user-defined clamping surfaces, which must be flat, the objective is to determine the optimal combination of clamping force and position. The clamping force is assumed to act perpendicular to the clamping surface. The goal is to minimize the maximum displacement within the part while ensuring that the clamping forces are sufficient to prevent movement during machining. This is achieved by enforcing static friction conditions at the locator nodes, which are modeled as clamped points.

The optimization problem is defined as follows. The design variables $\mathbf{x}$ are the coordinates of the locations where clamping forces are applied. The constraints include (i) ensuring that these locations lie within the defined clamping surfaces, satisfying the plane equation $A_i x + B_i y + C_i z + D_i = 0$, and (ii) maintaining the points within the clamp surface boundaries, which introduce additional constraints of the form $a_j x + b_j y + c_j z + d_j \geq 0$. However, this requires that the clamping surfaces as defined by the boundary edges of the elements must be convex, as shown in Figure 12.

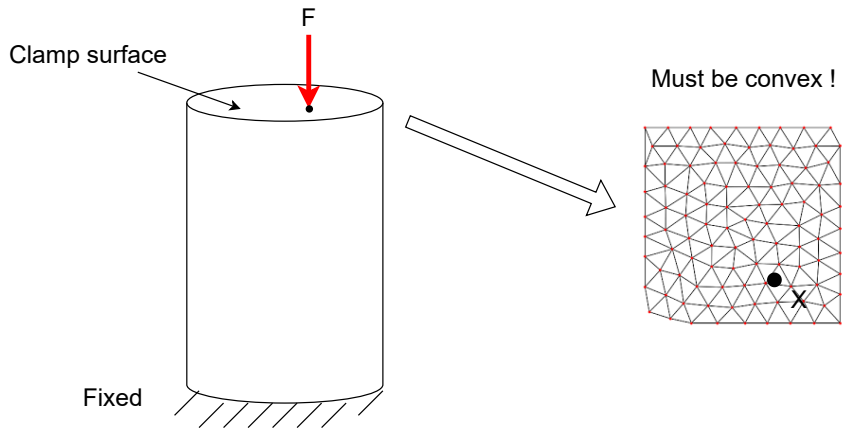


Figure 12: Schematic of the initial strategy for optimizing the clamping locations.

To evaluate the objective function, the displacement vector $\mathbf{q}$ is obtained by solving the static equilibrium $\mathbf{Kq} = \mathbf{f}$, where $\mathbf{K}$ is the constant structural stiffness matrix (which already incorporates the fixed locator nodes), and $\mathbf{f}$ is the external force vector, which depends on the design variables (the coordinates of the clamping points). For given design variables, the force vector $\mathbf{f}$ can be determined by adding the contributions of all concerned elementary force vectors $\mathbf{f}^e$ resulting from the clamping forces. These elementary forces are obtained by examining the consistent nodal loading resulting from a surface traction (Ponthot 2022):

$$\mathbf{f}^e = \int_S \mathbf{N}^T \bar{\mathbf{t}} dS \ , \tag{9}$$

where $\mathbf{N}$ is the shape function matrix and $\bar{\mathbf{t}}$ is an arbitrary surface traction. Since the clamping forces are modelled as point loads, the integral in Eq. 9 is zero everywhere on the surface, except where the force is applied. Thus, denoting by $\mathbf{F}$ the clamping force vector, we have

$$\mathbf{f}^e = \mathbf{N}^T \mathbf{F}. \tag{10}$$

Given what has been said above, the optimization problem is expressed as follows:

$$\begin{aligned} &\min_{\mathbf{x}} f(\mathbf{x}) = \max ||\mathbf{u}|| \\ &\text{s.t.} \quad h_i(\mathbf{x}) = A_i x + B_i y + c_i z + D_i = 0 \quad (i = 1, ..., n) \\ &\qquad g_{ij}(\mathbf{x}) = a_{ij} x + b_{ij} y + c_{ij} z + d_{ij} \geq 0 \quad (j = 1, ..., N_i) \end{aligned} \tag{11}$$

where $f(\mathbf{x}) = \max ||\mathbf{u}||$ is the cost function, i.e. the maximum displacement within the workpiece (or another metric based on the displacement), $h_i(\mathbf{x})$ are equality constraints corresponding to the cartesian equations of the clamping surfaces and $g_{ij}(\mathbf{x})$ are inequality constraints imposing that the point locations remain within their respective convex clamp surface. Note that the subscript i is associated to the clamp surface, whereas j takes value from 1 to N_i depending on the number of edges of the i-th surface.

3.1.2 Shortcomings and limitations

Several shortcomings can be identified with this approach. First, modeling clamping forces directly is not ideal; using structural elements attached at the outer nodes such as bars or beams would provide a more accurate representation. Second, this method lacks precision because the clamping forces are applied as point loads, which are then translated into nodal forces, an approximation that only ensures energy equivalence rather than a truly realistic load distribution. This is illustrated schematically in Figure 13.

Also, using bar or beam elements offers greater flexibility since their placement is not restricted to flat surfaces; they can be positioned freely as needed. In contrast, the current approach requires flat surfaces for clamping because planes are simple geometries to which a normal vector can be easily assigned. While, in theory, it may be possible to apply clamping forces on curved surfaces, this would require representing the surface with an equation $h(x) = 0$, defining a normal force vector for each point on this surface, and resolving uncertainties in determining the corresponding elementary force vector $\mathbf{f}^e$.

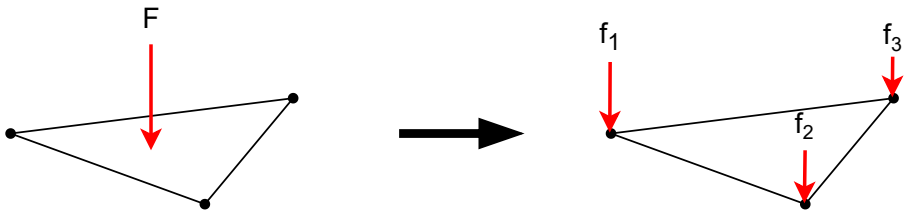


Figure 13: Schematic representation of energy consistent nodal loading.

It is important to note that modeling the clamping of a workpiece using external point loads does not provide insight into the system's vibration modes. To compute eigenmodes that account for clamping forces, these forces must be represented within a stiffness matrix. However, in this case, the clamping forces are modeled as a constant vector $\mathbf{f}$, and therefore do not contribute to the stiffness matrix. Additionally, the approach is further constrained by the requirement that clamping surfaces be convex, which can be restrictive in many cases. These limitations do not apply when using bars or beams as discussed hereafter.

3.2 Overall Approach

The chosen method for optimizing the clamping locations is described in the following section. This includes a description of the optimization problem as well as a discussion of assumptions and limitations.

3.2.1 Description of the optimization problem

The location of fixture clamping elements has been optimized by modeling the contact between the workpiece and the fixture as soldering points. For this purpose, bar elements have been placed at each node of the clamping surfaces, as shown in Figure 14. All bars share the same cross-sectional area $A_i = A$ and their Young's modulus E_i have been selected as variables. To define the absence or presence of material within each bar element, new design variables x_i have been introduced; these take values of 0 if no material is present and values of 1 for a full bar. Hence, the Young's modulus in each bar can be expressed as $E_i = x_i E_0$, where E_0 corresponds to the Young's modulus of the initial material. If $x_i = 0$, one has $E_i = 0$ and, if $x_i = 1$, one has $E_i = E_0$.

Since optimization problems evolve around continuous variables, the discrete variables x_i must be made continuous through a homogenization law. This has been done via the SIMP law:

$$\begin{aligned} E_i &= x_i^p E_0 \\ \rho_i &= x_i \rho_0 \end{aligned} \tag{12}$$

where $p > 1$ is a penalization parameter which can be tuned by the user. As mentioned in Section 2.1, this homogenization law allows the design variables x_i to vary continuously within $0 \leq x_i \leq 1$, while forcing them towards either 0 or 1.

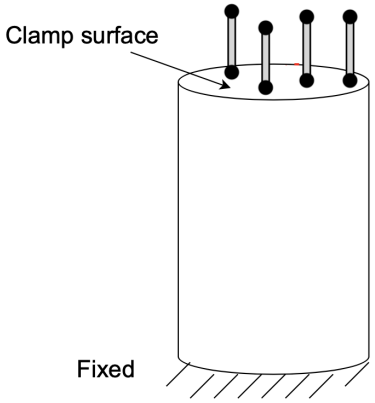


Figure 14: Schematic of the current strategy for optimizing the clamping locations.

Every optimization problem requires a metric to compare different designs between each other and come up with the optimal solution. This metric is achieved by the objective function $c_0(\mathbf{x})$, where $\mathbf{x} = [x_1 \quad x_2 \quad ... \quad x_n]^T$, which defines the 'goal' of the optimization process. This work is interested in finding the optimal clamping locations for a given workpiece and machining process that will ensure a high quality end product for which mechanical tolerances are respected. For this to be the case, one must ensure that the workpiece deforms as little as possible under the effect external forces. Therefore, the expression of the objective function will necessarily include the displacements $\mathbf{q}$ within the part.

Different expressions for the objective function can be considered. One might, for example, try to minimize the compliance $C = \mathbf{f}^T\mathbf{q}$ which is a common choice in the literature. However, in the context of machining, the external forces $\mathbf{f}$ result from the cutting tool's action on the workpiece and are modelled by applying a nodal load on specific points of the finite element mesh. Minimizing the compliance by playing with the locations of clamping points (modelled as soldering points) is not adequate here, because only the nodal displacements of the machined nodes will be considered in the optimization process. In fact, nodes on which no external loads are applied are associated to zero components in the force vector. Furthermore, compliance minimization is generally associated to problems were the structure itself needs to be optimized, that is the topology of the solid. Another approach consists in minimizing a metric that depends on the displacement values only, without considering the forces directly. Some cost functions that satisfy this criterion are: $c_0(\mathbf{x}) = \mathbf{q}^T\mathbf{q}$, $c_0(\mathbf{x}) = \max_k \mathrm{q}_{(k)}^2$, $c_0(\mathbf{x}) = \sum_k |q_{(k)}|$ or $c_0(\mathbf{x}) = \max_k |q_{(k)}|$, where k is an index on the components of $\mathbf{q}$. In machining, satisfying mechanical tolerances is one of the main goals and can only be guaranteed if the displacements are 'sufficiently' small.

The problem with objective functions like $c_0(\mathbf{x}) = \mathbf{q}^T\mathbf{q}$ or $c_0(\mathbf{x}) = \sum_k |q_{(k)}|$, is that an 'overall' representative value is calculated for evaluating the amount of deformation within the workpiece, which could lead to a solution that is not ideal. In fact, let us consider two separate cases. In the first case, a FE mesh of a workpiece presents small displacements at each of its nodes, for which the cost function evaluates as $c_0^{(1)}(\mathbf{x}) = \mathbf{q}_1^T\mathbf{q}_1$. In the second hypothetical case, the same mesh presents zero displacement at every node except for one DOF, where $q_i = \bar{q}$ is

a relatively large displacement. The objective function then evaluates as $c_0^{(2)}(\mathbf{x}) = \bar{q}^2$. Hence, if $c_0^{(1)}(\mathbf{x}) > c_0^{(2)}(\mathbf{x})$, an optimization algorithm could lead to a solution for which only specific nodes present large displacements (violating the tolerances), while other solutions would have presented acceptable displacement values at all nodes. Therefore, a cost function evolving around the maximum displacement has been considered here, that is $c_0(\mathbf{x}) = \max_k \mathrm{q}_{(k)}^2$ or $c_0(\mathbf{x}) = \max_k |q_{(k)}|$.

The generalized displacement vector itself is obtained by considering the static equilibrium equation $\mathbf{Kq} = \mathbf{f}$, where $\mathbf{f}$ an the external load vector and $\mathbf{K}$ is the structural stiffness matrix of the system. The vector $\mathbf{f}$ remains constant throughout the optimization process. The dependency of $\mathbf{q}$ on the design variables arises from the fact that the stiffness matrix is a function of $\mathbf{x}$, that is $\mathbf{K} = \mathbf{K}(\mathbf{x})$. More specifically, the structural stiffness matrix depends on the individual bar stiffnesses. An important observation is that $\mathbf{K}$ can be expressed as the sum of two distinct contributions: one from the workpiece and the other from the bars attached to it, i.e. $\mathbf{K}(\mathbf{x}) = \mathbf{K}_{\text{workpiece}} + \mathbf{K}_{\text{bars}}(\mathbf{x})$; only the second term is affected by changes in the design variables.

Another important aspect of any optimization problem is the imposition of constraints on the design variables. These ensure that a solution is sought in a feasible domain. For structural problems, constraints can include an upper bound on the equivalent stress, an upper bound on the mass, etc. In this work, different types of constraints can be mentioned:

1. By construction, the design variables x_i must be contained within 0 and 1, which is expressed as:
$$0 \leq x_i \leq 1 \qquad (i = 1, ..., n). \tag{13}$$

2. To ensure that bar elements are being 'removed' during the optimization process, one must impose a constraint on the total number of soldering points such that only the most useful ones remain. This can be imposed as follows:
$$\sum_{i=1}^{n} x_i \leq V \ , \tag{14}$$
where V can be tuned and corresponds to the number of bar elements in the solution.

3. To avoid resonance between the cutting tool forces and the workpiece-fixture assembly, one might also consider imposing a constraint on the natural frequencies of the system. Thus, a lower bound constraint can be imposed on the first eigenfrequency of the system to ensure that no resonance will occur:
$$\omega_1 \geq \Omega \ , \tag{15}$$
where the first eigenfrequency ω_1 is a function of the design variables.

Having described the context, the objective function, the design variables and the constraints, the optimization problem can now be formulated as follows. We are looking for the optimal distribution of soldering points, characterized by the design variables x_i $(i = 1, ..., n)$, and where the Young's modulus and the density are derived from Eq. 12, that minimize the

square of the largest components of the generalized displacement vector $\mathbf{q}$ under a given external machining load, and such that only V connections remain at the end of the process. Mathematically, this is expressed as follows:

$$\begin{aligned} &\min_{\mathbf{x}} \ \max_{k} \ \mathrm{q}^2_{(k)} \\ &\text{s.t.} \quad \sum_{i=1}^{n} x_i \leq V \qquad (i = 1, ..., n) \\ &\qquad\ \ 0 \leq x_i \leq 1. \end{aligned} \tag{16}$$

An additional constraint on the first eigenfrequency of the workpiece-fixture system (see Eq. 15) can be introduced to prevent resonance during the cutting process. However, this constraint is not essential for running the optimization algorithm.

3.2.2 Feasible initial guess

Optimization algorithms require an initial point (or guess) to begin the optimization process. To ensure that the initial guess $\mathbf{x}^0$ is feasible, it must satisfy both Eq. 13 and 14. A feasible starting points can be defined as follows:

$$x_i^0 = \frac{V}{n}, \quad \forall i = 1, \ldots, n.$$

Here, $0 < V < n$ ensures that $0 < x_i^0 < 1$, and the sum $\sum_{i=1}^{n} x_i^0 = V$ satisfies the constraint in Eq. 14. Another obvious choice is $x_i^0 = 0$, $\forall i = 1, ..., n$. Note that it is more difficult to define a feasible starting point when the constraint on the first eigenfrequency is considered. In particular, if Ω is chosen too large, it can happen that no such point exists.

3.2.3 Extension to multiple load cases

The optimization problem outlined in Section 3.2.1 considers a single machining load case. However, it can be easily extended to multiple load cases to account for a representative set of machining load samples for a given cutting tool path. This extension is important, because real-world machining operations are typically non-stationary, meaning the cutting tool does not remain in a fixed position. Consequently, multiple load cases must be incorporated to accurately model the varying conditions during machining.

Given a cutting tool path and S representative force samples, denoted $\mathbf{f}_s$ where $s \in \{1, ..., S\}$, the objective function can be reformulated as follows:

$$\max_{s} \max_{k} q^2_{s(k)} \ , \tag{17}$$

where $\mathbf{q}_s$ represents the generalized displacement vector resulting from the external force $\mathbf{f}_s$. The only modification from the original optimization problem is the updated expression for the objective function. Instead of solving for a single static equilibrium and finding the largest component of the displacement vector, the new formulation requires solving the static equilibrium equations for each force vector 'sample' and identifying the largest displacement component across all vectors. Consequently, the computational cost of evaluating the objective function increases, as each force sample must be processed individually to determine the 'worst-case' displacement.

3.2.4 Modeling the soldering points

As mentioned previously, clamping elements are modeled as soldering points. These, in turn, are represented by bar or beam elements in the finite element model, as discussed in (Beckers 1997). Beckers also highlights that beam elements provide a more accurate representation of soldering points due to their ability to capture rotational stiffness. However, it should be noted that, since the bar/beam elements are connected to the workpiece mesh via a single node, the results in these regions should be interpreted with caution.

In the present study, bar elements have been used. This choice is justified by the fact that the workpiece mesh, modeled using 3D solid elements, only includes translational degrees of freedom (DOFs) at the nodes; rotational DOFs are not present. Consequently, the use of beam elements would not be appropriate in this context.

The soldering points have been modeled as circular steel bars with a diameter of $D = 3$ [mm], a length $L = 3$ [mm], a Young's modulus $E_0 = 206.94$ [GPa], a Poisson's ratio $\nu = 0.288$, and a density $\rho_0 = 7{,}829$ [kg/m^3]. This choice is somewhat arbitrary and not based on a specific physical justification, other than approximating the dimensions of real soldering points. Later, the influence of these material properties is analyzed to assess their impact on the solution. In that case, the bar stiffness is parameterized as:

$$E_b = \alpha E_w \ , \tag{18}$$

where E_b and E_w denote the Young's moduli of the bar and the workpiece, respectively, and α is the bar-to-workpiece stiffness ratio. This approach has the advantage of defining the bar properties relative to those of the workpiece.

Finally, when adding a bar element to the workpiece mesh, it is important to ensure structural stability. One end of the bar is typically attached to a workpiece node, while the other end remains free. This configuration can render the global stiffness matrix $\mathbf{K}$ singular, making the computation of $\mathbf{q}$ impossible. To avoid this issue, the free ends must be fixed, as shown in Figure 15.

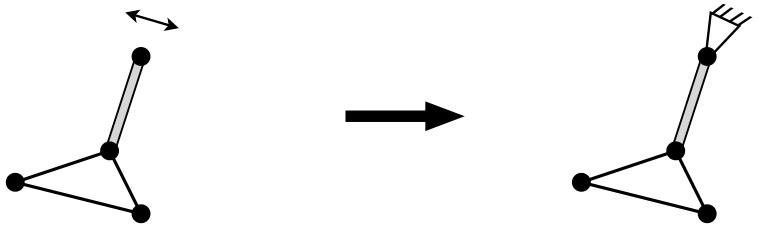


Figure 15: Schematic of a bar element attached to a workpiece mesh with both a free and an attached end.

3.2.5 Advantages and limitations

The approach presented above offers several notable advantages, alongside inherent assumptions and limitations, which are discussed in this subsection.

Advantages

The proposed method is notably general in nature. While much of the existing literature on fixture optimization focuses on simplified academic cases, particularly prismatic geometries using a 3-2-1 fixture layout, this approach imposes no restrictions on the geometry or shape of the workpiece. Any arbitrary mesh can be used, provided that the necessary element types (e.g., hexahedrons, second-degree elements, etc.) are implemented in the code.

Another key advantage is the simplicity of the method, which only relies on linear static analysis to compute the objective function. Additionally, clamping points are modeled using bar elements, which are better suited for connection with 3D tetrahedral meshes where nodes possess only translational degrees of freedom in the x, y, and z directions. Beam elements, which include rotational DOFs, are not compatible with this mesh configuration, further justifying the use of bar elements for modeling soldering points.

Assumptions and limitations

Several assumptions can be mentioned for this method. First, the workpiece is assumed to behave as a linear static body throughout the analysis. This may not fully capture the physical behavior during actual machining operations, where dynamic effects and nonlinearity could be significant.

The approach requires a priori knowledge of potential clamping regions. Specifically, the user must define possible locations on the workpiece's surface where bar elements can be placed. This requirement comes from the fact that the algorithm does not automatically determine feasible clamping regions. Instead, it is assumed that the user has sufficient knowledge of the machining process to identify areas that must remain free from constraints.

It is also assumed that both the mesh (including element and node positions) and the applied machining loads are known in advance. As described in Section 3.2.4, the bar elements used to model the soldering points are defined with steel material properties and arbitrary yet representative dimensions of 3 [mm] in both length and diameter, approximating real-world soldering points.

Another critical assumption is that the mesh remains unchanged throughout the optimization process. This implies that the optimization operates only on predefined node positions and does not involve remeshing or mesh adaptation.

Despite its advantages, the method comes with several limitations. One major concern is computational complexity: as mesh resolution and model detail increase, FEM-based analysis can become more expensive, limiting the scalability for industrial-scale applications. To address this, the potential use of model reduction techniques has been investigated to decrease

computational cost while maintaining good accuracy.

Also, the method does not account for fixture deformation, which could influence both stability and vibration characteristics during machining. The material properties of the workpiece are also assumed to be homogeneous and isotropic, which may not hold true for components made of composite materials for example.

3.3 Finite Element Modeling

This section is aimed at describing the elementary stiffness and mass matrices that are used to model the workpiece as well as the soldering points. These are essential for the assembly of structural finite element matrices.

3.3.1 Tetrahedral finite element

This section provides a detailed description of the 3D, four-node tetrahedral linear finite element used in the Python FEM code to model the workpiece. The development of this element is based on the work of Chandrupatla (2002), specifically from pages 275 to 281. Interested readers may refer to this source for further details.

Displacements, strain and stress

The finite element method formulation used in this work is as follows. The displacement vector is denoted $\mathbf{u} = [u \;\; v \;\; w]^T$ whose elements correspond to the displacement in x, y and z-directions respectively. Stress and strains tensors are denoted by $\boldsymbol{\sigma} = [\sigma_{xx} \;\; \sigma_{yy} \;\; \sigma_{zz} \;\; \tau_{yz} \;\; \tau_{xz} \;\; \tau_{xy}]^T$ and $\boldsymbol{\varepsilon} = [\varepsilon_{xx} \;\; \varepsilon_{yy} \;\; \varepsilon_{zz} \;\; \gamma_{yz} \;\; \gamma_{xz} \;\; \gamma_{xy}]^T$ respectively, where the strain is related to the displacements through the infinitesimal strain tensor:

$$\varepsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right). \tag{19}$$

The stress-strain relationship is given by:

$$\boldsymbol{\sigma} = \mathbf{D}\boldsymbol{\varepsilon}. \tag{20}$$

For an linear isotropic material with a Young's modulus E and a Poisson ratio ν, we have:

$$\mathbf{D} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2}-\nu & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2}-\nu & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2}-\nu \end{bmatrix} \tag{21}$$

Tetrahedral elements

As mentioned earlier, each tetrahedral element consists of 4 nodes, which are connected by the 6 edges of the tetrahedron. Each node has 3 degrees of freedom, and these are represented in the following local DOF vector:

$$\mathbf{q} = [q_1, q_2, ..., q_{12}]^T \;, \tag{22}$$

where local node i $(i = 1, 2, 3, 4)$ possesses the degrees of freedom q_{3i-2}, q_{3i-1} and q_{3i} which are the displacements in x, y and z-directions respectively.

To approximate the solution of a linear static problem, 4 linear shape functions are associated to each of the elements. These functions are expressed in terms of the isoparametric coordinates ξ, η and ζ, where the master element is shown in Figure 16:

$$N_1 = \xi \;\; , \quad N_2 = \eta \;\; , \quad N_3 = \zeta \;\; , \quad N_4 = 1 - \xi - \eta - \zeta. \tag{23}$$

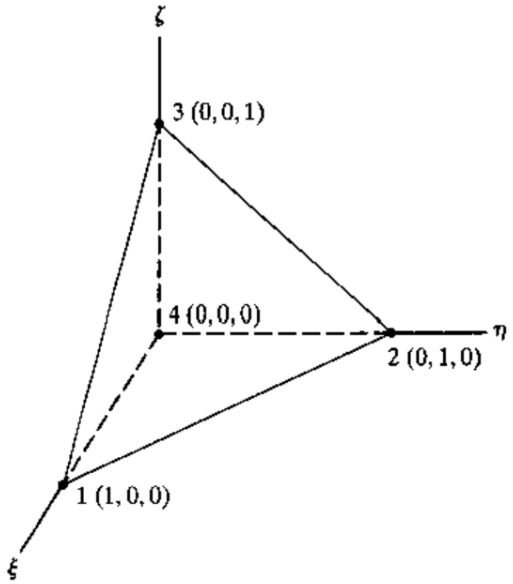


Figure 16: Tetrahedral master element (Chandrupatla 2002).

The displacements $\mathbf{u} = [u \quad v \quad w]^T$ for any point within the element can thus be expressed in terms of the nodal displacements $\mathbf{q}$ as follows:

$$\mathbf{u} = \mathbf{N}\mathbf{q} \ , \tag{24}$$

where the matrix $\mathbf{N}$ is given by

$$\mathbf{N} = \begin{bmatrix} N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & 0 & N_4 & 0 & 0 \\ 0 & N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & 0 & N_4 & 0 \\ 0 & 0 & N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & 0 & N_4 \end{bmatrix} \tag{25}$$

Analogously to Eq. 24, the x, y and z coordinates of any point inside the tetrahedron can be linked to the coordinates ξ, η and ζ through the following isoparametric transformation, provided that the general coordinates x_i, y_i and z_i $(i = 1, 2, 3, 4)$ of the four nodes are known:

$$\begin{aligned} x &= x_4 + x_{14}\xi + x_{24}\eta + x_{34}\zeta \\ y &= y_4 + y_{14}\xi + y_{24}\eta + y_{34}\zeta \\ z &= z_4 + z_{14}\xi + z_{24}\eta + z_{34}\zeta \end{aligned} \tag{26}$$

where the notation $x_{ij} = x_i - x_j$, $y_{ij} = y_i - y_j$ and $z_{ij} = z_i - z_j$ has been used.

In addition, it can be shown that the Jacobian matrix for this transformation can be expressed as

$$\mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} = \begin{bmatrix} x_{14} & y_{14} & z_{14} \\ x_{24} & y_{24} & z_{24} \\ x_{34} & y_{34} & z_{34} \end{bmatrix} . \tag{27}$$

The determinant is therefore constant and given by

$$\det \mathbf{J} = x_{14}(y_{24}z_{34} - y_{34}z_{24}) + y_{14}(z_{24}x_{34} - z_{34}x_{24}) + z_{14}(x_{24}y_{34} - x_{34}y_{24}). \tag{28}$$

Also, we can express the inverse of the Jacobian matrix as follows:

$$\mathbf{A} = \mathbf{J}^{-1} = \frac{1}{\det \mathbf{J}} \begin{bmatrix} y_{24}z_{34} - y_{34}z_{24} & y_{34}z_{14} - y_{14}z_{34} & y_{14}z_{24} - y_{24}z_{14} \\ z_{24}x_{34} - z_{34}x_{24} & z_{34}x_{14} - z_{14}x_{34} & z_{14}x_{24} - z_{24}x_{14} \\ x_{24}y_{34} - x_{34}y_{24} & x_{34}y_{14} - x_{14}y_{34} & x_{14}y_{24} - x_{24}y_{14} \end{bmatrix} . \tag{29}$$

It can be shown, from the previous equations, that the strain-displacement relationship is given by

$$\boldsymbol{\varepsilon} = \mathbf{B}\mathbf{q} \ , \tag{30}$$

where

$$\mathbf{B} = \begin{bmatrix} A_{11} & 0 & 0 & A_{12} & 0 & 0 & A_{13} & 0 & 0 & -\tilde{A}_1 & 0 & 0 \\ 0 & A_{21} & 0 & 0 & A_{22} & 0 & 0 & A_{23} & 0 & 0 & -\tilde{A}_2 & 0 \\ 0 & 0 & A_{31} & 0 & 0 & A_{32} & 0 & 0 & A_{33} & 0 & 0 & -\tilde{A}_3 \\ 0 & A_{31} & A_{21} & 0 & A_{32} & A_{22} & 0 & A_{33} & A_{23} & 0 & -\tilde{A}_3 & -\tilde{A}_2 \\ A_{31} & 0 & A_{11} & A_{32} & 0 & A_{12} & A_{33} & 0 & A_{13} & -\tilde{A}_3 & 0 & -\tilde{A}_1 \\ A_{21} & A_{11} & 0 & A_{22} & A_{12} & 0 & A_{23} & A_{13} & 0 & -\tilde{A}_2 & -\tilde{A}_1 & 0 \end{bmatrix} . \tag{31}$$

Note that, in Eq. 31, we used the notation $\tilde{A}_i = \sum_{j=1}^{3} A_{ij}$. Also, note that the terms in $\mathbf{B}$ are all constant leading to constant stresses and strains within each element.

Elementary stiffness matrix

The elementary stiffness matrix can be obtained by substituting Eq. 20 and 30 into the expression of the element strain energy U_e (Chandrupatla 2002), thus leading to

$$\mathbf{K}^e = V_e \mathbf{B}^T \mathbf{D} \mathbf{B}. \tag{32}$$

V_e denotes the volume of the element and it can be shown that we have $V_e = |\det \mathbf{J}|/6$, where the determinant of the Jacobian matrix is given in Eq. 28.

Elementary mass matrix

The elementary mass matrix is obtained by replacing the shape function matrix $\mathbf{N}$ given in Eq. 25 into

$$\mathbf{M}^e = \rho \int_V \mathbf{N}^T \mathbf{N} dV \ , \tag{33}$$

where the density ρ has been assumed constant over the element. After integrating, it can be shown (Chandrupatla 2002) that

$$\mathbf{M}^e = \frac{\rho V_e}{20} \begin{bmatrix} 2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ & 2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ & & 2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ & & & 2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ & & & & 2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ & & & & & 2 & 0 & 0 & 1 & 0 & 0 & 1 \\ & & & & & & 2 & 0 & 0 & 1 & 0 & 0 \\ & & & & & & & 2 & 0 & 0 & 1 & 0 \\ & & & \text{sym.} & & & & & 2 & 0 & 0 & 1 \\ & & & & & & & & & 2 & 0 & 0 \\ & & & & & & & & & & 2 & 0 \\ & & & & & & & & & & & 2 \end{bmatrix} . \tag{34}$$

3.3.2 Bar finite element

This section is aimed at describing the bar element which has been used to model the soldering points in the optimization problem. The following development is based on Géradin (2015), where the same notations as above have been used.

Elementary matrices in 1D

Each bar element is composed of 2 nodes. In a reference frame attached to the bar element, let us denote $\mathbf{q}^e = [u_1 \;\; u_2]^T$ the displacements of the first and second nodes respectively, as shown in Figure 17. It is assumed that the displacement for any point inside the bar can be expresses as

$$u(x) = u_1 N_1(x) + u_2 N_2(x) \; , \tag{35}$$

where $N_1(x) = 1 - x/L$ and $N_2(x) = x/L$ are two linear shape functions such that $N_1(0) = N_2(L) = 1$ and $N_1(L) = N_2(0) = 0$. This can be rewritten as

$$\mathbf{u} = \mathbf{N}^T \mathbf{q}^e \; , \tag{36}$$

where $\mathbf{N} = [N_1 \;\; N_2]^T$. By introducing Eq. 36 into the expressions for the strain energy and the kinetic energy of a bar (Géradin 2015), it can be shown that the stiffness and mass matrices are given by:

$$\mathbf{k}^e = \frac{EA}{L} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \qquad \text{and} \qquad \mathbf{m}^e = \frac{\lambda L}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \tag{37}$$

where λ denotes the linear mass density of the bar in [kg/m].

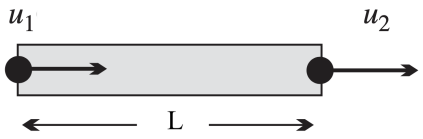


Figure 17: Bar element in 1D with nodal displacements (Géradin 2015).

Elementary matrices in 3D

Each bar element is composed of 2 nodes, both of which can move in the x, y and z-directions. Thus, at the local level, each element is assigned 6 DOFs:

$$\mathbf{q} = [q_1,\ q_2,\ ...,\ q_6]^T \tag{38}$$

where local node 1 possesses the degrees of freedom q_1, q_2 and q_3 and, for node 2, q_4, q_5 and q_6, corresponding the displacements in the structural axes in x, y and z-directions respectively.

Let us denote $\mathbf{X}_1 = (X_1, Y_1, Z_1)$ the coordinates of the first node in the structural (or global) reference frame and $\mathbf{X}_2 = (X_2, Y_2, Z_2)$ those of the second node. The displacements associated to both nodes are $\mathbf{U_1} = (U_1, V_1, W_1)$ and $\mathbf{U}_2 = (U_2, V_2, W_2)$ respectively. For each bar, we associate the unitary vector

$$\mathbf{e} = \frac{\mathbf{X}_2 - \mathbf{X}_1}{||\mathbf{X}_2 - \mathbf{X}_1||} \tag{39}$$

oriented along the central axis. A displacement of node i ($i = 1, 2$) can be linked to the local displacement u_i by projecting the displacement vectors $\mathbf{U}_i$ onto $\mathbf{e}$:

$$u_i = \mathbf{U}_i \cdot \mathbf{e}. \tag{40}$$

Thus, denoting $L = ||\mathbf{X}_2 - \mathbf{X}_1||$ the length of the bar element, $\mathbf{q}^e = [u_1 \quad u_2]^T$ and $\mathbf{q} = [q_1 \quad q_2 \quad ... \quad q_6]^T = [U_1 \quad V_1 \quad W_1 \quad U_2 \quad V_2 \quad W_2]^T$ are linked by the following relationship:

$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} \frac{X_2-X_1}{L} & \frac{Y_2-Y_1}{L} & \frac{Z_2-Z_1}{L} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{X_2-X_1}{L} & \frac{Y_2-Y_1}{L} & \frac{Z_2-Z_1}{L} \end{bmatrix} \begin{bmatrix} U_1 \\ V_1 \\ W_1 \\ U_2 \\ V_2 \\ W_2 \end{bmatrix}. \tag{41}$$

or

$$\mathbf{q}^e = \mathbf{R}\mathbf{q}\ , \tag{42}$$

where $\mathbf{R}$ is a rotation matrix. The element stiffness matrix in the structural axis is thus given by:

$$\mathbf{K}^e = \mathbf{R}^T \mathbf{k}^e \mathbf{R}. \tag{43}$$

Hence, adopting the notations $\Delta X = X_2 - X_1$, $\Delta Y = Y_2 - Y_1$ and $\Delta Z = Z_2 - Z_1$ and considering the expression of the stiffness matrix in the local frame in Eq. 37 as well as the expression of the rotation matrix in Eq. 41, the elementary stiffness matrix in structural axis can be expressed as follows:

$$\mathbf{K}^e = \frac{EA}{L^3} \begin{bmatrix} (\Delta X)^2 & \Delta X \Delta Y & \Delta X \Delta Z & -(\Delta X)^2 & -\Delta X \Delta Y & -\Delta X \Delta Z \\ \Delta X \Delta Y & (\Delta Y)^2 & \Delta Y \Delta Z & -\Delta X \Delta Y & -(\Delta Y)^2 & -\Delta Y \Delta Z \\ \Delta X \Delta Z & \Delta Y \Delta Z & (\Delta Z)^2 & -\Delta X \Delta Z & -\Delta Y \Delta Z & -(\Delta Z)^2 \\ -(\Delta X)^2 & -\Delta X \Delta Y & -\Delta X \Delta Z & (\Delta X)^2 & \Delta X \Delta Y & \Delta X \Delta Z \\ -\Delta X \Delta Y & -(\Delta Y)^2 & -\Delta Y \Delta Z & \Delta X \Delta Y & (\Delta Y)^2 & \Delta Y \Delta Z \\ -\Delta X \Delta Z & -\Delta Y \Delta Z & -(\Delta Z)^2 & \Delta X \Delta Z & \Delta Y \Delta Z & (\Delta Z)^2 \end{bmatrix}. \tag{44}$$

For the structural mass matrix, it is important to note that transverse displacements of the nodes contribute to the kinetic energy and it can be shown that (Géradin 2015):

$$\mathbf{M}^e = \frac{\lambda L}{6} \begin{bmatrix} 2 & 0 & 0 & 1 & 0 & 0 \\ 0 & 2 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 0 & 0 & 1 \\ 1 & 0 & 0 & 2 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 0 & 2 \end{bmatrix} \tag{45}$$

3.3.3 Comments on the choice of elements

It should be noted that, although tetrahedral elements have been chosen to model the workpiece, the code is flexible and allows users to implement other element types (e.g., hexahedral or second-order elements) if desired. The selection of tetrahedral elements has been motivated by two main factors. First, due to time constraints, we had to limit ourselves to a specific element type. Second, while tetrahedral elements may not always provide the highest precision, they are versatile, capable of modeling a wide range of geometries, and commonly supported by FEA software for automatic meshing (e.g., Siemens NX). Users are free to modify the code to incorporate them as needed.

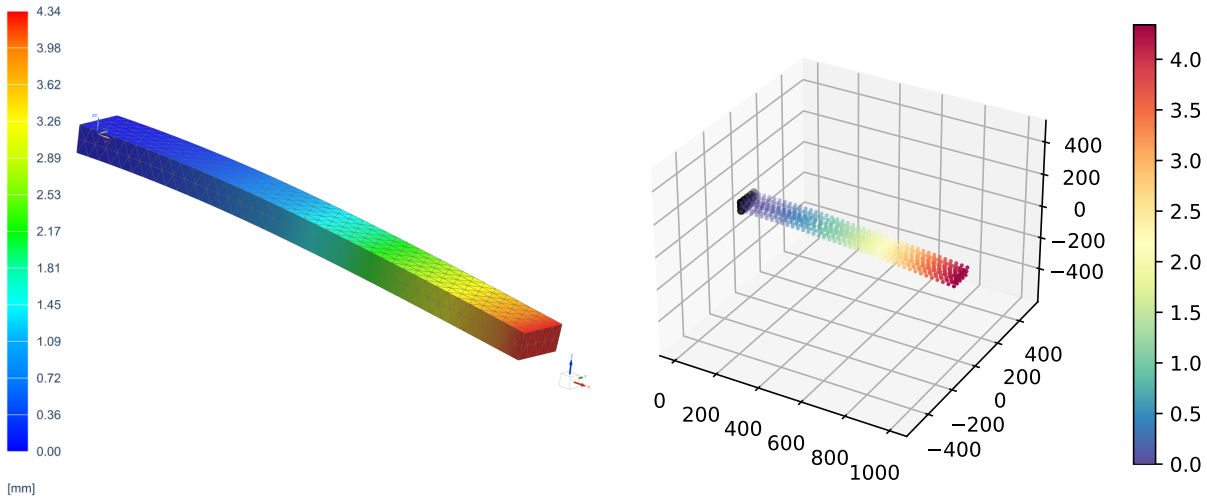
3.3.4 Code validation for tetrahedral elements

Stiffness matrix

To validate the Python FEM code for tetrahedral elements, simulations have been run for a $L = 1$ [m] long cantilever beam with a rectangular cross-section which is $w = 0.1$ [m] wide and $h = 0.05$ [m] in height. The material that has been considered is a steel with a density $\rho = 7{,}829$ [kg/m^3], a Young's modulus $E = 206.94$ [GPa] and a Poisson ratio $\nu = 0.288$ [$-$]. A downward uniform surface traction force $t = 10^6$ [N/m^2] has been applied on the free edge, such that the resultant force acting on the beam is $P = 5000$ [N].

Simulations in static linear elasticity have been performed using the FEM code and Siemens NX to compare the obtained results. In both cases, the same mesh has been considered; it consists of linear tetrahedral elements (T4) with an average size of 25 [mm]. The deformed shapes of the cantilever beam in NX are shown in Figure 18.

As can be seen in Figure 18a and 18b, both NX and Python lead to the same displacement field. In particular, both methods present identical maximum displacement of 4.34 [mm]. Even though results differ from the analytical estimation of 7.73 [mm], which could be explained by the used of tetrahedral elements, the Python code is validated for these finite elements.



(a) Siemens NX. (b) Python.

Figure 18: Displacement field for the deformed cantilever beam. All dimensions in [mm].

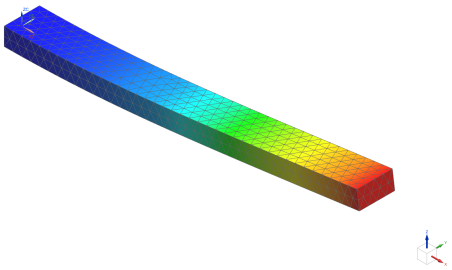
Mass matrix

The implementation for the stiffness matrices of T4 elements has been validated above. To validate the code for the mass matrices, the eigenfrequencies and mode shapes for the cantilever beam have been computed in Python and Siemens NX. Table 1 lists the results.

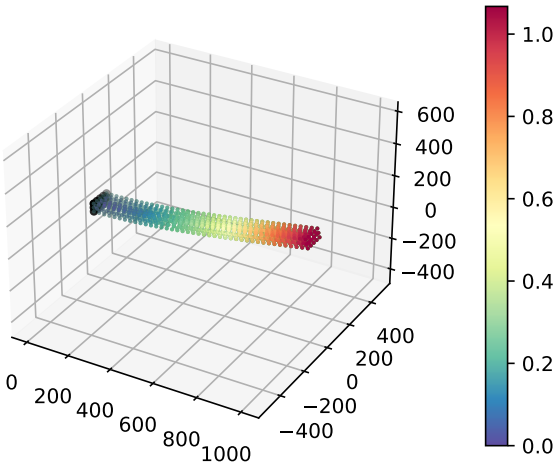
	Python [Hz]	Siemens NX [Hz]
Mode 1	55.476	55.48
Mode 2	88.015	88.02
Mode 3	344.9	345.15

Table 1: Eigenfrequencies of the cantilever beam in Python and Siemens NX.

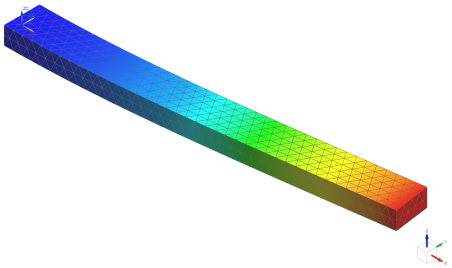
As shown in Table 1, the eigenfrequencies for the first 3 modes are almost identical with negligible differences. The corresponding mode shapes are shown in Figure 19. Hence, the code for the mass matrices of tetrahedral elements is validated also.



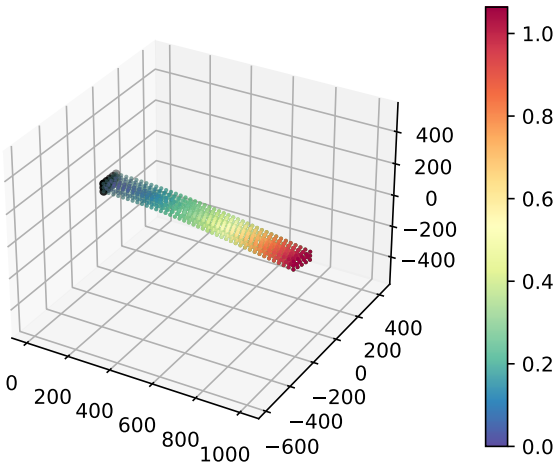
(a) NX - Mode 1.



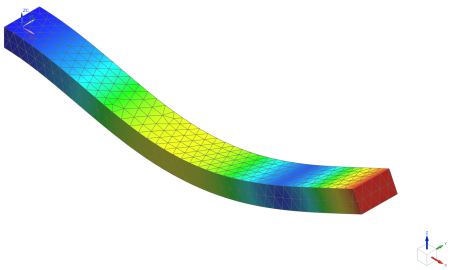
(b) Python - Mode 1.



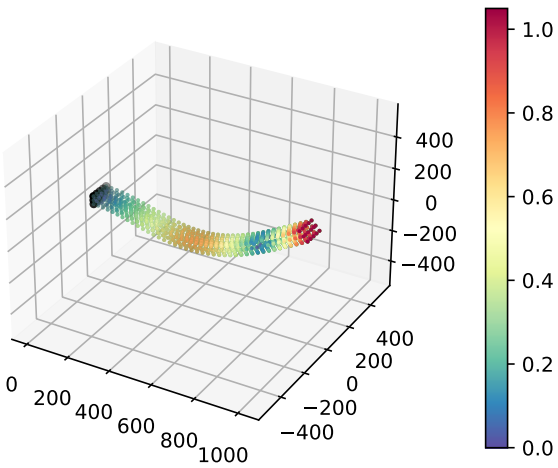
(c) NX - Mode 2.



(d) Python - Mode 2.



(e) NX - Mode 3.



(f) Python - Mode 3.

Figure 19: First 3 mode shapes computed in Siemens NX and Python. All dimensions in [mm].

3.3.5 Code validation for bar elements

Stiffness matrix

To validate the Python code for bar elements, simulations have been performed in Siemens NX and in Python for a simple bar truss. The truss is composed of 3 bars, as shown in Figure 20. The nodes at locations $(0, 0, 0)$, $(1, 0, 0)$ and $(0, 1, 0)$ [m] have been fixed, whereas a vertical load of 10 [kN] has been applied on the fourth node located at $(0, 0, 1)$. The material that has been considered is a steel with a density $\rho = 7{,}829$ [kg/m^3], a Young's modulus $E = 206.94$ [GPa] and a Poisson ratio $\nu = 0.288$ [$-$]. The bars have a circular cross-section with a diameter of 10 [mm].

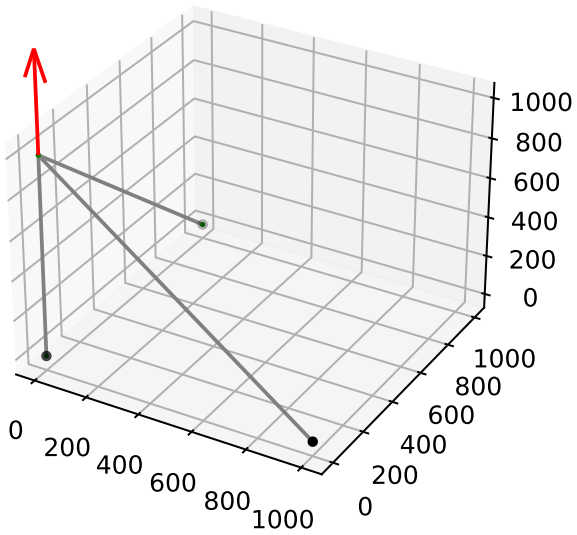


Figure 20: Simple bar truss with 3 bars, 3 fixed nodes and an applied vertical load. All dimensions in [mm].

The computed displacement of 1.066 [mm] for the upper node was identical in both cases, thus validating the bar element in Python.

Mass matrix

The implementation of the bar stiffness matrices have been validated above. To validate the mass matrices, the eigenfrequencies of the bar truss in Figure 20 have been computed (including boundary conditions) in Python and in NX. The results are given in Table 2.

	Python [Hz]	Siemens NX [Hz]
Mode 1	314.804	314.80
Mode 2	430.692	430.69
Mode 3	990.983	990.98

Table 2: Eigenfrequencies of the bar truss in Python and Siemens NX.

As seen in Table 2, the eigenfrequencies for the 3 modes are exactly the same in both cases, thus validating the code for the mass matrices of the bar elements.

3.3.6 Guyan-Irons reduction method

In many instances, it is not necessary to know the relationship between all the DOFs of a finite element mesh, as given by the structural matrices $\mathbf{K}$ and $\mathbf{M}$. In fact, for larger problems, it can be beneficial to transform a problem with a large number of DOFs into a problem with a reduced number of DOFs as computational expenses can be reduced. This is where 'reduction methods' come into play as they allow to economize DOFs retaining only a selected few (Géradin 2015).

The general idea behind reduction methods is the following. A general transient problem

$$\mathbf{Kq} + \mathbf{M\ddot{q}} = \mathbf{p}(t) \ , \tag{46}$$

is reduced by using a subspace $\mathbf{R}$ such that $\mathbf{q} \approx \mathbf{R\tilde{q}}$. Eq. 46 then becomes

$$\mathbf{KR\tilde{q}} + \mathbf{MR\ddot{\tilde{q}}} = \mathbf{p} + \mathbf{r} \ , \tag{47}$$

where $\mathbf{r}$ is the force residue associated to the approximation. A reduction matrix $\mathbf{R}$ such that the residue is zero, i.e. $\mathbf{R}^T\mathbf{r} = 0$, leads to the the following dynamic equilibrium equation:

$$\tilde{\mathbf{K}}\tilde{\mathbf{q}} + \tilde{\mathbf{M}}\ddot{\tilde{\mathbf{q}}} = \tilde{\mathbf{p}}(t) \ , \tag{48}$$

where $\tilde{\mathbf{K}} = \mathbf{R}^T\mathbf{KR}$, $\tilde{\mathbf{M}} = \mathbf{R}^T\mathbf{MR}$ and $\tilde{\mathbf{p}} = \mathbf{R}^T\mathbf{p}(t)$.

The Guyan–Irons method, also known as the 'static condensation method', is a specific type of reduction technique in which the generalized displacement vector $\mathbf{x}$ is partitioned into two components: the retained DOFs $\mathbf{x}_R$ and the condensed DOFs $\mathbf{x}_C$. The associated eigenvalue problem, $(\mathbf{K} - \omega^2\mathbf{M})\mathbf{x} = \mathbf{0}$, can then be written in block form as:

$$\begin{bmatrix} \mathbf{K}_{RR} & \mathbf{K}_{RC} \\ \mathbf{K}_{CR} & \mathbf{K}_{CC} \end{bmatrix} \begin{bmatrix} \mathbf{x}_R \\ \mathbf{x}_C \end{bmatrix} = \omega^2 \begin{bmatrix} \mathbf{M}_{RR} & \mathbf{M}_{RC} \\ \mathbf{M}_{CR} & \mathbf{M}_{CC} \end{bmatrix} \begin{bmatrix} \mathbf{x}_R \\ \mathbf{x}_C \end{bmatrix} . \tag{49}$$

The condensed DOFs $\mathbf{x}_C$ can be decomposed into a static and a dynamic contribution, $\mathbf{x}_C = \mathbf{x}_S + \mathbf{x}_D$. By neglecting the dynamic part $\mathbf{x}_D$, the full displacement vector can be approximated as (Géradin 2015):

$$\mathbf{x} \approx \begin{bmatrix} \mathbf{I} \\ -\mathbf{K}_{CC}^{-1}\mathbf{K}_{CR} \end{bmatrix} \mathbf{x}_R = \mathbf{R}\mathbf{x}_R. \tag{50}$$

Eq. 50 defines the reduction matrix $\mathbf{R}$ used in the Guyan–Irons method. Other reduction techniques differ primarily in how the reduction matrix $\mathbf{R}$ is constructed.

Validation

The Guyan-Irons method has been implemented in Python. To validate the code, the same cantilever beam considered in both linear static analysis and dynamic analysis (eigenvalue problem) has been used, where the structural matrices are reduced by retaining only the nodes at the end of the beam where the load is applied.

In the first case, the reduction resulted in exactly the same results as for the full model, which is expected since the Guyan-Irons method is exact in linear static analysis.

In the second case, the static condensation method led to the following eigenfrequencies for the first three modes:

	Structural [Hz]	Guyan-Irons [Hz]	Error [%]
Mode 1	55.476	55.75	0.49
Mode 2	88.015	88.5	0.55
Mode 3	344.9	552.82	60.3

Table 3: Eigenfrequencies of the cantilever beam in Python with and without the Guyan-Irons reduction method.

As shown in Table 3, the eigenfrequencies obtained using the Guyan-Irons method are consistently higher than those from the full structural case, with an increasing error. This outcome is expected, as the Guyan-Irons method always provides upper bound values for the eigenfrequencies (Géradin 2015). Also, it is important to note that the relative error increases with the mode number, but remains small for the first mode.

3.4 Sensitivity Analysis

Solving optimization problems requires the knowledge of the objective and constraint function $c_j(\mathbf{x})$ $(j = 0, 1, ..., m)$ values as well as their derivatives with respect to the design variables $\partial c_j/\partial x_i$ $(i = 1, ..., n)$ at the current design point $\mathbf{x}^k$. While these can be easily computed in the explicit case, their value is not straightforward for implicit problem where the objective function can only be computed at specific design points. This occurs, for example, through finite element analysis. In structural optimization, sensitivity analysis is aimed at determining the values of $\partial c_j/\partial x_i$ at a given point $\mathbf{x}^k$ (Duysinx 2020).

3.4.1 Sensitivity of the objective function

For static analysis, the equilibrium equation $\mathbf{Kq} = \mathbf{f}$ can be differentiated with respect to any design variable x_i $(i = 1, ..., n)$, leading to

$$\mathbf{K}\frac{\partial \mathbf{q}}{\partial x_i} = \frac{\partial \mathbf{f}}{\partial x_i} - \frac{\partial \mathbf{K}}{\partial x_i}\mathbf{q} := \tilde{\mathbf{f}}_i. \tag{51}$$

Thus, the derivative of the generalized displacement vector is obtained by solving the system for a new load case where $\tilde{\mathbf{f}}_i$ is introduced as the 'pseudo-load vector'. In the present case, the external load vector $\mathbf{f}$ is independent of the design variables, such that its derivative is equal to zero. Hence,

$$\tilde{\mathbf{f}}_i = -\frac{\partial \mathbf{K}}{\partial x_i}\mathbf{q}. \tag{52}$$

The sensitivity of the objective and constraint functions are then given by

$$\frac{dc_j(\mathbf{x}, \mathbf{q})}{dx_i} = \frac{\partial c_j}{\partial x_i} + \frac{\partial c_j}{\partial \mathbf{q}}\frac{\partial \mathbf{q}}{\partial x_i} = \frac{\partial c_j}{\partial x_i} + \mathbf{b}_j^T\frac{\partial \mathbf{q}}{\partial x_i}\ , \tag{53}$$

where $\mathbf{b}_j^T = \partial c_j/\partial \mathbf{q}$. Since the objective function $c_0(\mathbf{x}) = \max_k q_{(k)}^2$, given in Eq. 16, does not explicitly depend on the design variables, the derivative $\partial c_0/\partial x_i$ is equal to zero. Determining $\partial c_j/\partial q_k$ is not trivial, because of the 'max' expression in the objective function. Considering that a small perturbation Δx_i in the design variable x_i leads to a displacement vector $\mathbf{q}(\mathbf{x}+\Delta\mathbf{x}_i)$ for which the maximum index $k_{\max}$ is unchanged compared to $\mathbf{q}(\mathbf{x})$, the derivative is simply given by

$$\frac{\partial c_0}{\partial \mathbf{q}} = [0\ 0\ ...\ 2q_{k_{\max}}\ ...\ 0\ 0]^T\ , \tag{54}$$

where only the $k_{\max}$-th element is non-zero. In fact, consider a given generalized displacement vector; this vector possesses one (or more) component(s) $q_{k_{\max}}$ such that $q_{k_{\max}} > q_k$, $\forall k$. Since all other components are strictly smaller, adding an infinitesimal increment to any of the other components will not affect the objective function $c_0(\mathbf{x}) = \max_k q_{(k)}^2$. Thus, the derivative of c_0 with respect to q_k is zero $\forall k \neq k_{\max}$. On the other hand, if $k = k_{\max}$, it means that adding a small increment to q_k will affect the objective function, because it affects the largest component of $\mathbf{q}$. Therefore, the function c_0 will behave as the square function and its derivative with respect to q_k is $2q_k$ when $k = k_{\max}$. Finally, the derivative of the objective function with respect to the design variables is found by substituting the different terms into Eq. 53:

$$\frac{dc_0}{dx_i} = [0\ 0\ ...\ 2q_{k_{\max}}\ ...\ 0\ 0]\frac{\partial \mathbf{q}}{\partial x_i} = \mathbf{b}^T\frac{\partial \mathbf{q}}{\partial x_i}. \tag{55}$$

Only the derivative of the generalized displacement vector with respect to the design variables is unknown in Eq. 55. As mentioned above, it is obtained by solving the static equilibrium in Eq. 51 for the pseudo-load in Eq. 52. The sensitivity of the stiffness matrix can be obtained by considering the fact that $\mathbf{K}$ results from two distinct contributions $\mathbf{K}_{\text{workpiece}}$ and $\mathbf{K}_{\text{bars}}$. The former is associated to the workpiece mesh and is independent of the design variables. The latter corresponds to the contribution of the bars on the overall stiffness and is affected by changes in the design variables. Hence, we have:

$$\frac{\partial \mathbf{K}}{\partial x_i} = \frac{\partial \mathbf{K}_{\text{bars}}}{\partial x_i}. \tag{56}$$

The expression for the elementary stiffness matrix $\mathbf{K}_i^e$ of a bar element in Eq. 44, can be rewritten by considering the SIMP model in Eq. 12. In fact, the Young's modulus for each bar is given by $E_i = x_i^p E_0$, meaning that the elementary stiffness can be expressed as:

$$\mathbf{K}_i^e = x_i^p \bar{\mathbf{K}}_i^e \ , \tag{57}$$

where $\bar{\mathbf{K}}_i^e$ is the stiffness matrix of a 'full' bar, i.e. for $x_i = 1$. Eq. 57 leads to an analytical expression for the sensitivity of the stiffness matrix:

$$\frac{\partial \mathbf{K}_i^e}{\partial x_i} = p x_i^{p-1} \bar{\mathbf{K}}_i^e. \tag{58}$$

This analytical approach for the computation of the stiffness matrix sensitivity in Eq. 58 is an advantage of using SIMP, but is not always possible in the context of structural optimization. Indeed, often times it is necessary to approximate these sensitivities by employing a finite difference requiring the computation of $\mathbf{K}$ at multiple points close to the current design $\mathbf{x}^k$.

Once all $\partial \mathbf{K}/\partial x_i$ values are known, one may evaluate the generalized displacement sensitivity $\partial \mathbf{q}/\partial x_i$ by solving the pseudo-load case in Eq. 51 and then substituting the result into Eq. 55 to obtain the sensitivity of the objective function (direct approach):

$$\frac{dc_0}{dx_i} = \mathbf{b}^T \mathbf{K}^{-1} \tilde{\mathbf{f}}_i = -\mathbf{b}^T \mathbf{K}^{-1} \frac{\partial \mathbf{K}}{\partial x_i} \mathbf{q}. \tag{59}$$

This approach demands that n additional load cases are solved to have access to dc_0/dx_i, $\forall i = 1, ..., n$. Another approach (virtual load approach) consists in using the symmetry of the structural stiffness matrix $\mathbf{K} = \mathbf{K}^T \iff \mathbf{K}^{-1} = \mathbf{K}^{-T}$, to recognize that $\mathbf{b}^T \mathbf{K}^{-1} = \mathbf{b}^T \mathbf{K}^{-T} = (\mathbf{K}^{-1}\mathbf{b})^T = \mathbf{\Lambda}^T$, where $\mathbf{\Lambda}$ is the solution of

$$\mathbf{K}\mathbf{\Lambda} = \mathbf{b} = \frac{\partial c_0}{\partial \mathbf{q}} \ , \tag{60}$$

which requires only 1 additional load case to be solved (Duysinx 2020).

3.4.2 Extension to multiple load cases

When considering a machining process rather than a single load case, the approach must be slightly modified. In fact, if the process consists of S representative force samples $\mathbf{f}_s$, $s \in$

$\{1, 2, ..., S\}$, the displacement vectors related to each of those forces can be placed in a matrix of the form

$$\mathbf{Q} = [\mathbf{q}_1 \;\; \mathbf{q}_2 \;\; ... \;\; \mathbf{q}_S] = \mathbf{K}^{-1}[\mathbf{f}_1 \;\; \mathbf{f}_2 \;\; ... \;\; \mathbf{f}_S]. \tag{61}$$

The objective function in Eq. 17 is then obtained by picking the largest squared element in the matrix $\mathbf{Q}$. The total derivative of the objective function in Eq. 53 can be rewritten as follows:

$$\frac{dc_0(\mathbf{x}, \mathbf{q}_s)}{dx_i} = \frac{\partial c_0}{\partial x_i} + \sum_{s=1}^{S} \frac{\partial c_0}{\partial \mathbf{q}_s} \frac{\partial \mathbf{q}_s}{\partial x_i}. \tag{62}$$

Again, c_0 does not explicitly depend on the design variables, such that $\partial c_0/\partial x_i = 0$. To determine the derivatives $\partial c_0/\partial \mathbf{q}_s$, one must simply imagine that an infinitesimal quantity is added to the line k of $\mathbf{Q}$. If the line includes the maximum element, the derivative is $2q_k$. However, for all other lines, this increase has no effect on the objective function, because the increase is considered infinitesimal. Therefore, most of the vectors $\partial c_0/\partial \mathbf{q}_s$ are equal to zero except those that contain the largest element of $\mathbf{Q}$, which are given by Eq. 54. The procedure to find the derivatives $\partial \mathbf{q}_s/\partial x_i$ has already been outlined in the previous section.

3.4.3 Sensitivity of constraint functions

The sensitivity of the constraint function $c_1(\mathbf{x}) = \sum_{i=1}^{n} x_i$ in the present optimization problem given in Eq. 14 can be easily computed since the function explicitly depends on the design variables only. We have,

$$\frac{dc_1}{dx_i} = 1 \quad \forall i = 1, ..., n. \tag{63}$$

The second constraint function, given in Eq. 15, it is necessary to determine the sensitivity of the first eigenfrequency of the system. Consider the eigenvalue problem

$$(\mathbf{K} - \omega_k^2 \mathbf{M})\mathbf{q}_k = \mathbf{0} \;, \tag{64}$$

where ω_k^2 are the eigenvalues and the eigenmodes $\mathbf{q}_k$ are generally normalized with respect to the mass matrix $\mathbf{M}$, which writes as follows:

$$\mathbf{q}^T \mathbf{M} \mathbf{q} = 1. \tag{65}$$

By differentiating Eq. 64 with respect to x_i, one gets:

$$(\mathbf{K} - \omega_k^2 \mathbf{M}) \frac{\partial \mathbf{q}_k}{\partial x_i} = \frac{\partial \omega_k^2}{\partial x_i} \mathbf{M} \mathbf{q}_k - (\frac{\partial \mathbf{K}}{\partial x_i} - \omega_k^2 \frac{\partial \mathbf{M}}{\partial x_i}) \mathbf{q}_k. \tag{66}$$

Hence, premultiplying Eq. 66 by $\mathbf{q}_k^T$ and taking into account the normalization in Eq. 65, one obtains:

$$\frac{\partial \omega_k^2}{\partial x_i} = \mathbf{q}_k^T \left(\frac{\partial \mathbf{K}}{\partial x_i} - \omega_k^2 \frac{\partial \mathbf{M}}{\partial x_i} \right) \mathbf{q}_k. \tag{67}$$

The sensitivity in the second term on the right side of the equation above is easily computed. In fact, the structural mass matrix $\mathbf{M}$ can be seen as a sum of two contributions $\mathbf{M}_{\text{workpiece}}$ and $\mathbf{M}_{\text{bars}}$, where only the latter depends on the design variables. The elementary mass matrix for bar elements in Eq. 45 is proportional to $\lambda_i = \rho_i A = x_i \rho_0 A$. Hence, expressing the mass matrix as follows:

$$\mathbf{M}_i^e = x_i \bar{\mathbf{M}}_i^e \quad \Longrightarrow \quad \frac{\partial \mathbf{M}_i^e}{\partial x_i} = \bar{\mathbf{M}}_i^e \;, \tag{68}$$

where $\bar{\mathbf{M}}_i^e$ is the elementary mass matrix for a 'full' bar, i.e. for $x_i = 1$ (Duysinx 2020).

3.5 Sequential Convex Programming

This section describes the principle behind sequential convex programming, an optimization algorithm used in structural optimization. The particular case of the Method of Moving Asymptotes (MMA) has been discussed below.

3.5.1 Overview

Sequential convex programming (SCP) refers to methods that turn general optimization problems, which are non-linear, implicit and require many iterations to converge, into a sequence of subproblems that are explicit, convex and separable. This is done by using approximations (CONLIN, MMA, GCMMA, etc.) of the objective and constraint functions at the current point $\mathbf{x}^k$. These subproblems are easier to solve and admit only one solution due to convexity, which is obtained via dual methods and is then used as the next approximation point. Because the subproblems are separable, dual methods are very efficient for finding the solution. Two prominent examples of approximation methods are CONLIN (see Appendix B) and MMA, the latter of which has been discussed hereafter. The flowchart in Figure 21 shows the principle of SCP (Bruyneel 2024).

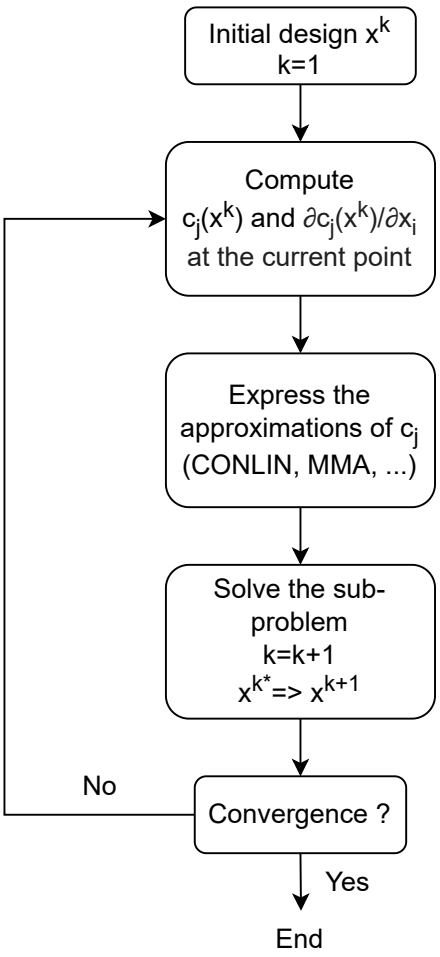


Figure 21: Principle of sequential convex programming (Bruyneel 2024).

As shown in Figure 21, the objective function c_0 and the constraint functions c_j $(j = 1, ..., m)$ as well as their derivatives with respect to the design variables x_i $(i = 1, ..., n)$ must be evaluated at the current design point $\mathbf{x}^k$. These values are used to define the approximations $\tilde{c}_j$ $(j = 0, ...m)$ which form a new subproblem. This subproblem is then solved to obtain the next design x^{k+1}, generally using dual methods (Svanberg 1987).

3.5.2 MMA - Subproblem formulation

The *Method of Moving Asymptotes* (MMA) is a SCP method where each convex and separable subproblem is generated by linearizing the objective and constraint functions in terms of $1/(x_i - L_i)$ and $1/(U_i - x_i)$, for which the 'moving asymptotes' L_i and U_i depend on the sign of the derivatives of c_j $(j = 0, 1, ..., m)$ at the current design point $\mathbf{x}^k$. At each iteration, their value is chosen to either stabilize or relax the algorithm if necessary. This method presents itself as a generalization of CONLIN. Note also that, unlike CONLIN, the design variables are not required to be positive (Svanberg 1987).

Given the following general nonlinear optimization problem:

$$\begin{aligned} &\min_{\mathbf{x}} c_0(\mathbf{x}) \quad (\mathbf{x} \in \mathbb{R}^n) \\ &\text{s.t.} \quad c_j(\mathbf{x}) \leq \hat{c}_j \quad (j = 1, ..., m) \\ &\qquad \underline{\mathrm{x}}_i \leq x_i \leq \bar{x}_i, \end{aligned} \tag{69}$$

the subproblems are generated by considering the following convex approximations:

$$c_j^{(k)}(\mathbf{x}) = r_j^{(k)} + \sum_{i=1}^{n} \left(\frac{p_{ji}^{(k)}}{U_i^{(k)} - x_i} + \frac{q_{ji}^{(k)}}{x_i - L_i^{(k)}} \right), \tag{70}$$

where

$$\begin{aligned} p_{ji}^{(k)} &= (U_i^{(k)} - x_i^{(k)})^2 \frac{\partial c_j}{\partial x_i} \quad \text{if} \quad \frac{\partial c_j}{\partial x_i} > 0 \quad \text{and} \quad 0 \quad \text{if} \quad \frac{\partial c_j}{\partial x_i} \leq 0 \\ q_{ji}^{(k)} &= -(x_i^{(k)} - L_i^{(k)})^2 \frac{\partial c_j}{\partial x_i} \quad \text{if} \quad \frac{\partial c_j}{\partial x_i} < 0 \quad \text{and} \quad 0 \quad \text{if} \quad \frac{\partial c_j}{\partial x_i} \geq 0 \\ r_j^{(k)} &= c_j(\mathbf{x}^{(k)}) - \sum_{i=1}^{n} \left(\frac{p_{ji}^{(k)}}{U_i^{(k)} - x_i^{(k)}} + \frac{q_{ji}^{(k)}}{x_i^{(k)} - L_i^{(k)}} \right), \end{aligned} \tag{71}$$

where the derivatives $\partial c_j / \partial x_i$ are all evaluated at the current design point $\mathbf{x}^{(k)}$. The moving asymptotes are chosen following a given selection strategy which has been discussed hereafter.

The 'moving asymptotes' $L_i^{(k)}$ and $U_i^{(k)}$ are always chosen such that $L_i^{(k)} < x_i^{(k)} < U_i^{(k)}$ $(i = 1, ..., n)$. To choose their values at each iteration, Svanberg (1987) has proposed the following approach:

1. The process is stabilized by moving the asymptotes closer to the current design point $\mathbf{x}^{(k)}$ if oscillations occur.

2. For a slow and monotone process, the asymptotes may be moved further away from $\mathbf{x}^{(k)}$ to accelerate the convergence.

This may be accomplished by considering the following rule. For $k = 0$ and $k = 1$,

$$\begin{cases} L_i^{(k)} = x_i^{(k)} - (\bar{x}_i - \underline{x}_i) \\ U_i^{(k)} = x_i^{(k)} + (\bar{x}_i - \underline{x}_i) \end{cases} \tag{72}$$

For $k \geq 2$, if the sign of $x_i^{(k)} - x_i^{(k-1)}$ and $x_i^{(k-1)} - x_i^{(k-2)}$ are opposite, then

$$\begin{cases} L_i^{(k)} = x_i^{(k)} - s(x_i^{(k-1)} - L_i^{(k-1)}) \\ U_i^{(k)} = x_i^{(k)} + s(U_i^{(k-1)} - x_i^{(k-1)}) \end{cases} \tag{73}$$

Else, if the signs are equal,

$$\begin{cases} L_i^{(k)} = x_i^{(k)} - (x_i^{(k-1)} - L_i^{(k-1)})/s \\ U_i^{(k)} = x_i^{(k)} + (U_i^{(k-1)} - x_i^{(k-1)})/s \ , \end{cases} \tag{74}$$

where s is a parameter between 0 and 1 which can be tuned to change the behavior of the method (Svanberg 1987).

3.5.3 MMA - Subproblem resolution

The convex approximations in Eq. 70 lead to the following subproblem:

$$\begin{aligned} \min \quad & \sum_{i=1}^{n} \left(\frac{p_{0i}}{U_i - x_i} + \frac{q_{0i}}{x_i - L_i} \right) + r_0 \\ \text{s.t.} \quad & \sum_{i=1}^{n} \left(\frac{p_{ji}}{U_i - x_i} + \frac{q_{ji}}{x_i - L_i} \right) \leq b_j \quad (j = 1, ..., m) \\ & \alpha_i \leq x_i \leq \beta_j \quad (i = 1, ..., n) \ , \end{aligned} \tag{75}$$

where the notation of the iteration index k has been suppressed and $b_j = \hat{c}_j - r_j$. The Lagrangian function associated to the subproblem in Eq. 75 writes

$$\begin{aligned} l(\mathbf{x}, \mathbf{y}) &= c_o^{(k)}(\mathbf{x}) + \sum_{j=1}^{m} y_j c_j^{(k)}(\mathbf{x}) \\ &= r_0 - \mathbf{y}^T \mathbf{b} + \sum_{i=1}^{n} l_i(x_i, \mathbf{y}) \end{aligned} \tag{76}$$

where $\mathbf{y} \geq 0$ is the vector containing the dual variables and

$$l_i(x_i, \mathbf{y}) = \frac{p_{0i} + \mathbf{y}^T \mathbf{p}_i}{U_i - x_i} + \frac{q_{0i} + \mathbf{y}^T \mathbf{q}_i}{x_i - L_i}. \tag{77}$$

The Lagrangian functions in Eq. 76 and 77 lead to the following dual function, which corresponds to the minimum with respect to $\mathbf{x}$ for a given $\mathbf{y}$:

$$\begin{aligned} W(\mathbf{y}) &= \min_{\mathbf{x}} \{ l(\mathbf{x}, \mathbf{y}); \alpha_i \leq x_i \leq \beta_i, \forall i \} \\ &= r_0 - \mathbf{y}^T \mathbf{b} + \sum_{i=1}^{n} W_i(\mathbf{y}) \ , \end{aligned} \tag{78}$$

where

$$W_i(\mathbf{y}) = \min_{x_i}\{l_i(x_i, \mathbf{y}); \alpha_i \leq x_i \leq \beta_i\}. \tag{79}$$

It can be shown (Svanberg 1987) that the primal-dual relationship $x_i(\mathbf{y})$, i.e. the values of the design variables that minimize the dual function in Eq. 78 for a given $\mathbf{y}$, is given by:

1. If $l_i'(\alpha_i, \mathbf{y}) \geq 0$, then $x_i(\mathbf{y}) = \alpha_i$,
2. If $l_i'(\beta_i, \mathbf{y}) \geq 0$, then $x_i(\mathbf{y}) = \beta_i$,
3. If $l_i'(\alpha_i, \mathbf{y}) < 0$ and $l_i'(\beta_i, \mathbf{y}) > 0$, then one has

$$x_i(\mathbf{y}) = \frac{\sqrt{p_{0i} + \mathbf{y}^T\mathbf{p}_i}\ L_i + \sqrt{q_{0i} + \mathbf{y}^T\mathbf{q}_i}\ U_i}{\sqrt{p_{0i} + \mathbf{y}^T\mathbf{p}_i} + \sqrt{q_{0i} + \mathbf{y}^T\mathbf{q}_i}}. \tag{80}$$

By substituting the expression of the primal-dual relationship $x_i(\mathbf{y})$ into the dual function in Eq. 78, it can be seen that W is a fully explicit function of $\mathbf{y}$. The dual problem then consists in maximizing $W(\mathbf{y})$ ($\mathbf{y} \geq 0$) using any gradient method, such as the conjugate gradient method for example. Then, to find the solution to the MMA subproblem in Eq. 75, one simply has to replace the solution to the dual problem into the primal-dual relationship (Svanberg 1987).

Artificial variables z_j ($j = 1, ..., m$) can be introduced in the MMA subproblem to deal with the case where the subproblem is infeasible. The solution returned by the MMA is then as close as possible to being feasible. If the subproblem is feasible, then these new variables become equal to zero in the optimal solution. Interested readers may refer to (Svanberg 1987) for more details.

3.5.4 MMA - Python module

To optimize the clamping locations after completing the preliminary steps, such as FEM simulations, sensitivity analysis, and related tasks, a dedicated Python module developed by Arjen Deetman has been used (Deetman 2024). This module provides a Python implementation of the MMA, originally introduced by Svanberg in MATLAB and released under the GNU General Public License.

3.6 Code Structure and Implementation

This section provides a detailed discussion of the practical implementation of the methodology, alongside an overview of the overall structure and organization of the Python files. The complete code is included in the appendices for reference.

3.6.1 Code overview

Several components must be implemented and considered to successfully perform the optimization. First and foremost, the input data must be properly defined and handled. Relevant inputs include: a list of node coordinates, element connectivities (for both tetrahedral and bar elements), fixed nodes (if applicable), material properties, and machining loads. The format and handling of these inputs are discussed in more detail in a subsequent section. The Python file `inputs.py` is responsible for processing this data and organizing it into structured arrays for further use. For those interested, the code can be found in Appendix C.

Next, the element-level mass and stiffness matrices for both tetrahedral and bar elements must be implemented. This is handled in the file `elementary.py`. The analytical expressions used for these matrices have been derived in Section 3.3. Once the elementary stiffness matrices are defined, the structural mass and stiffness matrices can be assembled, a task carried out in `assembly.py`. As mentioned in Section 3.2, the structural matrices can be decomposed into two distinct contributions corresponding to the tetrahedral and bar elements. Specifically, for the stiffness matrix, we have:

$$\mathbf{K} = \mathbf{K}_{\text{workpiece}} + \mathbf{K}_{\text{bars}}, \tag{81}$$

where $\mathbf{K}_{\text{workpiece}}$ represents the stiffness contribution from the tetrahedral elements modeling the bulk of the structure, and $\mathbf{K}_{\text{bars}}$ accounts for the contributions from the clamping bar elements. The former is constant, while the latter depends on the design variables $\mathbf{x} = [x_1 \;\; x_2 \;\; \dots \;\; x_n]^T$, i.e., $\mathbf{K}_{\text{bars}} = \mathbf{K}_{\text{bars}}(\mathbf{x})$, since variations in the mechanical properties of the bars directly affect the stiffness of the system.

As described in Section 3.3.6, the structural matrices can be reduced using the Guyan–Irons method to obtain a smaller system with fewer degrees of freedom. Functions related to this reduction have been implemented in the file `reduction.py`.

While the objective function can be evaluated once the generalized displacements are computed from the equilibrium equation, determining the sensitivities (i.e. the derivatives) is not straightforward. Hence, the file `sensitivity.py` has been developed to compute and return the required derivatives for any given design point $\mathbf{x}^k$. The methodology for obtaining these sensitivities has been explained in detail in Section 3.4.

The optimization itself is carried out using an external Python module implementing the MMA, as described in Section 3.5. This module was written by Deetman (2024). Further explanation of its implementation and the parameters it uses is provided in a following section. To function properly, the module requires a routine that returns the values of the objective and constraint functions, as well as their derivatives with respect to the design variables, for any given design point $\mathbf{x}^k$.

The optimization process using the MMA is performed in the file `optimization_MMA.py`, where the objective and constraints functions must be evaluated at each iteration point, as well as their derivatives with respect to the design variables. Also, an alternative script, `optimization_scipy.py`, implements the same problem using a generic optimization method from `scipy.optimize`. However, this method is not well-suited to structural optimization and is included primarily for comparison purposes.

To display the meshes, forces, displacements, and results in general, an additional file `display.py` has been created. It contains functions that can be invoked once the optimal clamping distribution has been determined.

3.6.2 Inputs

The program requires different inputs which have been provided to the Python code via text files. These are the following:

1. *nodeList.txt*: At first, the user has to provide all the nodes that appear in the finite element model, including their respective coordinates. This is done by supplying a list in which the lines correspond to the different nodes and the columns correspond to x, y and z coordinates respectively. In particular, the i-th line of this list is associated to node i, giving access to the location in the three-dimensional structural frame of reference. All coordinates should be expressed in millimeters [mm].

2. *elemListWorkpiece.txt*: Next, a mesh of tetrahedral elements must be provided. This mesh is associated to the workpiece and takes the form of a list where lines correspond to the various elements and where 4 columns are dedicated to their respective nodes (T4 connectivity). The i-th line corresponds to element number i. The sequence of these nodes corresponds to the labeling of the nodes in the tetrahedral master element shown in Figure 16.

3. *elemListBars.txt*: The list in this file is made of different lines representing the bar element, whose nodes are given in 2 columns (bar connectivity). Crucially, the order in which these bars appear corresponds to the sequence of design variables. The i-th line represents the i-th bar, for which the Young's modulus is determined by the i-th design variable x_i.

4. *machiningLoads.txt*: A new list has been created to be able to introduce machining loads which act on the workpiece. Uniquely describing an imposed load requires two things: the node on which the force is acting and the components of the force vector. Hence, it has been decided to use a list for this purpose, where the first column corresponds to the node and the other 3 columns to the x, y and z components of the nodal force vector (in [N]). In cases where multiple lines appear in the list (i.e. forces applied to different nodes), the user can choose to either consider each force separately (as part of the machine tool path) or optimize the clamping locations for the entire loading scenario.

5. *fixedNodes.txt*: This file contains a list with the nodes that are considered fixed (or clamped). It is important to remember to clamp the free ends of each bar; otherwise, the system will have a mechanism, and solving it will not be possible. Additionally, it may

be necessary to clamp some of the workpiece nodes. For instance, if the optimization process begins with $\mathbf{x}^0 = \mathbf{0}$, there are no bars to restrict the motion of the workpiece and the stiffness matrix will be singular.

In the current version of the program, all these files should be placed in a folder labeled '*Inputs*', which will be accessed by the `inputs.py` file.

3.6.3 Reduction process

To improve computational efficiency in large-scale FE models with a high number of DOFs, model reduction techniques, particularly the Guyan-Irons method, which is exact for linear static problems, can be beneficial (Géradin 2015). In the case of a workpiece-bars configuration, the workpiece typically contributes the majority of the system's DOFs, leading to high computational costs. In contrast, the bar elements, which are attached to the external nodes of the mesh, generally have fewer DOFs. Therefore, it is advantageous to reduce the workpiece mesh by retaining only the DOFs of nodes that are connected to bar elements, as illustrated in Figure 22.

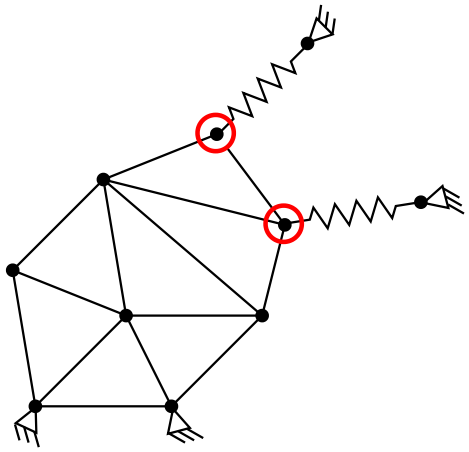


Figure 22: Schematic showing the retained nodes (highlighted in red) used for model reduction.

Hence, the structural stiffness matrix (with boundary conditions already applied) is given by

$$\mathbf{K} = \mathbf{K}_{\text{workpiece}} + \mathbf{K}_{\text{bars}}, \tag{82}$$

where $\mathbf{K}$ has size $(N \times N)$ and where $\mathbf{K}_{\text{workpiece}}$ is independent of $\mathbf{x}$. Following the procedure outlined in Section 3.3.6, this matrix can be reduced to a smaller system

$$\tilde{\mathbf{K}} = \tilde{\mathbf{K}}_{\text{workpiece}} + \tilde{\mathbf{K}}_{\text{bars}}, \tag{83}$$

of size $(3n \times 3n)$, where n is the number of bars and $3n$ accounts for the 3 DOFs per bar element (each bar has 1 fixed node). The reduced system contains only the DOFs associated with the nodes where the bars are attached, as described above. These DOFs are stored in the reduced displacement vector $\tilde{\mathbf{q}}$, and the reduced force vector is denoted by $\tilde{\mathbf{f}}$. Although the external force may be applied to other nodes in the workpiece mesh, the reduction procedure allows for

a corresponding reduced force vector to be derived (see Section 3.3.6). The static equilibrium equation then becomes

$$\tilde{\mathbf{K}}\tilde{\mathbf{q}} = \tilde{\mathbf{f}}. \tag{84}$$

The system in Eq. 84 is computationally easier to solve than the full structural system.

Thus, at each iteration of the optimization process, instead of solving the full structural problem, the following steps can be performed: first, the reduction matrix $\mathbf{R}$, $\tilde{\mathbf{K}}_{\text{workpiece}}$ and $\tilde{\mathbf{f}}$ are computed once, as they do not depend on the design variables $\mathbf{x}$; then, at the current design point $\mathbf{x}^k$, $\tilde{\mathbf{K}}_{\text{bars}}$ is computed; next, the reduced stiffness matrix is constructed from Eq. 83; following this, the equilibrium equation in Eq. 84 is solved; and finally, the objective function is evaluated using the structural displacement vector, $\mathbf{q} = \mathbf{R}\tilde{\mathbf{q}}$.

In addition to the considerations mentioned above, the sensitivity analysis must also be slightly modified. Starting with the expression for the derivative of the objective function with respect to the design variables, as given in Eq. 55, and noting that the reduction matrix $\mathbf{R}$ is independent of the design variables, we have:

$$\frac{dc_0}{dx_i} = \mathbf{b}^T\mathbf{R}\frac{\partial\tilde{\mathbf{q}}}{\partial x_i}. \tag{85}$$

Next, by differentiating Eq. 84 with respect to x_i, considering that the external force is independent of the design variables and the reduced stiffness matrix of the workpiece is also unaffected by the design variables, we obtain:

$$\frac{dc_0}{dx_i} = -\boldsymbol{\Lambda}^T\frac{\partial\tilde{\mathbf{K}}_{\text{bars}}}{\partial x_i}\tilde{\mathbf{q}}, \tag{86}$$

where the derivative of the reduced stiffness matrix of the bars can be easily determined from Eq. 58. The vector $\boldsymbol{\Lambda}$ is obtained by solving the system:

$$\tilde{\mathbf{K}}\boldsymbol{\Lambda} = \mathbf{R}^T\mathbf{b}. \tag{87}$$

To fully make use of the benefits of the reduction method, it is essential that the reduction is performed only once at the beginning of the optimization process. This initial step yields the reduction matrix $\mathbf{R}$ and the reduced stiffness matrix of the workpiece, $\tilde{\mathbf{K}}_{\text{workpiece}}$. Performing the reduction at each iteration would defeat its purpose, as the computational cost of reducing the system is comparable to that of solving the full equilibrium equation. Therefore, reduction is executed once, after which only the reduced system is solved in subsequent iterations.

The effectiveness of this approach lies in the fact that the reduced stiffness matrix associated with the bars, $\tilde{\mathbf{K}}_{\text{bars}}$, can be assembled efficiently. This is because the reduced displacement vector $\tilde{\mathbf{q}}$ corresponds directly to the retained DOFs, allowing the bar stiffness contributions to be easily added to the appropriate entries during assembly. The same holds true for $\partial\tilde{\mathbf{K}}/\partial x_i = \partial\tilde{\mathbf{K}}_{\text{bars}}/\partial x_i$.

3.6.4 Optimization

As previously mentioned, the optimization process is handled by an external Python module that implements the MMA (Deetman 2024). This subsection provides a brief overview of the module, its usage, and the problem formulation it employs.

To use the module, a function defining the optimization problem is required. This function must take the current iteration point $\mathbf{x}^k$ as its argument and return the following:

1. The value of the objective function at the current point, $c_0(\mathbf{x}^k)$,
2. The values of the derivatives of the objective function with respect to the design variables, $\partial c_0(\mathbf{x}^k)/\partial x_i$, arranged in a column vector,
3. The values of the constraint functions, $c_j(\mathbf{x}^k)$ $(j = 1, \ldots, m)$, also arranged in a column vector,
4. The values of the derivatives of the constraint functions with respect to the design variables, $\partial c_j(\mathbf{x}^k)/\partial x_i$, arranged in matrix form.

This function is then utilized within an iterative loop, where MMA subproblems are formulated and solved for consecutive design points (see Section 3.5). The process continues until either a convergence criterion based on the Karush-Kuhn-Tucker (KKT) conditions is met or the maximum number of iterations is reached.

The problem formulation implemented in the code is based on Svanberg (2007) and is given by:

$$\begin{aligned} \min_{\mathbf{x}} \;\; & f_0(\mathbf{x}) + a_0 z + \sum_{j=1}^{m} \left(c_j y_j + \frac{1}{2} d_j y_j^2 \right) \\ \text{s.t.} \;\; & f_j(\mathbf{x}) - a_j z - y_j \leq 0 \quad (j = 1, \ldots, m) \\ & \underline{x}_i \leq x_i \leq \bar{x}_i, \quad \mathbf{y} \geq 0, \quad z \geq 0 \; , \end{aligned} \tag{88}$$

where the constants satisfy $a_0 > 0$, $a_j \geq 0$, $c_j \geq 0$, $d_j \geq 0$, and $c_j + d_j > 0$ for all $j = 1, \ldots, m$, with the additional condition $a_j c_j > a_0$ when $a_j > 0$. The variables y_j and z are artificial variables introduced to handle cases where the subproblem may be infeasible.

Svanberg (2007) provides recommended values for a_0, a_j, c_j, and d_j for various types of optimization problems. The relevant cases are summarized below:

- **For the general formulation:**

$$\begin{aligned} \min_{\mathbf{x}} \;\; & f_0(\mathbf{x}) \\ \text{s.t.} \;\; & f_j(\mathbf{x}) \leq 0 \quad (j = 1, \ldots, m) \\ & \underline{x}_i \leq x_i \leq \bar{x}_i \; , \end{aligned} \tag{89}$$

  an equivalent formulation can be obtained by selecting $a_0 = 1$, $a_j = 0$, $d_j = 1$, and choosing c_j to be a large number. If the original problem is feasible, then all y_j values will be zero in the final solution. Otherwise, some $y_j > 0$, indicating an infeasible problem, in which case the algorithm returns a solution that is as feasible as possible.

- **For min-max formulations:**

$$\begin{aligned} &\min_{\mathbf{x}} \max_{j=1,\dots,p} \{h_j(\mathbf{x})\} \\ &\text{s.t.} \quad g_j(\mathbf{x}) \leq 0 \quad (j = 1, \dots, q) \\ &\qquad \underline{x}_i \leq x_i \leq \bar{x}_i \ , \end{aligned} \tag{90}$$

  the problem becomes equivalent to Eq. 88 by taking:

  - $f_0(\mathbf{x}) = 0$
  - $f_j(\mathbf{x}) = h_j(\mathbf{x})$ for $j = 1, \dots, p$
  - $f_{p+j}(\mathbf{x}) = g_j(\mathbf{x})$ for $j = 1, \dots, q$
  - $a_0 = a_1 = \dots = a_p = 1$, $a_{p+1} = \dots = a_m = 0$
  - $d_j = 1$ for $j = 1, \dots, m$
  - c_j is a large number for $j = 1, \dots, m$

  where $m = p + q$.

These parameter values are essential for correctly setting up and utilizing the Python module. However, note that after running simulations, it has been observed that the choice of problem formulation has no impact on the final results, whether using only the constraint on the sum of the design variables x_i, or also including the constraint on the first eigenfrequency. The only noticeable difference lies in the convergence time: the min-max formulation takes longer to converge. This is likely due to the fact that the h_j functions represent the system's DOFs, which are numerous and thus increase the computational load. Therefore, the general formulation has been adopted.

When using the code, it is advisable to scale the constraints, objective function, and design variables to mitigate potential numerical errors. Additionally, care should be taken when selecting values for c_j since excessively large values can lead to numerical problems. If the final solution contains non-zero values of y_j, this indicates that the solution is infeasible. In such cases, the values of c_j should be progressively increased to impose a stronger penalty on the artificial variables and drive the solution closer to feasibility (Svanberg 2007).

As said above, the code employs a convergence criterion based on the KKT conditions (see Section 1.3), where convergence is determined by evaluating the residual against a specified tolerance ε_{KKT}. For further details on the practical implementation of the MMA algorithm used in the module, interested readers are referred to (Svanberg 2007).

4 Results and Discussion

The MMA-based optimization algorithm has been applied to a set of case studies to illustrate the quality of the results and to examine the influence of various parameters on the final design. To ensure clarity and consistency, Table 4 summarizes the main notations used throughout the results section.

Notation	Explanation
x_i	Design variable associated with bar element i.
n	Total number of design variables.
V	Upper bound constraint on the sum of design variables ($\sum_i x_i \leq V$).
ε_{KKT}	Convergence tolerance based on the KKT residual norm.
p	SIMP penalization exponent ($E_i = x_i^p E_0$).
α	Bar stiffness ratio ($E_b = \alpha E_w$).
$\frac{4}{n}\sum_i x_i(1-x_i)$	Normalized metric to quantify the amount of intermediate values in the solution: equals 0 for fully binary solutions and 1 when all $x_i = 0.5$.

Table 4: Summary of notations used in the results section.

More specifically, the program has been applied to two different geometries. The first is a simple cylindrical workpiece, serving primarily as an academic example to explore the behavior of the algorithm in a controlled setting. The second is a more realistic impeller workpiece, chosen to demonstrate the method's applicability to practical engineering scenarios.

4.1 Cylinder Workpiece

In this section, the optimization code is applied on a simple academic example consisting of a cylindrical workpiece. At first, the problem has been described, then simulations have been run for different parameters and results have been discussed.

4.1.1 Problem description

Geometry: The geometry consists of a cylindrical workpiece with a height of 100 [mm] and a diameter of 50 [mm].

Materials: The cylinder's material is an aluminum alloy with a Young's modulus $E_w = 73.119$ [GPa] and a Poisson ratio $\nu_w = 0.33$. The characteristics of the bar elements correspond to a steel material, described in Section 3.2.4. The effect of the bar stiffness ratio α has been analyzed afterward.

Loading: Two different load cases have been considered: an external tangential load of 10 [kN] and a radial load of 10 [kN], both applied at $(x, y, z) = (0, -25, 75)$ [mm].

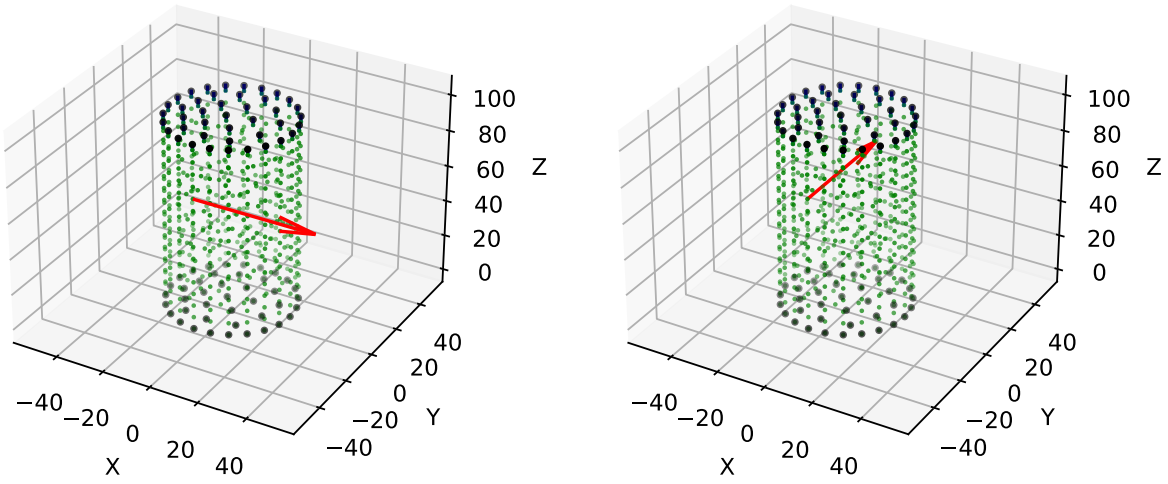
Mesh: A finite element mesh of the workpiece has been generated using automatic meshing tools in Siemens NX. The mesh data has then been imported into the Python code for

further processing. It consists of around 2,000 tetrahedral elements, which are described in Section 3.3.1.

Boundary conditions: The nodes on the lower surface of the cylinder ($z = 0$) are fixed. This constraint is essential to properly restrain the system and ensure that the stiffness matrix is invertible.

Bar elements: A total of 42 bar elements have been placed on the upper surface of the cylinder ($z = 100$ [mm]) to simulate the presence of potential soldering points, whose distribution has to be optimized. The free node of each bar element has also been fixed to ensure invertibility of the stiffness matrix.

Figure 23 shows the full system for both load cases, including the mesh, loads, constraints, and bars.



(a) Tangential force.

(b) Radial force.

Figure 23: Mesh of the cylindrical workpiece with fixed nodes, bar elements and two load cases. Element edges are omitted for clarity. All dimensions in [mm].

Objective: The optimization problem, given by Eq. 16, consists in determining the optimal clamping location on the upper face of the cylinder, for two different load cases shown in Figure 23.

Extreme Cases: To assess the full potential for improvement, simulations have been conducted for the two extreme configurations: $x_i = 0$ and $x_i = 1$ for all $i = 1, \ldots, n$. These represent the lower and upper bounds of the design space. Table 5 summarizes the resulting maximum displacements for both tangential and radial loading conditions.

The values in Table 5 set realistic expectations for the optimization process by identifying the best-case scenario that can be achieved.

	Worst case ($x_i = 0$)	Best case ($x_i = 1$)
Tangential force	115 μm	80.6 μm
Radial force	96.5 μm	51.4 μm

Table 5: Maximum displacement results for the extreme cases.

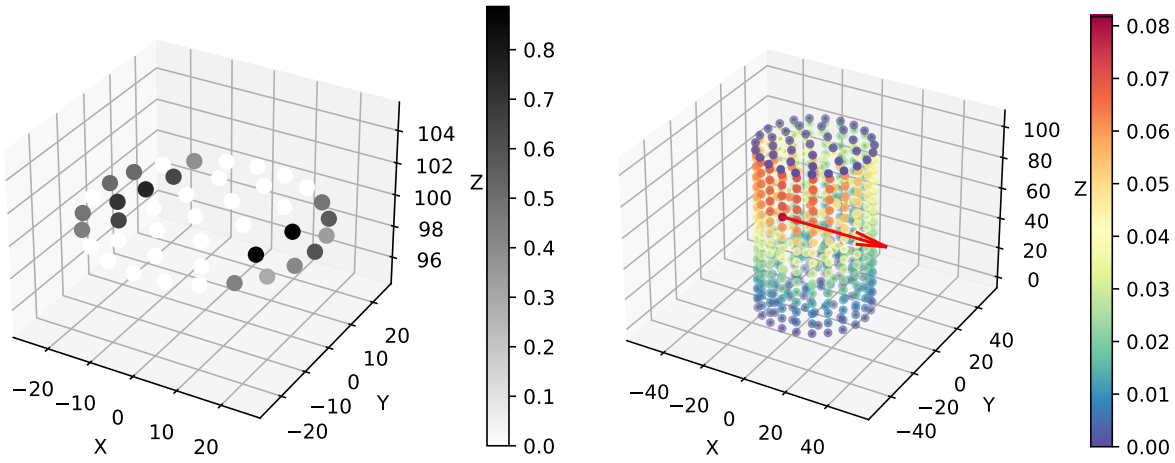
Remark: The generalized displacement vector $\mathbf{q}$, obtained as a function of the design variables $\mathbf{x}$, has been multiplied by an amplifying factor 10^6 to avoid having to deal with very small numbers during the optimization. The end solution is obviously not affected by this change. It should also be noted that the realism of the obtained displacement values is not assessed. As already mentioned, the use of tetrahedral elements is known to be inferior to other options. Hence, this study is intended as a proof-of-concept.

4.1.2 Results

Simulation results for the problem described in Section 4.1.1 are presented in this section. Chosen parameters are: $V = 10$, $p = 3$, $\varepsilon_{\text{KKT}} = 10^{-3}$ and $\mathbf{x}^0 = \mathbf{0}$.

Load case 1 - Tangential force

Figure 24 shows the obtained solutions for the first load case and Figure 25 shows the evolution of objective and constraint functions during the iterations.



(a) Optimal distribution.

(b) Displacement field.

Figure 24: Optimal clamping point distribution and associated displacement field for the first load case. All dimensions in [mm].

As seen in Figure 24, the solution exhibits a distribution of points on two opposite sides of the cylinder's upper surface. This suggests that these clamping areas are the most critical for reducing displacement in the workpiece under the given load. In fact, the algorithm retains only the most effective bar elements, discarding the less efficient ones. Interestingly, not all points are binary (0 or 1); some exhibit intermediate values. This suggests that the algorithm

favors a more distributed clamping configuration, with a larger overall contact surface, rather than concentrating stiffness at a few specific locations ($x_i = 1$). Although the SIMP model with $p = 3$ encourages values to move toward the bounds (0 or 1), this regularization appears insufficient, as the algorithm still prefers a more spread-out solution.

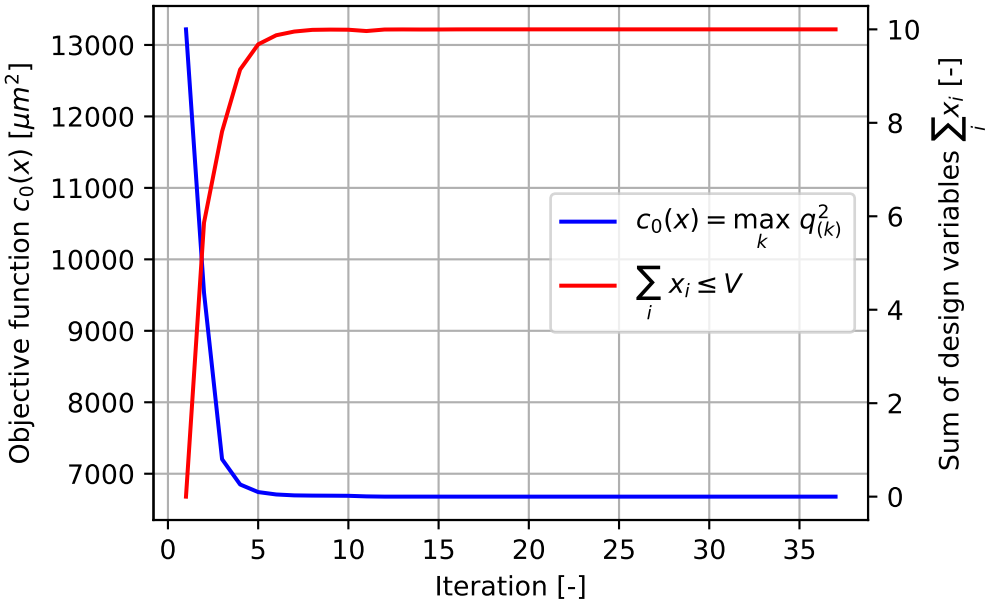


Figure 25: Evolution of the objective and constraint functions during the optimization process for the first load case.

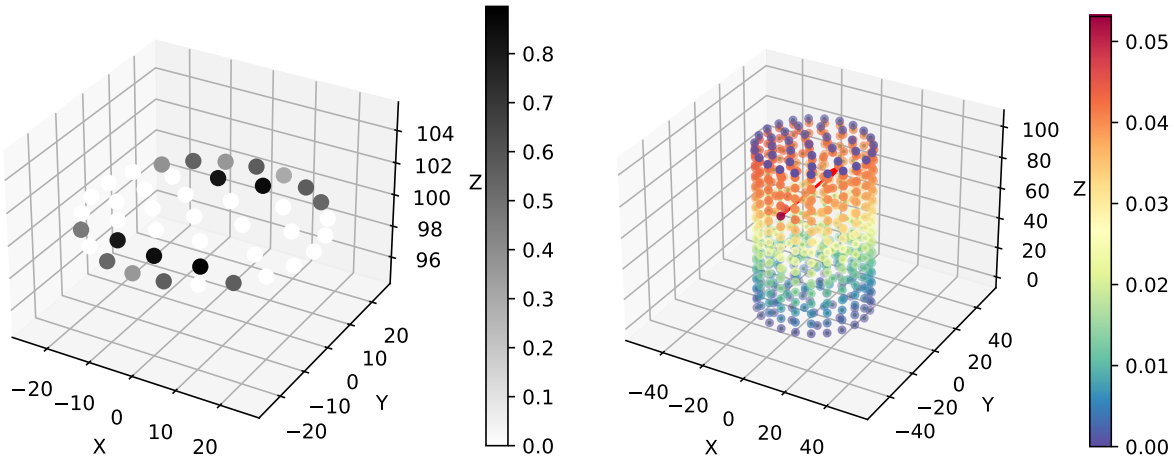
In Figure 25, the objective function rapidly converges to its minimum value within the first five iterations, while the sum of the design variables approaches the upper bound $V = 10$. The subsequent iterations are primarily dedicated to satisfying the KKT optimality conditions within the desired tolerance. The process converged in 37 steps (i.e. function evaluations) taking 108.5 seconds.

To address the intermediate values and enforce that all points take on binary values, the penalization parameter can be further increased to encourage stricter convergence toward the bounds. This approach is explored in an upcoming section.

The maximum displacement within the cylindrical workpiece for the obtained distribution is 81.8 [μm], which is very close to the best-case scenario, corresponding to a uniform distribution with $x_i = 1$ for all i, yielding 80.6 [μm].

Load case 2 - Radial force

Figure 26 shows the obtained solutions for the second load case and Figure 27 shows the evolution of objective and constraint functions during the iterations.



(a) Optimal distribution. (b) Displacement field.

Figure 26: Optimal clamping point distribution and associated displacement field for the second load case. All dimensions in [mm].

As seen in Figure 26, the optimal clamping areas for the radial load have the same shape as before, but rotated by 90 degrees around the cylinder axis. Once again, the most critical regions are located toward the outer edge rather than the interior surface. Additionally, the solution exhibits intermediate values.

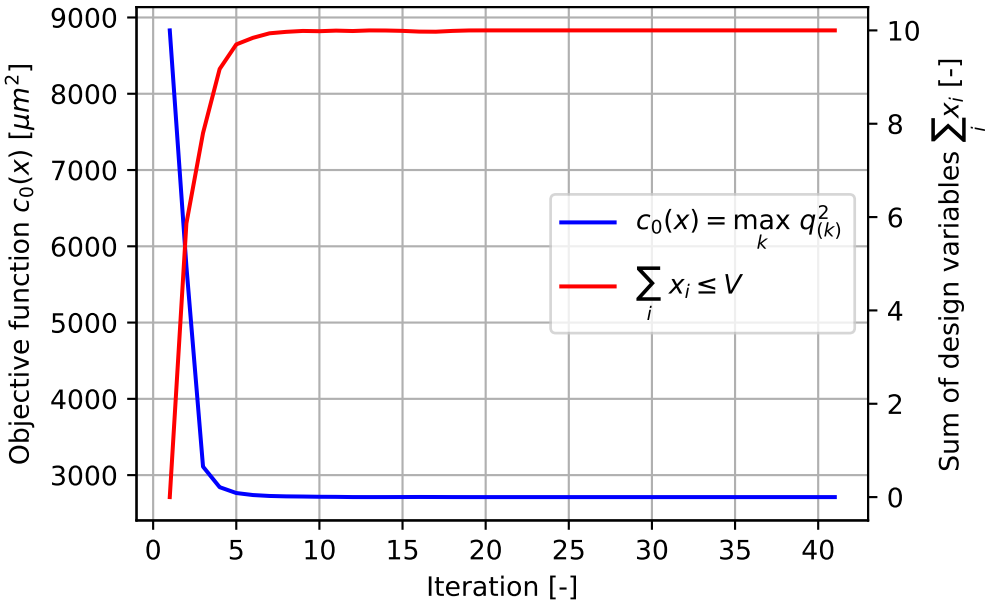


Figure 27: Evolution of the objective and constraint functions during the optimization process for the second load case.

The process converged in 41 iterations taking 124 seconds in total. The evolution of the involved functions, shown in Figure 27, is similar to the previous case.

4.1.3 Effect of SIMP penalization parameter

To study how the SIMP penalization parameter introduced in Eq. 4 influences the results, simulations have been carried out for $p = 1, 2, \ldots, 7$. In particular, we aim to determine whether it is possible to reduce or eliminate intermediate values in the solution by forcing the x_i's toward their bounds. Chosen parameters are: $V = 10$, $\varepsilon_{\text{KKT}} = 10^{-3}$ and $\mathbf{x}^0 = \mathbf{0}$.

To quantify the 'amount' of intermediate values, a metric that can be used is the following:

$$\frac{4}{n}\sum_{i=1}^{n} x_i(1 - x_i). \tag{91}$$

The expression in Eq. 91 is normalized such that it is equal to 0 when all design variables $x_i \in \{0, 1\}$ and equal to 1 when $x_i = 0.5$, $\forall i = 1, ..., n$. Figure 28 shows the evolution of this metric as p increases, while Figure 29 displays the clamping point distributions in some instances.

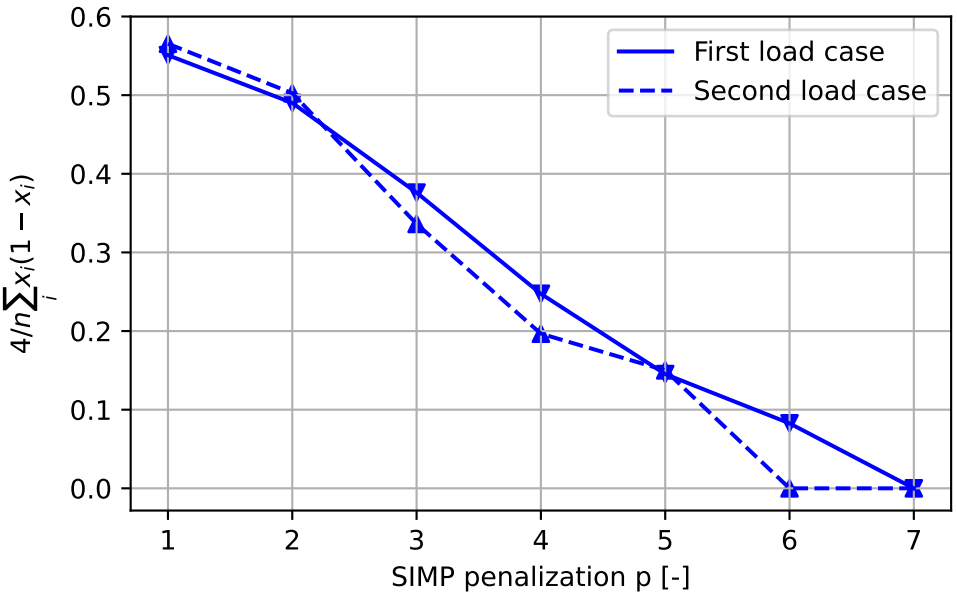
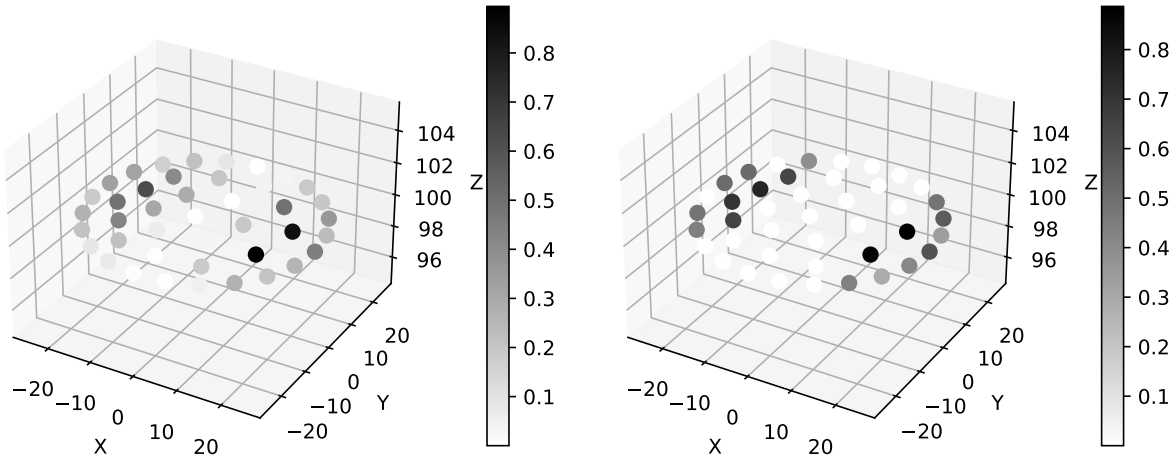


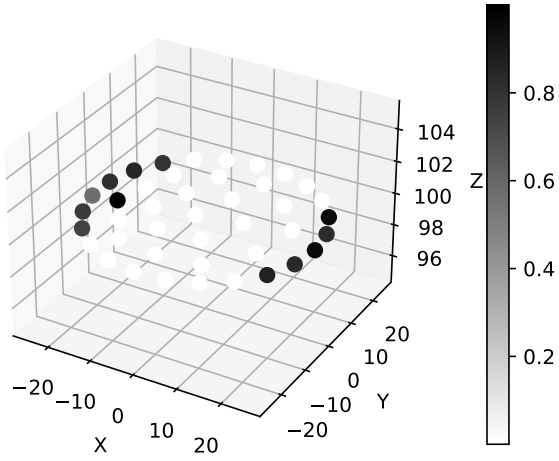
Figure 28: Metric quantifying intermediate values in the solution as a function of the SIMP penalization p.

As shown in Figure 28, increasing the penalization parameter has the effect of penalizing intermediate values, thereby forcing the design variables toward 0 or 1. The larger the value of p, the more 'distinct' the solution becomes, as it is increasingly defined. For $p = 7$, the solutions exhibit binary solutions in both load cases (see Figure 29). For smaller p, this is not the case, which suggests that the algorithm finds it more advantageous to spread out the clamping points rather than increasing stiffness of individual bar elements.

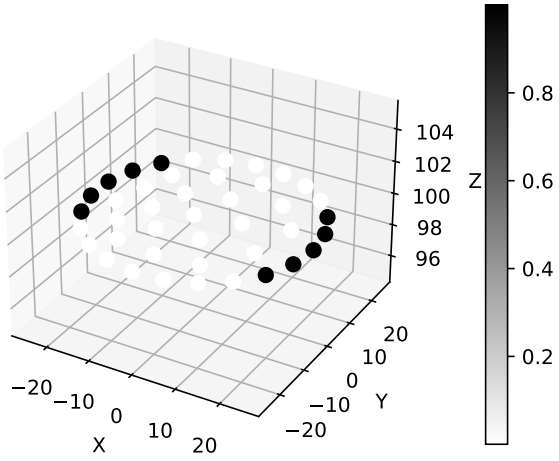


(a) $p = 1$.

(b) $p = 3$.



(c) $p = 5$.



(d) $p = 7$.

Figure 29: Optimal clamping point distributions for several p and for the first load case. All dimensions in [mm].

Figure 30 shows the evolution of the Young's modulus of individual bar elements as a function of their corresponding design variables. As discussed in a previous section, increasing the parameter p in the SIMP homogenization scheme accentuates the slope of the curve representing the bar's Young's modulus as a function of the design variables. Specifically, the slope becomes steeper for values of x_i close to 1 and shallower near 0. As a result, the potential gain in stiffness relative to the cost of increasing x_i becomes more significant as p becomes larger. This observation explains the behavior noted above.

In structural analysis, evaluating the objective function (i.e. FEA computations) can be computationally expensive, making it essential to minimize its cost. Specifically, evaluating the objective function for a given design vector $\mathbf{x}^k$ involves determining the structural stiffness matrix $\mathbf{K}$ and solving the static equilibrium equations. These operations are computationally demanding, especially for large-scale problems. As such, the number of objective function evaluations becomes a critical metric that must be optimized in the context of structural op-

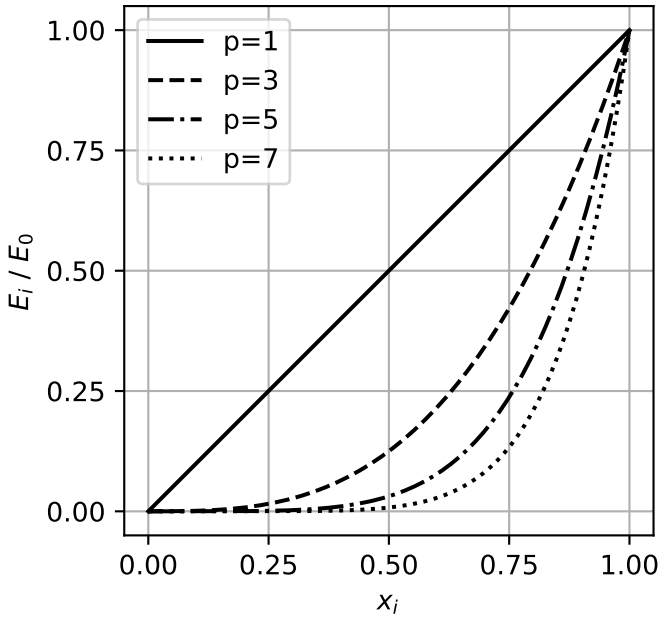


Figure 30: Young's modulus of bar elements as a function of the design variables under the SIMP model ($E_i = x_i^p E_0$).

timization. Figure 31 illustrates the evolution of the number of objective function calls as a function of p.

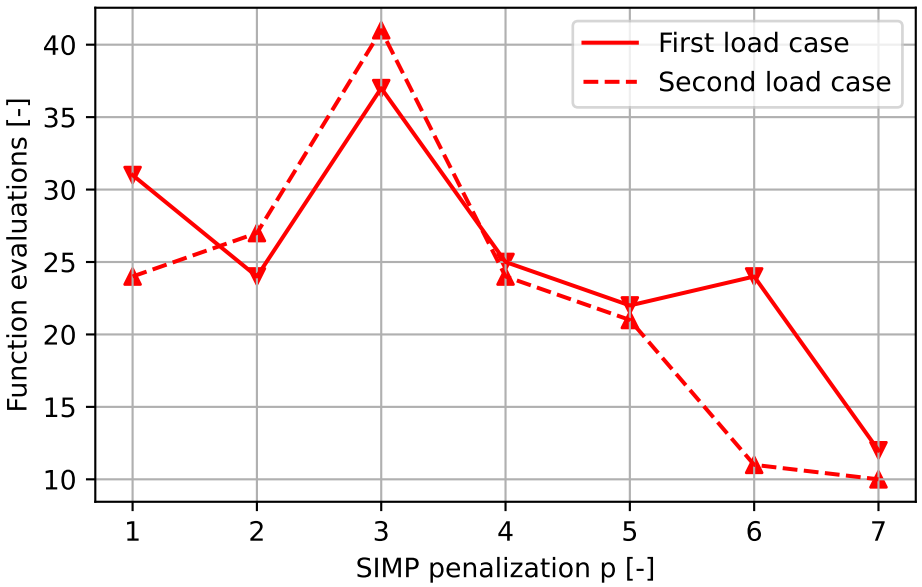
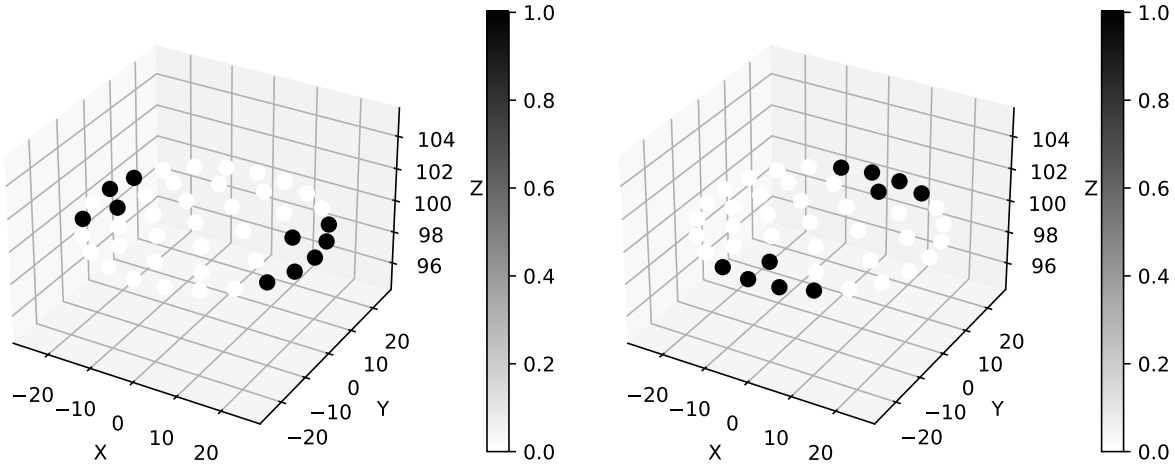


Figure 31: Number of objective function calls as a function of the SIMP penalization parameter p.

As seen in Figure 31, there is no clear correlation between the number of function evaluations required by the MMA algorithm and p. However, there appears to be a tendency for the number of function calls to decrease for larger p. As the penalization parameter increases, a slight increase in the maximum component of $\mathbf{q}$ has been observed for the final solution. This can be attributed to the fact that the clamping locations become less dispersed. Ultimately, choosing a penalization exponent p larger than 3 appears to be better adapted for this problem.

4.1.4 Comparison with the SLSQP algorithm

The optimization for both load cases has been carried out with $p = 3$, using the `minimize` function from `SciPy` and the Sequential Least Squares Programming (SLSQP) method. Interestingly, the solver produced a more binary solution compared to MMA, as illustrated for $V = 10$ in Figure 32.



(a) Tangential load.

(b) Radial load.

Figure 32: Optimization results with SciPy using the Sequential Least Squares Programming (SLSQP) method for both load cases.

Although the solution obtained is more defined, which is desirable, it has been observed that the number of objective function evaluations is substantially higher than with MMA. The SLSQP method required 129 and 87 evaluations for the first and second load cases respectively, compared to only 37 and 41 for MMA. It is also important to note that this method does not allow for defining a convergence criterion based on the KKT residual.

The increased number of FEA computations can be attributed to the fact that `SciPy` must numerically evaluate the gradient of the objective function, unlike MMA where the gradients are obtained directly from the sensitivity analysis and thus do not require significant additional computation (see Section 3.4).

4.1.5 Mesh convergence study

To gain a deeper understanding of the algorithm's behavior as the mesh is refined, a mesh convergence study has been performed using the same cylinder geometry as described previously. Chosen parameters are: the first load case, $V = 10$, $\varepsilon_{\text{KKT}} = 10^{-3}$, $p = 5$ and $\mathbf{x}^0 = \mathbf{0}$.

Figure 33 illustrates how the computation time varies with the number of DOFs in the cylinder mesh; the corresponding number of elements is also shown for reference.

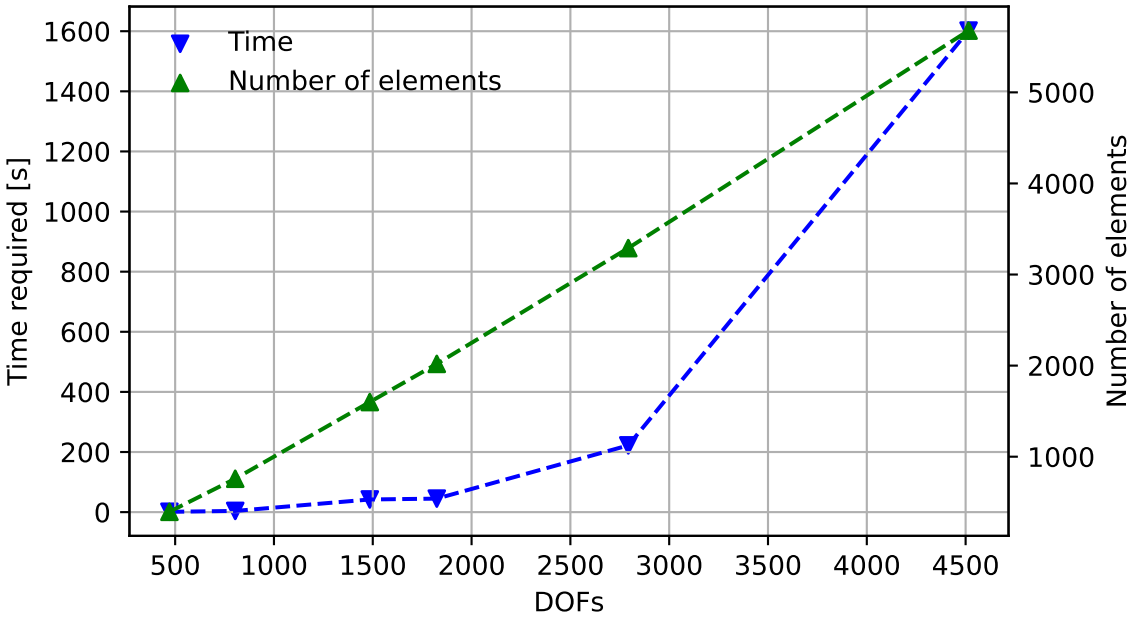


Figure 33: MMA computation time as a function of the number of degrees of freedom in the cylinder workpiece mesh. Computations have been performed using a 'full' matrix solver, rather than a 'sparse' solver, which affects the scaling of computation time.

As shown in Figure 33, the computation time required by the algorithm increases significantly with the number of DOFs. A particularly sharp rise is observed between 2,800 and 4,500 DOFs, where the simulation time reaches approximately 26 minutes. This duration is excessive for smaller-scale applications like the one considered here. The observed increase can be attributed to the growing size of the structural matrices, which makes the assembly process and the matrix inversion of $\mathbf{K}$ increasingly computationally expensive. It is important to note that a 'full' matrix solver has been used to solve $\mathbf{Kq} = \mathbf{f}$, which makes the inversion less efficient compared to 'sparse' solvers.

Interestingly, it has been observed that the number of objective function evaluations is not affected by the mesh refinement, which indicates that the number of function calls is primarily influenced by other factors, such as the KKT tolerance, the initial point $\mathbf{x}^0$, the optimization algorithm used or the number of design variables. On the other hand, the average time per iteration increases gradually.

4.1.6 Applying reduction methods

As mentioned earlier, refining the mesh increases the number of DOFs, leading to larger system sizes. As a result, the structural matrices become larger, which in turn increases the computational cost (time) of solving the static equilibrium equations.

However, it is important to note that most of the significant behavior occurs not inside the cylinder but on the external surfaces of interest, such as the region where the external load is applied and the top surface where the bars are located. From a practical perspective, the detailed behavior inside the cylinder is less critical; what matters most is understanding the relationship between the applied force, the largest displacement (likely occurring at the node where the force is applied), and the top surface. Consequently, detailed computation of the mesh inside the cylinder is unnecessary, which is why reduction methods can be employed. A great advantage in using static condensation (or Guyan-Irons) consists in the fact that the method leads to exact results in linear static analysis (not true for the eigenfrequencies). Hence, the computation time can be reduced significantly.

Section 3.6.3 details the procedure to introduce reduction in the problem formulation. This approach enables the use of significantly finer meshes while reducing computation time compared to using the full structural matrices (see Figure 33). To evaluate the efficiency gains provided by the Guyan-Irons method, an additional mesh convergence analysis has been conducted. The results are presented in Figure 34.

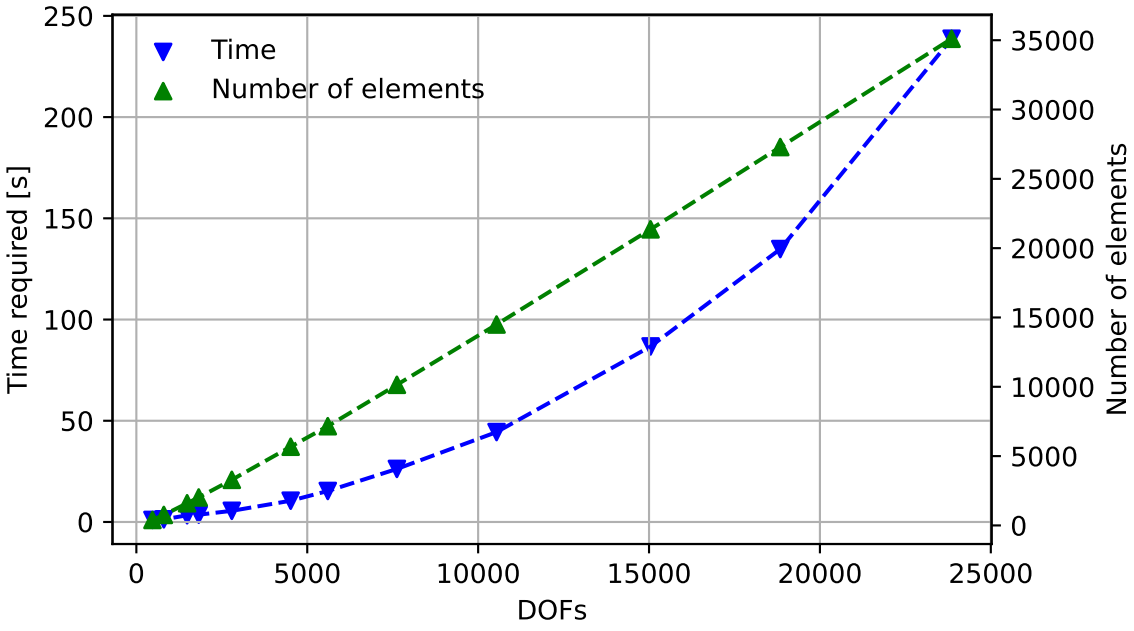


Figure 34: MMA computation time as a function of the number of degrees of freedom in the cylinder workpiece mesh, for the reduced case.

As shown in Figure 34, a larger-scale problem with approximately 27,000 DOFs, when reduced, requires around 4 minutes of computation time, which is comparable to that of the unreduced model with roughly 2,800 DOFs. The resulting decrease in computation time is

therefore substantial and proves highly beneficial for the present work. Note also that most of the computational effort is dedicated to reducing the workpiece mesh from $\mathbf{K}_{\text{workpiece}}$ to $\tilde{\mathbf{K}}_{\text{workpiece}}$ before the optimization process actually begins.

4.1.7 Constraint on the first eigenfrequency

As discussed in Section 3.2.1, it is also possible to impose constraints on the system's eigenfrequencies. In practical applications, it is important to avoid workpiece-fixture systems with low eigenfrequencies, as these can resonate with the spindle's cutting frequency. Such resonance can lead to poor surface finish or even prevent the machining process from meeting the tolerance requirements specified by the customer. Therefore, it is essential to ensure that the system avoids this type of dynamic instability.

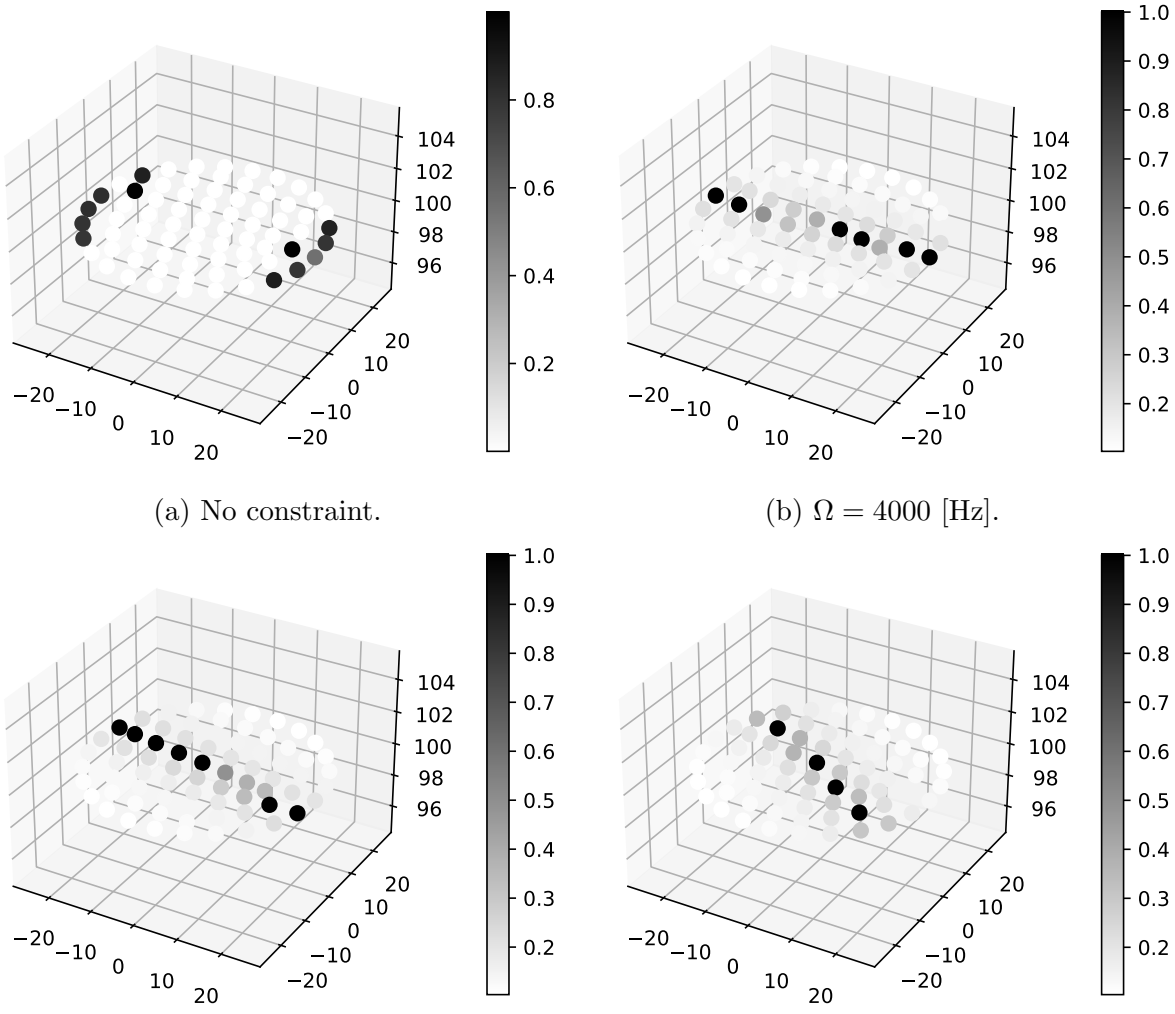
A straightforward way to enforce this is by requiring the first (i.e., lowest) natural frequency ω_1 to exceed a certain threshold ($\geq \Omega$). Since all higher modes naturally occur at higher frequencies, ensuring that the first eigenfrequency lies above a critical value suffices to avoid problematic resonance during machining.

The spindle's cutting frequency depends on multiple factors, including depth of cut, feed rate, tool geometry, and especially the workpiece material (Hoffmann Group 2016). Although determining the exact frequency range during an operation can be complex, approximate intervals can still provide useful insight. Appendix A lists typical frequency ranges for various workpiece materials. These values directly inform the eigenfrequency constraint, which is typically set to lie above the upper bound of the expected cutting frequency range, possibly with an added safety margin. In the specific case of an aluminum cylinder, the first eigenfrequency should be above approximately 2,000 [Hz] to avoid resonance during operation.

To evaluate the impact of introducing the second constraint $\omega_1 \geq \Omega$, the optimization problem has been reformulated in Python. Chosen parameters are: the first load case, $V = 10$, $p = 5$, $\varepsilon_{\text{KKT}} = 10^{-3}$ and a mesh consisting of approximately 2,800 degrees of freedom (or 3,290 T4 elements). To limit computational cost, the problem has been solved using a reduced-order model for the eigenfrequency calculation. Specifically, the Guyan-Irons reduction method is employed to obtain a reasonable approximation of the first eigenfrequency with significantly reduced computational effort.

Without this new constraint, the first eigenfrequency of the optimal clamping configurations has been measured at 3,756 [Hz]. Simulations have been performed to observe how the solution behaves if higher frequencies are desired, that is as the lower bound Ω is being increased. Figure 35 shows the results obtained.

As shown in Figure 35, the solutions differ significantly from the case without an eigenfrequency constraint. For lower bounds of 4,000, 4,500, and 5,000 [Hz], the resulting configurations exhibit clamping points aligned approximately along a diameter of the upper surface. This alignment varies slightly with the imposed frequency bound, resulting in a minor tilt of the clamping line. Moreover, it is noteworthy that the solution is not binary (i.e., strictly 0 or 1); instead, the values are distributed continuously.



(a) No constraint.

(b) $\Omega = 4000$ [Hz].

(c) $\Omega = 4500$ [Hz].

(d) $\Omega = 5000$ [Hz].

Figure 35: Clamping point distribution obtained when introducing the constraint $\omega_1 \geq \omega$, for the initial point $\mathbf{x}^{(0)} = \mathbf{0}$. All dimensions in [mm].

To assess the behavior of the optimization algorithm throughout the iterations, the evolution of the first eigenfrequency and the sum $\sum_i x_i$ is shown in Figure 36, for the case where $\Omega = 4000$ [Hz] and the initial design $\mathbf{x}^{(0)} = \mathbf{0}$.

As shown in Figure 36, both the eigenfrequency and the sum of design variables converge toward stable values; however, these values do not satisfy the imposed constraints. Specifically, the constraints $\sum_i x_i \leq V$ and $\omega_1 \geq \Omega$ are violated in the final solution. This behavior is consistent with the presence of non-zero artificial variables y_i at the end of the process, which are introduced to handle infeasibility in subproblems (see Section 3.6.4). As noted by Svanberg (1987), the final solution in such cases is the best approximation within the infeasible region, that is, it is 'as feasible as possible'.

Interestingly, not only are the constraints violated when the eigenfrequency constraint is included, but the resulting eigenfrequency is also *lower* than in the unconstrained case (ap-

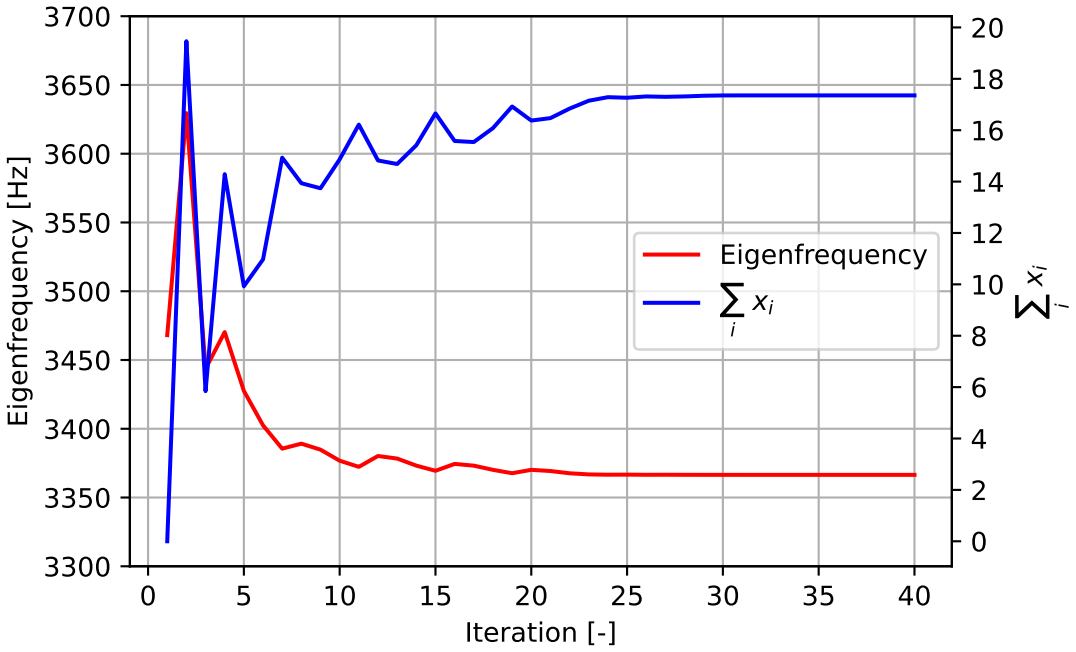


Figure 36: Evolution of the first eigenfrequency and $\sum_i x_i$ during the optimization process, starting from $\mathbf{x}^{(0)} = \mathbf{0}$, with a constraint threshold $\omega_1 \geq \Omega = 4000$ [Hz].

proximately 3,756 [Hz]). This suggests that imposing a constraint on the first eigenfrequency may result in a design with *worse* dynamic performance.

This outcome can be explained by two key factors. First, the additional constraint reduces the feasible design space, potentially to the point where no solution can simultaneously satisfy all requirements. In particular, initial points like $\mathbf{x}^0 = \mathbf{0}$ can become infeasible. Second, minimizing the maximum displacement implicitly leads to high global stiffness, which naturally leads to higher eigenfrequencies. Therefore, excluding the eigenfrequency constraint not only simplifies the problem but can also yield better results because the feasible domain is larger. For reference, the first eigenfrequency is equal to 4,964 [Hz] when all $x_i = 1$; meaning that imposing $\omega_1 > \Omega = 5{,}000$ is infeasible.

It has also been observed that different optimal solutions are obtained when using alternative initial conditions, such as $\mathbf{x}^0 = [V/n \; V/n \; \ldots \; V/n]^T$. Moreover, further penalizing the artificial variables by increasing the numerical parameters c_j did not resolve this issue, indicating that the optimization outcome is sensitive to the choice of initial design.

4.1.8 Influence of bar stiffness ratio

In the initial setup, the cylindrical workpiece has been modeled as aluminum while the bars were assigned properties of steel (corresponding to a stiffness ratio $\alpha \approx 2.8$). To make the bar properties more flexible and directly dependent on the workpiece material, a bar stiffness ratio α has been introduced, defined as $E_b = \alpha E_w$, where E_b is the Young's modulus of the bars and E_w that of the workpiece.

To investigate how the stiffness ratio α influences the distribution of bars in the optimized solution, a series of simulations have been performed for increasing values of α. To quantify the presence of intermediate design variable values, the same metric is used as in Section 4.1.3: $4/n \sum_i x_i(1 - x_i)$. This expression is equal to zero when all design variables $x_i \in \{0, 1\}$, and reaches its maximum when $x_i = 0.5, \forall i = 1, ..., n$. Chosen parameters are: $V = 10$, $\varepsilon_{\text{KKT}} = 10^{-3}$ and $\mathbf{x}^0 = \mathbf{0}$. The results are presented in Figure 37.

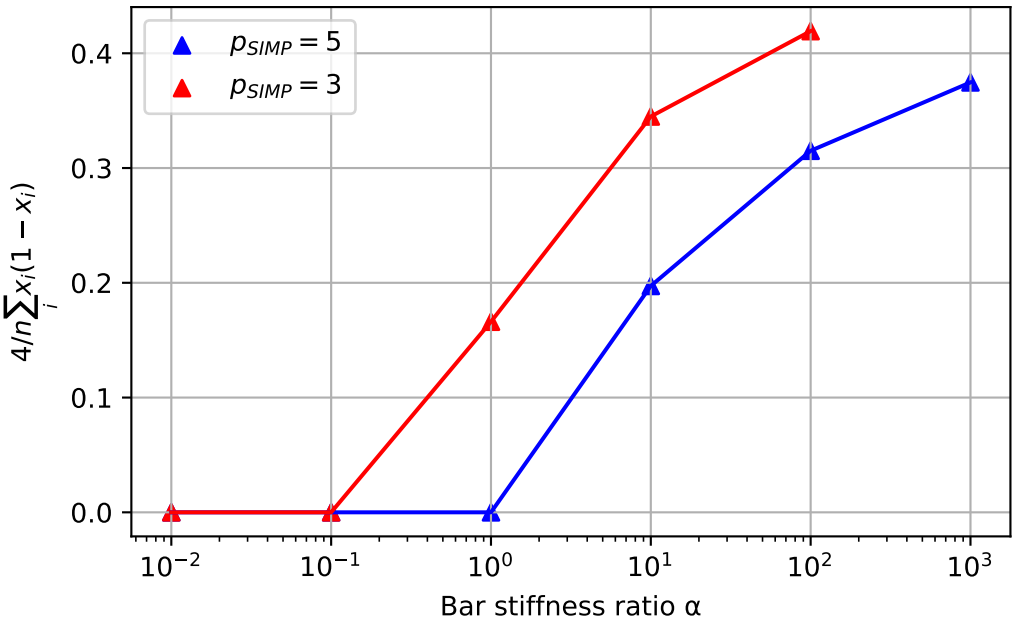


Figure 37: Influence of the bar stiffness ratio $\alpha = E_b/E_w$ on the metric $4/n \sum_i x_i(1 - x_i)$. Different SIMP penalization parameters have been considered.

As shown in Figure 37, intermediate values vanish as α decreases, that is when the stiffness of the bars is smaller or identical to that of the workpiece, the solution becomes binary. However, for larger stiffness ratios, the metric increases signaling that the solution becomes more spread out.

This behavior can be attributed to the relative stiffness between the bar elements and the workpiece. When the bar elements are significantly stiffer than the workpiece, the algorithm can distribute intermediate values more freely to reduce overall displacement. However, as the relative stiffness of the bars approaches or falls below that of the workpiece, their influence diminishes. In such cases, the additional stiffness introduced by the bars becomes less significant, and the algorithm must account for this reduced contribution when determining the optimal placement of values.

Therefore, it might be better to choose the Young's modulus E_b such that $\alpha = 1$ instead of using arbitrary properties (e.g. steel) for the bar elements.

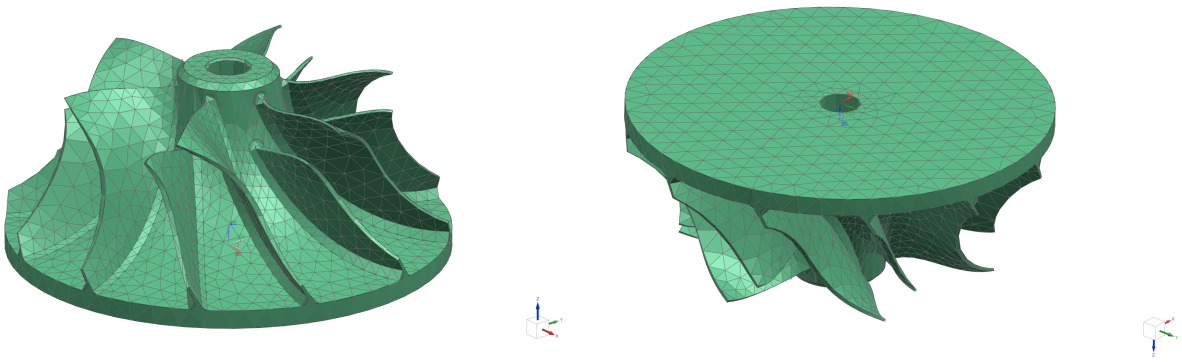
4.2 Impeller Workpiece

In this section, the optimization program is applied to a more realistic example of an impeller undergoing a milling operation. The goal is to identify an optimal clamping configuration on the outer diameter. The section begins with a description of the problem, followed by the presentation of simulation results.

4.2.1 Problem description

Geometry: To demonstrate the application of the optimization algorithm in a more practical scenario, consider the following example. We analyze a turbine impeller consisting of 12 curved blades, 6 larger blades and 6 smaller blades positioned between them. The geometry used in this study has been sourced from a publicly available CAD model on GrabCAD (Husein 2017).

Mesh: The impeller has been meshed using linear tetrahedral elements, as depicted in Figure 38. The mesh of the workpiece consists of 3,797 nodes and 12,888 elements.



(a) Upper side. (b) Lower side.

Figure 38: Tetrahedral mesh of an impeller.

Fixed nodes and bar elements: In this case study, the nodes on the bottom face of the impeller are fixed (e.g. suction cup mechanism), while clamping nodes (358 bar elements in total) have been selected on the outer diameter of the upper face, as shown in Figure 39. This particular region is more suitable for clamping, as physical clamps can be positioned there, making it a logical choice over other potential locations. The selected clamping points determine the type of solution that is found, as the outcome can vary depending on the specific points chosen for clamping.

Loading: To simulate a machining operation, it is assumed that the cutting process occurs at the end of one of the larger blades, starting from the outer diameter and moving inward. In Figure 39, the red dots represent the nodes where external machining loads are applied. In reality, machining forces are influenced by a wide range of factors. Interested readers can refer to Appendix A for more accurate estimates of cutting forces specific to their applications. It is important to note that the theoretical loads vary throughout the machining process. For

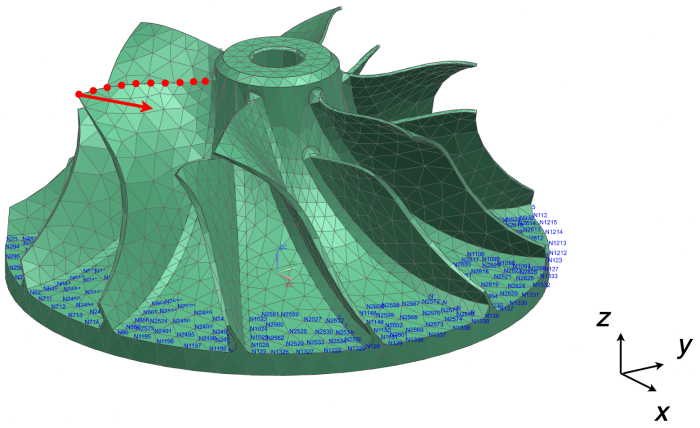


Figure 39: Selection of clamping nodes for the impeller mesh (indicated by blue labels) and machining nodes (indicated by red dots).

simplicity, we assume that the force remains constant along the tool path in this case. Thus, a load with components of 5 [kN] in both the x and y-directions is considered. The primary objective here is to demonstrate the optimization algorithm on a more realistic example, rather than to achieve high accuracy.

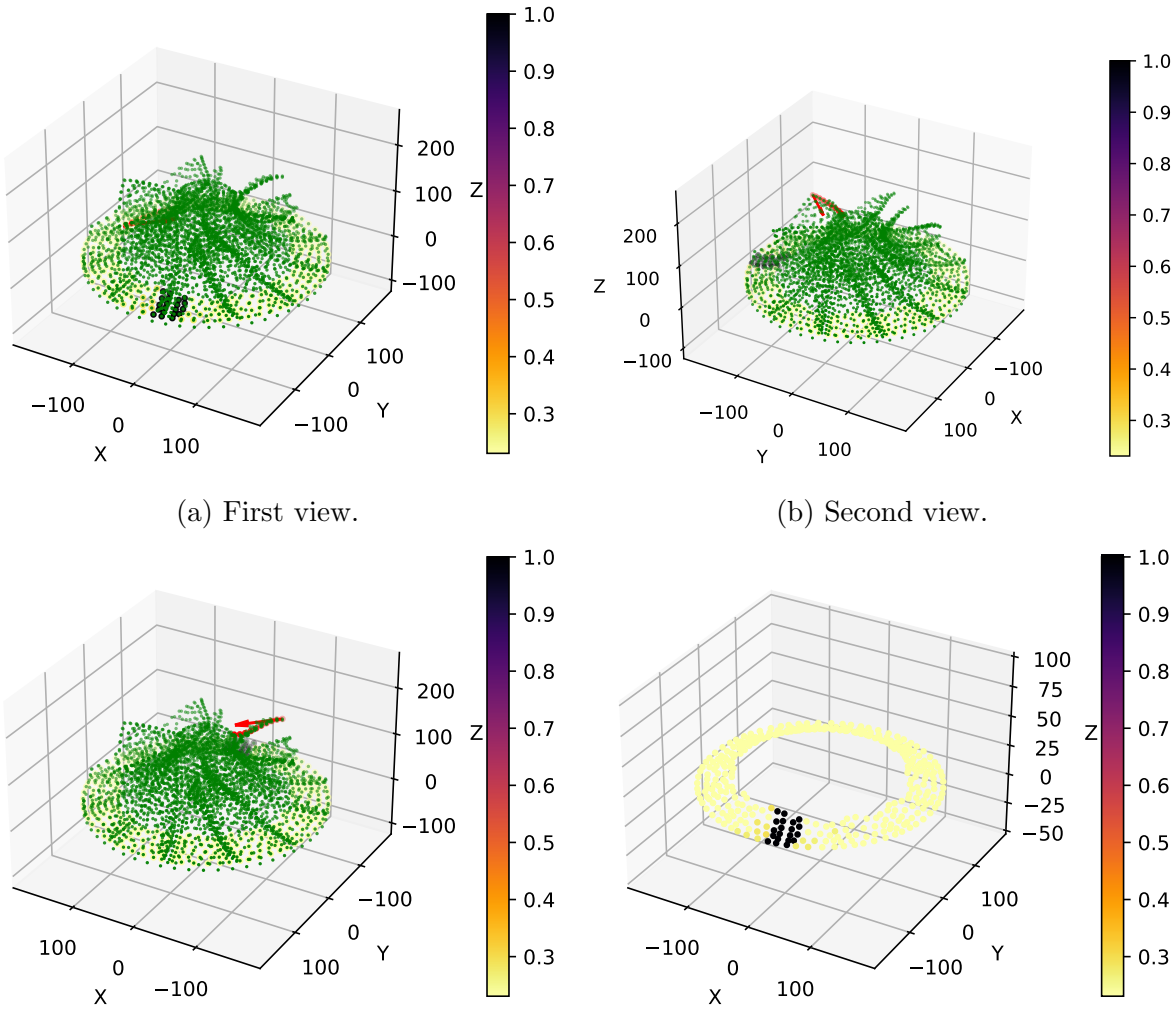
Materials: The material considered is steel with Young's modulus $E_w = 206.94$ [GPa], density $\rho = 7{,}829$ [kg/m^3] and Poisson ratio $\nu = 0.288$.

Chosen parameters: Other parameters are the following:

1. Upper bound on the number of bars in the solution: $V = 100$.
2. Bar stiffness ratio: $\alpha = 1$.
3. KKT norm residual tolerance: $\varepsilon_{\text{KKT}} = 10^{-3}$.
4. SIMP penalization parameter: $p = 5$.
5. Initial guess: $\mathbf{x}^0 = [0\ 0\ ...\ 0]^T$

4.2.2 Optimization results

This section presents the solution to the optimization problem described above. The simulation converged in approximately 3 minutes, and the resulting optimal distribution of clamping nodes is illustrated in Figure 40.



(a) First view.

(b) Second view.

(c) Third view.

(d) View without the mesh.

Figure 40: Optimal clamping point distribution for the impeller workpiece undergoing a machine tool path. All dimensions in [mm].

As shown in Figure 40, the algorithm produced an optimal distribution of clamping nodes in which several nodes near the blade subjected to loading exhibit values close to 1. Interestingly, other nodes distributed along the contour also have non-zero values (approximately 0.23), indicating a partial activation. As a result, the solution is not fully binary, contrary to the desired outcome. Figure 41 shows the evolution of the objective function and the sum of the design variables x_i as a function of the iteration number.

Figure 41 illustrates the convergence behavior of the optimization algorithm. As seen in the plot, the algorithm rapidly approaches convergence within the first 6 iterations, after which the remaining iterations are primarily used to refine the solution and satisfy the KKT optimality conditions to within the specified tolerance. The final value of the objective function is $c_0(\mathbf{x}) = 45.5 \cdot 10^{-3}$ [μm^2], which is notably small for this particular impeller geometry. This can be traced back to the use of tetrahedral elements and, especially, the elements with a large aspect ratio at the blade level. Also, it is important to note that this result is specific to the current configuration. In the case of more flexible or thin-walled workpieces, significantly

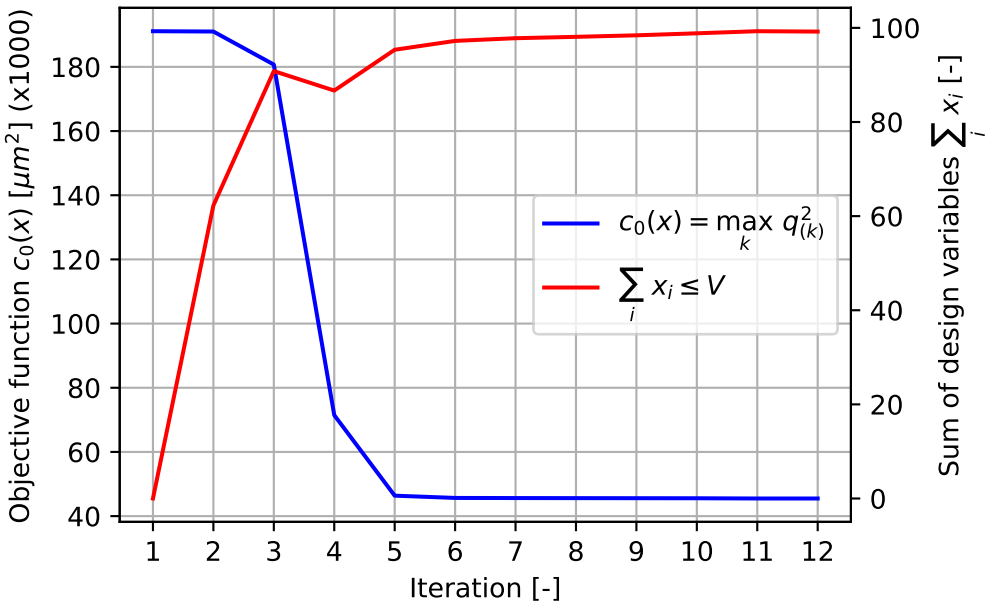


Figure 41: Evolution of the objective function and the sum $\sum_i x_i$ during the optimization process for the impeller.

larger displacements could be expected under similar loading conditions, and the importance of optimal clamping becomes more critical in those scenarios.

Since intermediate values are not practically useful (the purpose of the algorithm is to identify optimal clamping points to reduce deformation, and a result like "partially use this bar" offers little actionable guidance) such values can be removed during post-processing. A simple thresholding rule may be applied: retain a bar if its associated variable $x_i > 0.5$, and discard it otherwise. Alternatively, other strategies can be used to promote binary solutions. One approach is to reduce the stiffness ratio α, which, as shown earlier, naturally leads to more discrete designs. Another option is to further penalize intermediate values by playing directly with the objective function.

The results produced by the optimization program can serve multiple purposes. They may be used directly to guide the design of a physical fixture, or alternatively, to identify the most effective clamping regions for minimizing workpiece deformation. It is important to emphasize, however, that the quality and relevance of the solution depend strongly on how the optimization problem is formulated. In the present case, the lower surface of the workpiece has been fixed. Different choices of boundary conditions, such as fixing a different set of nodes or selecting alternative candidate clamping nodes, could lead to different optimal solutions.

4.2.3 Decreasing the bar stiffness ratio

The bar stiffness ratio α has been decreased to investigate whether the solution would become more binary as a result. To this end, additional simulations have been performed for $\alpha = 10^{-1}$ and $\alpha = 10^{-2}$. However, no significant change has been observed: the optimal distribution continued to exhibit intermediate values, particularly along the contour.

4.2.4 Comments on the eigenfrequencies

Following the post-processing step, the optimal solution was employed to compute the eigenfrequencies of the system. The first eigenfrequency was found to be approximately 28,000 [Hz]. However, this value should be interpreted with caution due to the use of tetrahedral elements in the discretization. Additionally, it is important to note that no mesh convergence study was conducted specifically for the eigenfrequency analysis in this case.

5 Conclusion and Outlook

The primary objective of this thesis has been to explore and develop methods for the optimization of machining fixtures, with a specific focus on the distribution of clamping locations on CNC-machined workpieces. In particular, the goal has been to minimize deformation during machining operations by identifying optimal clamping points. This has been achieved by formulating the problem as a structural optimization task, supported by finite element analysis (FEA) and sensitivity-based optimization methods.

A preliminary phase of the project has involved reviewing existing literature to understand the current state of the art. Many studies in this area combine FEA with machine learning techniques or rely on analytical formulations. To define a clear and manageable scope for this thesis, the decision has been made to model clamping points as virtual soldering points, implemented using bar elements.

The methodology has been implemented in Python and centers on a finite element model in which the workpiece is meshed using linear 3D tetrahedral (T4) elements, and the clamping locations are modeled via bar elements. The design variables are the normalized densities of these bar elements, where values 0 and 1 represent absence or presence of material respectively. The optimization objective is to minimize the maximum squared displacement under predefined machining loads, subject to a volume constraint on the total bar material used. A sensitivity analysis has been carried out to compute the gradients of the objective and constraints functions.

To solve the resulting problem, the Method of Moving Asymptotes (MMA), a sequential convex programming algorithm, has been employed through an available Python implementation. As the number of degrees of freedom (DOFs) in the model increases, computation time becomes a significant factor. To address this, model reduction techniques have been used, specifically the Guyan-Irons method, which is exact in the context of linear static analysis and proved to be highly effective for the task.

The optimization program has been applied to two case studies: a simple cylindrical workpiece used as an academic example, and a more realistic impeller geometry. The significant computational benefits of model reduction have been highlighted, as shown by a mesh convergence study. During the simulations, it has been observed that introducing an additional constraint on the first eigenfrequency (to avoid resonance with the machine tool) can lead to infeasibility in many cases. As a result, it has been concluded that enforcing only the volume constraint during optimization and checking eigenfrequencies in post-processing is a more robust approach.

It has been shown that the presence of intermediate values in the solution is strongly influenced by the bar stiffness ratio α and the SIMP penalization parameter p. Higher values of α and insufficient penalization tend to yield intermediate densities, which are not practically useful since they provide ambiguous guidance on whether to retain or remove a clamp. To address this, it has been shown that tuning these parameters can promote binary solutions. For cases where intermediate values still appear (e.g., along the impeller's contour), a simple

post-processing step using a threshold criterion (e.g., $x_i > 0.5$) can be applied to recover a practical fixture configuration.

While the proposed method demonstrates promising results, it is important to recognize its limitations. The use of linear tetrahedral elements is known to be less accurate than higher-order or hexahedral elements. Additionally, the static workpiece mesh is assumed to remain unchanged throughout the optimization, which does not account for the material removal that occurs during machining. Therefore, this work is best regarded as a proof-of-concept rather than a finalized solution.

There are several directions for future research. One extension is to account for the changing geometry of the workpiece due to material removal during machining. This could lead to more realistic fixture designs. Additionally, more accurate finite elements (such as second-order or hexahedral elements) could be used to improve simulation fidelity. Validation of the proposed method on a physical test case would also be a valuable next step. Finally, other optimization algorithms, such as Globally Convergent MMA (GCMMA), or even nongradient-based methods, could be explored to improve robustness and solution quality.

Overall, this work provides a foundation for fixture optimization using structural topology optimization. It shows that finite element modeling, combined with optimization techniques, can yield insights into how and where to clamp workpieces to minimize deformation during machining.

References

Barreau, V., Denimal, E., & Salles, L. (2022). "Topological optimisation and 3D printing of a Bladed disc". ASME TurboExpo 2022 - Turbomachinery Technical Conference & Exposition, Rotterdam, Netherlands.

Beckers, M. (1997). "Optimisation de structures en variables discrètes". Collection des Publications de la Faculté des Sciences appliquées n° 181, Université de Liège.

Benoist, V. (2020). "Développement d'une méthodologie pour la fabrication additive dans un contexte de fabrication de pièces métalliques par usinage". PhD thesis, Institut National Polytechnique de Toulouse.

Burri, S., & Legay, A. (2023). "Static reinforcement and vibration reduction of structures using topology optimization". Mechanics & Industry 24, 14.

Bruyneel, M., & Tossings, P. (2024). "MECA0027 - Structural and multidisciplinary optimization". ULiège lecture notes.

Calabrese, M., Primo, T., & Del Prete, A. (2017). "Optimization of machining fixture for aeronautical thin-walled components". CIRP 60 (2017) 32-37, University of Salento, Elsevier. Lecce, Italy, 27th CIRP.

Chandrupatla, T.R., & Belegundu, A.D. (2002). *Introduction to Finite Elements in Engineering.* Prentice-Hall, Third Edition.

Chen, T., & Guestrin, C. (2016). "Xgboost: A scalable tree boosting system". Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 785–794.

Cioata, V.G., Kiss, I., Alexa, V., & Ratiu, S.A. (2020). "Study of the contact forces between workpiece and fixture using dynamic analysis". Journal of Physics: Conference Series, 1426(2020) 012040.

Cioata, V.G., Kiss, I., Alexa, V., & Ratiu, S.A. (2017). "The optimization of the position and the magnitude of the clamping forces in machining fixtures". IOP Conf. Series: Materials Science and Engineering 200 (2017) 012015.

Deetman, A. (Contributor). (2024). "GCMMA-MMA-Python: Python implementation of the Method of Moving Asymptotes". 18 August 2024. Zenodo. 10.5281/zenodo.13337495

Duysinx, P. (1997). "Layout Optimization : A Mathematical Programming Approach". Report OA-41, University of Liège.

Duysinx, P. (2019). "Optimisation topologique". LTAS - Ingénieurie des Véhicules Aérospatiale et Mécanique.

Duysinx, P., & Tossings, P. (2020). "MECA0027 - Structural and multidisciplinary optimization". ULiège lecture notes.

Feng, Q., Maier, W., Stehle, T., & Möhring, H.-C. (2021). "Optimization of a clamping concept based on machine-learning". Production Engineering 16:9-22.

Fleury, C. (1989). "CONLIN: an efficient dual optimizer based on convex approximation concepts". Structural Optimization 1, 81-89 (1989), Springer-Verlag.

Géradin, M., & Rixen, D.J. (2015). *Mechanical Vibrations - Theory and Application to Structural Dynamics*. Wiley, Third Edition.

Husein. (2017). "Turbine Rotor Design (Impeller)". GrabCAD. Available at: https://grabcad.com/library/turbine-rotor-design-impeller-1. Accessed May 10, 2025.

Hoffman, E. (2012). "Jig and fixture design". Cengage Learn.

Hoffmann Group. (2016). "Zerspanungshandbuch - Grundlagen". Hoffmann GmbH Qualitätswerkzeuge, München.

Horn Group. (2023). "Fraise Carbure Monobloc". Hartmetall-Werkzeugfabrik Paul Horn GmbH, Tübingen.

Li, B., & Melkote, S.N. (2001). "Fixture Clamping Force Optimization and its Impact on Workpiece Location Accuracy". International Journal of Advanced Manufacturing Technology 17:104-113.

Liu, C., Wang, J., Zhou, B., Yu, J., Zheng, Y., & Liu, J. (2024). "Development of Fixture Layout Optimization for Thin-Walled Parts: A Review". Chinese Journal of Mechanical Engineering 37:17.

Möhring, H.-C., Wiederkehr, P. (2016). "Intelligent fixtures for high performance machining". Procedia CIRP 46(1):383-390.

Pereira, E.G.G. (2023). "Design, Dimensioning, and Optimization of a fixture for CNC machining a pump housing part". Master's thesis in mechanical engineering, Faculty of Engineering of the University of Porto.

Ponthot, J.-P. (2022). "MECA0036 - Finite Element Method". ULiège lecture notes.

Poncelet, F., Fleury, C., Remouchamps, A., & Grihon, S. (2005). "Topol: A topological optimization tool for industrial design". 6th World Congress of Structural and Multidisciplinary Optimization.

Qazani, M.R.C., Parvaz, H., & Pedrammehr, S. (2022). "Optimization of fixture locating lay-

out design using comprehensive optimized machine learning". The International Journal of Advanced Manufacturing Technology, 122:2701-2717.

Rakotondrainibe, L., Allaire, G., & Orval, P. (2020). "Topology optimization of connections in mechanical systems". Structural and Multidisciplinary Optimization (2020) 61:2253-2269.

Selvakumar, S., Arulshri, K.P., Padmanaban, K.P., & Sasikumar, K.S.K. (2010). "Clamping Force Optimization for Minimum Deformation of Workpiece by Dynamic Analysis of Workpiece-fixture System". World Applied Sciences Journal 11(7):840-846.

Svanberg, K. (1987). "The Method of Moving Asymptotes - A New Method For Structural Optimization". International Journal For Numerical Methods In Engineering, Vol. 24, 359-373.

Svanberg, K. (2007). "MMA and GCMMA - two methods for nonlinear optimization", vol 1. 2007, pp. 1-15.

Todorovic, P.M., Buchmeister, B., Djapan, M.J., Vukelic, D., Milosevic, M.D., Tadic, B., & Radenkovic, M.M. (2014). "Comparative Model Analysis of Two Types of Clamping Elements in Dynamic Conditions". Technical Gazette 21, 6:1273-1279.

Vukelic, D., Simunovic, G., Tadic, B., Buchmeister, B., Saric, T., & Simeunovic, N. (2016). "Intelligent Design and Optimization of Machining Fixtures". Technical Gazzete 23, 5, 1325-1334.

Xiong, C.-H., Wang, M.Y., & Xiong, Y.-L. (2008). "On Clamping Planning in Workpiece-Fixture Systems". IEEE Transactions on Automation Science and Engineering, Vol. 5, No. 3.

Zhang, W., An, L., Chen, Y., Xiong, Y., & Liao, Y. (2021). "Optimisation for clamping force of aircraft composite structure assembly considering form defects and part deformations". Advances in Mechanical Engineering Vol. 13(4) 1-13.

Appendix A - Cutting forces

This section focuses on analyzing the interaction between the machine tool and the workpiece, specifically in terms of the magnitude of the forces involved and the excitation spectrum. From a dynamic perspective, understanding the frequencies at play during the milling process, particularly during the finishing of functional surfaces, is essential. It is also vital to avoid resonance when machining the workpiece mounted on the printed fixture. Therefore, this section aims to provide a detailed exploration and deeper understanding of the forces at play.

Frequency interval for different materials

In simple terms, the excitation frequency applied by the cutting tool on the workpiece corresponds to the frequency at which the tool's teeth hit the material, i.e. the *tooth passing frequency* f_{TPF}, which is found by multiplying the rotational frequency f_{rot} by the number of teeth Z on the tool:

$$f_{TPF} = Z \cdot f_{rot} = Z \cdot \frac{n}{60} \ , \tag{92}$$

where f_{TPF} and f_{rot} are expressed in [Hz] and n in [1/min].

To obtain an estimate of the spindle rotation speed, the first step is to determine the cutting speed v_c, typically expressed in [m/min]. This speed is primarily dependent on the material being cut and can be found in catalogs, such as (Horn Group 2023) for example. Some materials, like aluminum, for example, require higher cutting speeds than others, such as titanium. Once the cutting speed is known, the rotation speed is found through the following expression:

$$n = \frac{1000 \cdot v_c}{D \cdot \pi} \ , \tag{93}$$

where D is the tool's diameter in [mm]. The rotation speed n from Eq. 93 and the number of teeth Z lead to the excitation frequency given in Eq. 92.

To estimate the range of excitation frequencies exerted by the cutting tool on the workpiece, the following assumptions are made: the tool diameter ranges from 10 [mm] to 100 [mm], and the number of teeth does not exceed 8. Using typical cutting speeds for various workpiece materials (as provided in manufacturer catalogs), along with the established relationships between cutting speed, tool geometry, and rotational speed, the tooth passing frequency (i.e. the excitation frequency) can be calculated. While this approach does not yield exact values, it provides a reasonable approximation of the expected excitation frequency intervals. Figure 42 presents the estimated excitation frequency intervals for various workpiece materials.

As shown in Figure 42, the excitation frequency is significantly higher for aluminum alloys, with a maximum reaching around 2000 Hz. In contrast, the excitation frequencies for other materials are much lower. This is attributed to the fact that aluminum alloys require higher cutting speeds ($v_c \approx 450$ [m/min]). For materials such as titanium and steels, the excitation frequencies generally do not exceed 1000 Hz. It is important to emphasize that these values represent approximate excitation frequencies during milling. The exact frequencies are influenced by numerous factors, including tool coating, depth of cut, and other machining parameters. However, the material being machined remains the primary factor influencing cutting speed,

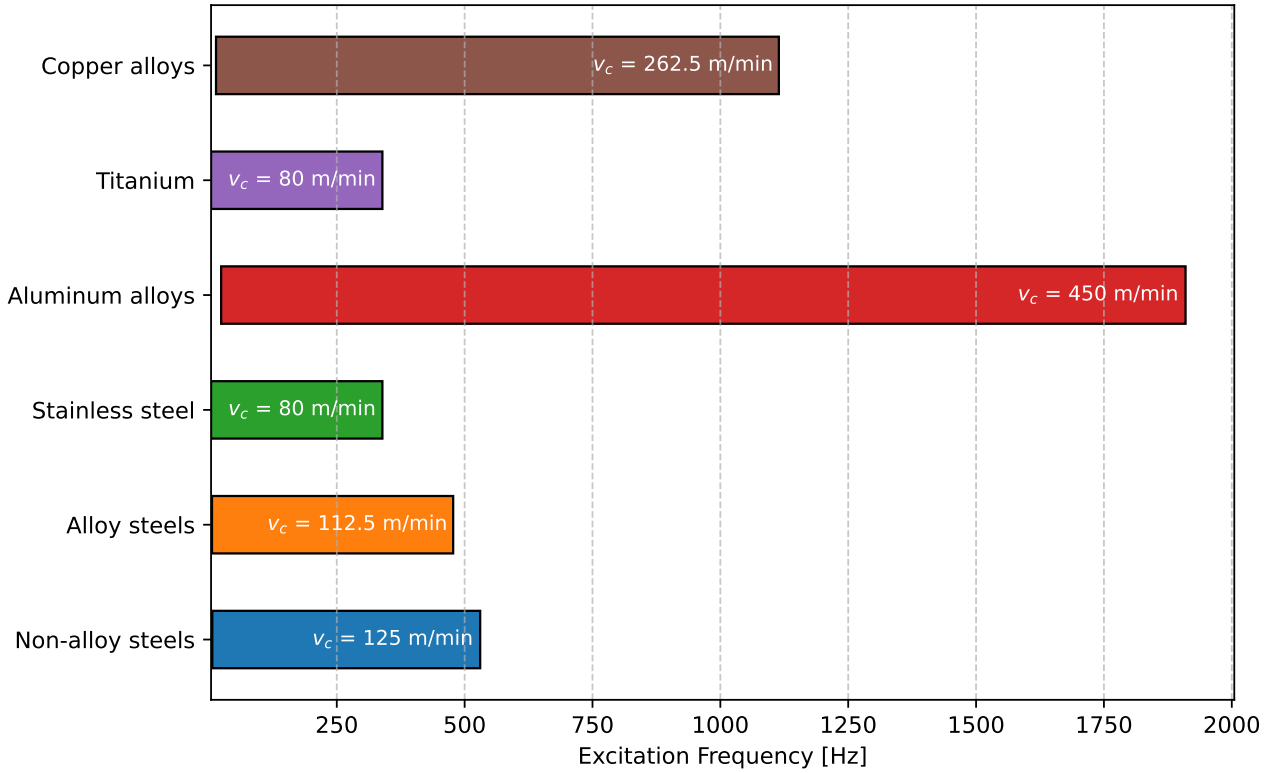


Figure 42: Estimation of the excitation frequency intervals for different materials (10 [mm] $\leq D \leq$ 100 [mm], $Z \leq 8$). Cutting speeds obtained from (Horn Group 2023).

which provides a reasonable estimate for the excitation frequency range (Hoffmann Group 2016).

Force magnitudes

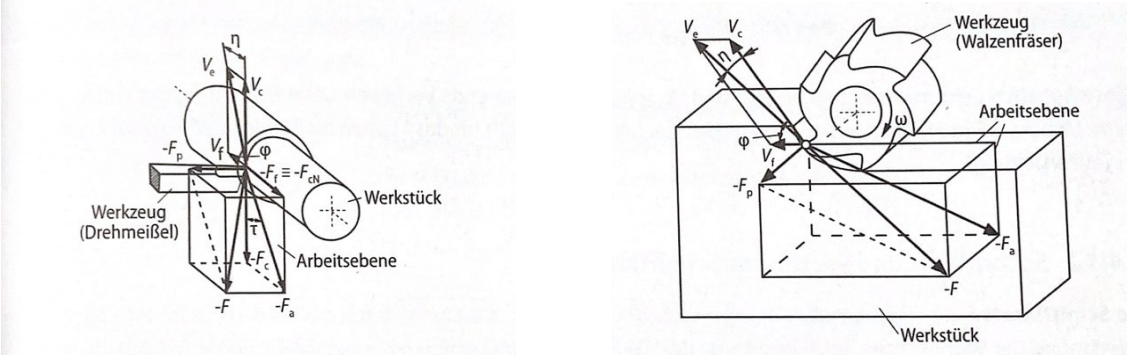
To obtain theoretical values of the forces acting during milling and turning operations, we will consider a simplified model described in (Hoffmann Group 2016). The *machining force* F is defined as the global force that acts on the workpiece. This force can be decomposed into 2 components, a *passive force* F_p, that is oriented along the axial direction of the cutting tool, and an *active force* F_a, which lays in the cutting plane orthogonal to F_p. The force F_a can be further decomposed into a *cutting force* F_c, which is the component that is used for the power calculation as it is responsible for the actual removal of the chip, and a *feed force* F_f. These forces are illustrated in Figure 43.

Expression of the cutting force F_c

The cutting force F_c depends on the material of the workpiece, the depth of cut and the geometry of the cutting tool. For milling and turning operations, it is expressed as follows (in [N]):

$$F_c = A \cdot k_c = b \cdot h \cdot k_c \ , \tag{94}$$

where A is the chip cross-section in [mm^2], b is the chip width in [mm], h is the chip thickness in [mm] and k_c is the specific cutting force in [N/mm^2]. The latter depends on the material



(a) Turning. (b) Milling.

Figure 43: Forces involved during turning and milling operations (Hoffmann Group 2016). *Werkzeug (en: tool), Arbeitsebene (en: working plane), Werkstück (en: workpiece).*

strength of the workpiece and the depth of cut:

$$k_c = \frac{k_{c1.1}}{h^m} \ . \tag{95}$$

The values for the principal specific cutting force $k_{c1.1}$ (in [N/mm^2]) and the parameter m can be obtained for various materials from tables as given in (Hoffmann Group 2016). Correction factors are generally applied to the cutting force to take into account the rake angle, the tool wear condition, etc. The expression for the force then becomes

$$F_c = b \cdot h \cdot k_c \cdot K_\gamma \cdot K_v \cdot K_{sch} \cdot K_{ver}. \tag{96}$$

In Eq. 96, K_γ accounts for a change in the cutting angle, K_v accounts for a change in the cutting speed, K_{sch} accounts for the material of the tool and K_{ver} accounts for the wear condition of the tool. Note also that the chip thickness is generally replaced by an *average chip thickness* h_m to take into account changes in thickness during milling (Hoffmann Group 2016).

Expression of the feed and passive forces

Similarly to the calculation of the cutting force, the feed and passive forces are expressed as follows (in [N]):

$$F_f = b \cdot k_{f1.1} \cdot h^{1-m_f} \qquad \text{and} \qquad F_p = b \cdot k_{p1.1} \cdot h^{1-m_p} \ , \tag{97}$$

where $k_{f1.1}$ and $k_{p1.1}$ are specific forces depending on the material and can be obtained from tables in (Hoffmann Group 2016).

Interested readers may refer to (Hoffmann Group 2016) for a more detailed explanation of the calculation of b and h_m.

Cutting power

Once the cutting force F_c is known, the cutting power P_c can be computing as follows (in [kW]):

$$P_c = \frac{F_c \cdot v_c}{60000} \ , \tag{98}$$

where F_c and v_c are expressed in [N] and [m/min] respectively. The driving power P_a depends on the efficiency η of the machining process:

$$P_a = \frac{P_c}{\eta} \ . \tag{99}$$

Appendix B - CONLIN Method

This section aims to describe the CONLIN method, which is a particular case of MMA. The subproblem formulation as well as its resolution in the dual space has been discussed below.

CONLIN subproblem formulation

The CONLIN method, which stands for *Convex Linearization method*, is a method which transforms a general non-linear optimization problem into a series of convex sub-problems that are both explicit and separable. Consider the following problem:

$$\begin{aligned} &\min_{\mathbf{x}} c_0(\mathbf{x}) \\ &\text{s.t.} \quad c_j(\mathbf{x}) \leq 0 \qquad (j = 1, ..., m) \\ &\qquad \underline{x}_i \leq x_i \leq \bar{x}_i \quad (i = 1, ..., n). \end{aligned} \tag{100}$$

The convex linearization of the objective function c_0 and the constraint functions c_j at $\mathbf{x}_0$ is given by:

$$c(\mathbf{x}) = c(\mathbf{x}_0) + \sum_{+} c_i^0 (x_i - x_i^0) - \sum_{-} (x_i^0)^2 c_i^0 \big(\frac{1}{x_i} - \frac{1}{x_i^0}\big), \tag{101}$$

where $\sum_+$ and $\sum_-$ denote the summation of the terms for which c_i^0 is positive or negative respectively, and where

$$c_i^0 = \left.\frac{\partial c}{\partial x_i}\right|_{x_i^0}. \tag{102}$$

Eq. 101 can be rewritten as

$$c(\tilde{\mathbf{x}}) = c(\mathbf{x}_0) + \sum_{+} \tilde{c}_i (\tilde{x}_i - 1) - \sum_{-} \tilde{c}_i \big(\frac{1}{\tilde{x}_i} - 1\big), \tag{103}$$

if the the following normalization of the design variables is used:

$$\tilde{x}_i = \frac{x_i}{x_i^0} \qquad \Longrightarrow \qquad \tilde{c}_i = c_i^0 x_i^0. \tag{104}$$

The associated convex subproblems are obtained by replacing the normalized expression of convex linearized function from Eq. 103 in the optimization problem given from Eq. 100:

$$\begin{aligned} &\min_{\mathbf{x}} \sum_{+} c_{i0} x_i - \sum_{-} \frac{c_{i0}}{x_i} - \bar{c}_0 \\ &\text{s.t.} \ \sum_{+} c_{ij} x_i - \sum_{-} \frac{c_{ij}}{x_i} \leq \bar{c}_j \quad (j = 1, ..., m) \\ &\qquad \underline{x}_i \leq x_i \leq \bar{x}_i \qquad\qquad (i = 1, ..., n), \end{aligned} \tag{105}$$

where the s̃ymbol has been dropped, c_{ij} denotes the normalized first order derivative at $\mathbf{x}_0$ and

$$\bar{c}_j = \sum_i |c_{ij}| x_i^0 - c_j(\mathbf{x}_0) \qquad (j = 0, ..., m). \tag{106}$$

The subproblem in Eq. 105 is explicit, convex and separable, meaning that it is well suited for dual algorithms (Fleury 1989) (Duysinx 2020).

Dual method for CONLIN subproblems

This section describes the use of dual optimizer on subproblems that are generated using the CONLIN method described above. The development have been taken from (Fleury 1989) and (Duysinx 2020).

To solve the optimization problem in Eq. 105, the dual approach consists in considering the corresponding dual problem given by:

$$\max_{\boldsymbol{\lambda} \geq 0} l(\boldsymbol{\lambda}), \tag{107}$$

where $l(\boldsymbol{\lambda})$ is called *dual function* and is obtained by minimizing the Lagrangian function associated to the initial optimization problem:

$$\min_{\underline{x}_i \leq x_i \leq \bar{x}_i} L(\mathbf{x}, \boldsymbol{\lambda}) = \sum_{j=0}^{m} \lambda_j \left(\sum_{+} c_{ij} x_i - \sum_{-} \frac{c_{ij}}{x_i} - \bar{c}_j \right). \tag{108}$$

The function that gives the primal design variables that minimize the Lagrangrian function for a given $\boldsymbol{\lambda}$ is referenced as the *primmal-dual relationship* and denoted by $\mathbf{x} = \mathbf{x}(\boldsymbol{\lambda})$. In the present case, because the functions c_j are separable, the minimization of the Lagrangian in Eq. 108 can be decomposed into n 1D independent minimization problems:

$$\min_{\underline{x}_i \leq x_i \leq \bar{x}_i} L_i(x_i) = a_i x_i + \frac{b_i}{x_i}, \tag{109}$$

where $a_i = \sum_+ c_{ij} \lambda_j \geq 0$ and $b_i = -\sum_- c_{ij} \lambda_j \geq 0$. By differentiating Eq. 109 and setting the derivative equal to zero, one finds:

$$\begin{aligned} x_i &= \sqrt{\frac{b_i}{a_i}} \quad \text{if} \quad \underline{x}_i^2 \leq \frac{b_i}{a_i} \leq \bar{x}_i^2, \\ x_i &= \bar{x}_i \quad \text{if} \quad \frac{b_i}{a_i} \leq \underline{x}_i^2 \\ x_i &= \bar{x}_i \quad \text{if} \quad \bar{x}_i^2 \leq \frac{b_i}{a_i}. \end{aligned} \tag{110}$$

Hence, the dual problem in Eq. 107 can be expressed as follows:

$$\max_{\boldsymbol{\lambda} \geq 0} l(\boldsymbol{\lambda}) = \sum_{j=0}^{m} \lambda_j \left(\sum_{+} c_{ij} x_i(\boldsymbol{\lambda}) - \sum_{-} \frac{c_{ij}}{x_i(\boldsymbol{\lambda})} - \bar{c}_j \right). \tag{111}$$

It can be shown that the first- and second-order derivates of the dual function are respectively given by:

$$g_j = \frac{\partial l}{\partial \lambda_j} = \sum_{+} c_{ij} x_i(\boldsymbol{\lambda}) - \sum_{-} \frac{c_{ij}}{x_i(\boldsymbol{\lambda})} - \bar{c}_j \tag{112}$$

and

$$H_{ij} = \frac{\partial^2 l}{\partial \lambda_j \partial \lambda_k} = -\frac{1}{2} \sum_{i \in I} n_{ij} n_{ik} \frac{x_i}{a_i}, \tag{113}$$

for which $I = \{i \mid \underline{x}_i < x_i < \bar{x}_i\}$ and where $n_{ij} = c_{ij}$ if $c_{ij} > 0$ and $n_{ij} = c_{ij}/x_i^2$ if $c_{ij} < 0$. Note that the summation in Eq. 113 only applied to so-called *free primal variable* x_i (as opposed to *fixed primal variables*) which have not reached either one of their upper bounds $\underline{x}_i$ or $\bar{x}_i$. Whenever a free primal variable becomes fixed, or conversely, certain terms in the expression of the dual Hessian matrix will disappear, leading to a discontinuous behavior. This is a major difficulty when solving dual problems. Fleury (1989) has therefore proposed to use a sequential quadratic programming (SQP) method to solve the dual problem that is the topic of the following section.

To summarize, the initial minimization problem (Eq. 100) is decomposed into a series of simpler subproblems (Eq. 105) that are explicit, convex and separable. Each of these CONLIN subproblems are solved using a dual formulation (Eq. 111), which is itself solved via SQP.

Appendix C - Python Code

This appendix includes the Python optimization code. To conserve space and maintain a compact layout, the code is presented in a smaller font size.

File *inputs.py*

```
"""
INPUTS
------

Description: This code processes the input files.

Author: J. Adler

Academic year: 2024-2025
"""


import numpy as np

#Extract node list
nodeList = np.loadtxt('Inputs/nodeList.txt')/1000 #[m]

nodes, dofPerNode = nodeList.shape

#Extract workpiece element list
elemListWorkpiece = np.loadtxt('Inputs/elemListWorkpiece.txt', dtype=int)

workpieceElem = len(elemListWorkpiece)

#Extract bars element list
elemListBars = np.loadtxt('Inputs/elemListBars.txt', dtype=int)

barElem = len(elemListBars)

#Define dof list
dofList = np.zeros((nodes, dofPerNode))

for i in range(nodes):
    for j in range(dofPerNode):
        dofList[i,j] = 1 + j + 3*i

#Define locel matrices
locelWorkpiece = np.zeros((workpieceElem, 4*dofPerNode))

for i in range(workpieceElem):
    for j in range(4):

        J = dofPerNode*j
        JJ = dofPerNode*(j+1)

        locelWorkpiece[i,J:JJ] = dofList[elemListWorkpiece[i][j]-1]

locelBars = np.zeros((barElem, 2*dofPerNode))

for i in range(barElem):
    for j in range(2):

        J = dofPerNode*j
        JJ = dofPerNode*(j+1)

        locelBars[i,J:JJ] = dofList[elemListBars[i][j]-1]

#Extract fixed nodes list
fixedList = np.loadtxt('Inputs/fixedNodes.txt')

#Extract load list
loadList = np.loadtxt('Inputs/machiningLoads.txt')
if loadList.ndim==1:
    loadList = np.array([loadList])

#Define the total number of dofs
dofs = int(dofList.max())
```

File *elementary.py*

```
"""
ELEMENTARY
---------

Description: This code contains the elementary stiffness and mass matrices.

Author: J. Adler

Academic year: 2024-2025
"""


import numpy as np
import inputs

#Define D matrix relating the strains and the stress for an isotropic material
def D(E, nu):
    return np.array([[1-nu, nu, nu, 0, 0, 0],
                     [nu, 1-nu, nu, 0, 0, 0],
                     [nu, nu, 1-nu, 0, 0, 0],
                     [0, 0, 0, 0.5-nu, 0, 0],
                     [0, 0, 0, 0, 0.5-nu, 0],
                     [0, 0, 0, 0, 0, 0.5-nu]]) * E/((1+nu)*(1-2*nu))

#Define elementary stiffness matrix of T4 element
def Ke_T4(elem, E, nu):

    e = int(elem-1)
    nodesInElem = inputs.elemListWorkpiece[e]
    n1, n2, n3, n4 = nodesInElem[:] #Nodes in the tetrahedral element

    n1 = int(n1-1)
    n2 = int(n2-1)
    n3 = int(n3-1)
    n4 = int(n4-1) #Indices of the nodes

    x1, y1, z1 = inputs.nodeList[n1]
    x2, y2, z2 = inputs.nodeList[n2]
    x3, y3, z3 = inputs.nodeList[n3]
    x4, y4, z4 = inputs.nodeList[n4]

    x14, x24, x34 = x1-x4, x2-x4, x3-x4
    y14, y24, y34 = y1-y4, y2-y4, y3-y4
    z14, z24, z34 = z1-z4, z2-z4, z3-z4

    J = np.array([[x14, y14, z14],
                  [x24, y24, z24],
                  [x34, y34, z34]]) #Jacobian

    #Determinant of Jacobian
    det_J = x14 * (y24*z34 - y34*z24) + y14 * (z24*x34 - z34*x24) + z14 * (x24*y34 - x34*y24)

    A = np.array([[y24*z34 - y34*z24, y34*z14 - y14*z34, y14*z24 - y24*z14],
                  [z24*x34 - z34*x24, z34*x14 - z14*x34, z14*x24 - z24*x14],
                  [x24*y34 - x34*y24, x34*y14 - x14*y34, x14*y24 - x24*y14]]) / det_J #Inverse of Jacobian

    A1 = A[0].sum()
    A2 = A[1].sum()
    A3 = A[2].sum()

    #B matrix relating nodal displacement to strains
    B = np.array([[A[0,0], 0, 0, A[0,1], 0, 0, A[0,2], 0, 0, -A1, 0, 0],
                  [0, A[1,0], 0, 0, A[1,1], 0, 0, A[1,2], 0, 0, -A2, 0],
                  [0, 0, A[2,0], 0, 0, A[2,1], 0, 0, A[2,2], 0, 0, -A3],
                  [0, A[2,0], A[1,0], 0, A[2,1], A[1,1], 0, A[2,2], A[1,2], 0, -A3, -A2],
                  [A[2,0], 0, A[0,0], A[2,1], 0, A[0,1], A[2,2], 0, A[0,2], -A3, 0, -A1],
                  [A[1,0], A[0,0], 0, A[1,1], A[0,1], 0, A[1,2], A[0,2], 0, -A2, -A1, 0]])


    ke = abs(det_J)/6 * (B.T @ D(E,nu) @ B) #Elementary stiffness matrix

    return ke

#Define elementary stiffness matrix of bar element
def Ke_bar(elem, E, A):

    e = int(elem-1)
    nodesInElem = inputs.elemListBars[e]
    n1, n2 = nodesInElem[:] #Nodes in the bar element

    n1 = int(n1-1)
    n2 = int(n2-1) #Indices of the nodes

    x1, y1, z1 = inputs.nodeList[n1]
    x2, y2, z2 = inputs.nodeList[n2]

    dX = x2 - x1
    dY = y2 - y1
    dZ = z2 - z1

    L = np.sqrt((x2 - x1)**2 + (y2 - y1)**2 + (z2 - z1)**2)
```

```
    return np.array([[dX**2, dX*dY, dX*dZ, -dX**2, -dX*dY, -dX*dZ],
                     [dX*dY, dY**2, dY*dZ, -dX*dY, -dY**2, -dY*dZ],
                     [dX*dZ, dY*dZ, dZ**2, -dX*dZ, -dY*dZ, -dZ**2],
                     [-dX**2, -dX*dY, -dX*dZ, dX**2, dX*dY, dX*dZ],
                     [-dX*dY, -dY**2, -dY*dZ, dX*dY, dY**2, dY*dZ],
                     [-dX*dZ, -dY*dZ, -dZ**2, dX*dZ, dY*dZ, dZ**2]]) * (E*A / L**3)

#Define elementary mass matrix of bar elements
def Me_bar(elem, A, rho):

    e = int(elem-1)
    nodesInElem = inputs.elemListBars[e]
    n1, n2 = nodesInElem[:]

    n1 = int(n1-1)
    n2 = int(n2-1)

    x1, y1, z1 = inputs.nodeList[n1]
    x2, y2, z2 = inputs.nodeList[n2]

    L = np.sqrt((x2 - x1)**2 + (y2 - y1)**2 + (z2 - z1)**2)

    return np.array([[2, 0, 0, 1, 0, 0],
                     [0, 2, 0, 0, 1, 0],
                     [0, 0, 2, 0, 0, 1],
                     [1, 0, 0, 2, 0 ,0],
                     [0, 1, 0, 0, 2, 0],
                     [0, 0, 1, 0, 0, 2]]) * (rho*A*L/6)

#Define elementary mass matrix of T4 elements
def Me_T4(elem, rho):

    e = int(elem-1)
    nodesInElem = inputs.elemListWorkpiece[e]
    n1, n2, n3, n4 = nodesInElem[:] #Nodes in the tetrahedral element

    n1 = int(n1-1)
    n2 = int(n2-1)
    n3 = int(n3-1)
    n4 = int(n4-1) #Indices of the nodes

    x1, y1, z1 = inputs.nodeList[n1]
    x2, y2, z2 = inputs.nodeList[n2]
    x3, y3, z3 = inputs.nodeList[n3]
    x4, y4, z4 = inputs.nodeList[n4]

    x14, x24, x34 = x1-x4, x2-x4, x3-x4
    y14, y24, y34 = y1-y4, y2-y4, y3-y4
    z14, z24, z34 = z1-z4, z2-z4, z3-z4

    #Determinant of Jacobian
    det_J = x14 * (y24*z34 - y34*z24) + y14 * (z24*x34 - z34*x24) + z14 * (x24*y34 - x34*y24)
    Ve = np.abs(det_J)/6

    return np.array([[2, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0],
                     [0, 2, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0],
                     [0, 0, 2, 0, 0, 1, 0, 0, 1, 0, 0, 1],
                     [1, 0, 0, 2, 0, 0, 1, 0, 0, 1, 0, 0],
                     [0, 1, 0, 0, 2, 0, 0, 1, 0, 0, 1, 0],
                     [0, 0, 1, 0, 0, 2, 0, 0, 1, 0, 0, 1],
                     [1, 0, 0, 1, 0, 0, 2, 0, 0, 1, 0, 0],
                     [0, 1, 0, 0, 1, 0, 0, 2, 0, 0, 1, 0],
                     [0, 0, 1, 0, 0, 1, 0, 0, 2, 0, 0, 1],
                     [1, 0, 0, 1, 0, 0, 1, 0, 0, 2, 0, 0],
                     [0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 2, 0],
                     [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 2]]) * (rho*Ve/20)
```

File *assembly.py*

```
"""
ASSEMBLY

Description: This code assembles the structural stiffness and mass matrices.

Author: J. Adler

Academic year: 2024-2025
"""


import numpy as np
import inputs
import elementary
import display
import reduction

#Define mechanical properties of the workpiece
E_w = 73.119 * 1e9 #[Pa]
nu_w = 0.33 #[-]
rho_w = 2700 #[kg/m^3]

#E_w = 206.94 * 1e9 #[Pa]
#nu_w = 0.288 #[-]
#rho_w = 7829 #[kg/m^3]

#Define mechanical properties of the bars
alpha = 1
#E_b = alpha * E_w
E_b = 206.94 * 1e9 #[Pa]
nu_b = nu_w #[-]
A_b = (10**2 * np.pi / 4) * 1e-6 #[m^2]
rho_b = rho_w #[kg/m^3]

#Penalization parameter
p_SIMP = 3

#Simply isotropic material with penalization
def E(x, p=p_SIMP):
    if p<1 or x<0 or 1<x:
        raise ValueError("Problem in 'E'")
    return E_b * x**p

def rho(x):
    if x<0 or 1<x:
        raise ValueError("Problem in 'rho'")
    return rho_b * x

#Define the structural workpiece stiffness matrix
def K_workpiece():

    K = np.zeros((inputs.dofs, inputs.dofs))

    for e in range(inputs.workpieceElem):
        ke = elementary.Ke_T4(e+1, E_w, nu_w) #Elementary stiffness matrix of tetrahedral elements

        loc = inputs.locelWorkpiece[e] - 1 #Indices of the dofs

        for i in range(4 * inputs.dofPerNode):
            for j in range(4 * inputs.dofPerNode):
                ii, jj = int(loc[i]), int(loc[j])
                K[ii, jj] += ke[i][j]

    return K

#Define the structural workpiece mass matrix
def M_workpiece():

    M = np.zeros((inputs.dofs, inputs.dofs))

    for e in range(inputs.workpieceElem):
        me = elementary.Me_T4(e+1, rho_w) #Elementary stiffness matrix of tetrahedral elements

        loc = inputs.locelWorkpiece[e] - 1 #Indices of the dofs

        for i in range(4 * inputs.dofPerNode):
            for j in range(4 * inputs.dofPerNode):
                ii, jj = int(loc[i]), int(loc[j])
                M[ii, jj] += me[i][j]

    return M

#Define the structural bars stiffness matrix
def K_bars(x):

    K = np.zeros((inputs.dofs, inputs.dofs))

    for e in range(inputs.barElem):
        ke = elementary.Ke_bar(e+1, E(x[e]), A_b) #Elementary stiffness matrix of bar elements

        loc = inputs.locelBars[e] - 1 #Indices of the dofs
```

```
            for i in range(2 * inputs.dofPerNode):
                for j in range(2 * inputs.dofPerNode):
                    ii, jj = int(loc[i]), int(loc[j])
                    K[ii, jj] += ke[i][j]

    return K

#Define the structural bars mass matrix
def M_bars(x):

    M = np.zeros((inputs.dofs, inputs.dofs))

    for e in range(inputs.barElem):
        me = elementary.Me_bar(e+1, A_b, rho(x[e])) #Elementary stiffness matrix of bar elements

        loc = inputs.locelBars[e] - 1 #Indices of the dofs

        for i in range(2 * inputs.dofPerNode):
            for j in range(2 * inputs.dofPerNode):
                ii, jj = int(loc[i]), int(loc[j])
                M[ii, jj] += me[i][j]

    return M

#Define the force vector
def f():

    F = np.zeros(inputs.dofs)

    for force in inputs.loadList:
        node_index = int(force[0]-1)
        i,j,k = inputs.dofList[node_index,:]

        i = int(i-1)
        j = int(j-1)
        k = int(k-1) #Indices of the dofs

        Fx, Fy, Fz = force[1:] #Components of the nodal force

        F[i] += Fx
        F[j] += Fy
        F[k] += Fz

    return F

#Matrix of force vectors for machining operation
def f_individual():

    F = np.zeros((inputs.dofs, len(inputs.loadList)))

    for i in range(len(inputs.loadList)):
        force = inputs.loadList[i]
        node_index = int(force[0]-1)
        I,J,K = inputs.dofList[node_index,:]

        I = int(I-1)
        J = int(J-1)
        K = int(K-1) #Indices of the dofs

        Fx, Fy, Fz = force[1:] #Components of the nodal force

        F[np.array([I,J,K]), i] = np.array([Fx, Fy, Fz])

    return F


#Apply the fixed nodes boundary conditions
def applyBC(K, dim=2):
    dofs_index = np.array([], dtype=int)

    for node in inputs.fixedList:
        node_index = int(node-1)
        dofs_index = np.concatenate((dofs_index, np.arange(inputs.dofPerNode * node_index,
        inputs.dofPerNode * (node_index+1), 1, dtype=int)), dtype=int)

    dofs_index = np.sort(dofs_index)

    row_mask = np.ones(len(K), dtype=bool)
    col_mask = np.ones(len(K), dtype=bool)

    row_mask[dofs_index] = False
    col_mask[dofs_index] = False

    if dim==2: #For K and M
        K = K[row_mask, :]
        K = K[:, col_mask]

        return K

    if dim==1: #For f
        K = K[row_mask]

        return K
```

```
    raise ValueError("Problem in 'applyBC'")

#Get full displacement vector (including fixed nodes)
def q_full(q):
    dofs_index = np.array([], dtype=int)

    for node in inputs.fixedList:
        node_index = int(node-1)
        dofs_index = np.concatenate((dofs_index, np.arange(inputs.dofPerNode * node_index,
        inputs.dofPerNode * (node_index+1), 1, dtype=int)), dtype=int)

    dofs_index = np.sort(dofs_index)

    Q = np.zeros(inputs.dofs)

    I = 0
    for i in range(inputs.dofs):
        if i in dofs_index:
            I += 1
            continue
        Q[i] = q[i-I]

    return Q

#Reduced stiffness matrix for the bars
def K_bars_tilde(x):
    #List of active bar nodes
    elemListBars = inputs.elemListBars.flatten()
    fixedList = inputs.fixedList
    active_nodes = np.sort([item for item in elemListBars if item not in fixedList])

    #Initialize the matrix
    n = inputs.dofPerNode * inputs.barElem
    K_b_tilde = np.zeros((n, n))

    #Loop over the bars
    for i in range(len(inputs.elemListBars)):
        n1, n2 = inputs.elemListBars[i][:]

        K_b_e = elementary.Ke_bar(i+1, E(x[i]), A_b)

        if n1 in active_nodes and n2 in active_nodes:
            raise ValueError("Problem in 'K_bars_tilde")

        elif n1 in active_nodes:
            I = np.where(active_nodes == n1)[0][0]
            J = I+1

            K_b_tilde[3*I:3*J, 3*I:3*J] += K_b_e[:3,:3]

        elif n2 in active_nodes:
            I = np.where(active_nodes == n2)[0][0]
            J = I+1

            K_b_tilde[3*I:3*J, 3*I:3*J] += K_b_e[3:,3:]

        else:
            raise ValueError("Problem in 'K_bars_tilde")

    return K_b_tilde

#Reduced mass matrix of the bars
def M_bars_tilde(x):
    #List of active bar nodes
    elemListBars = inputs.elemListBars.flatten()
    fixedList = inputs.fixedList
    active_nodes = np.sort([item for item in elemListBars if item not in fixedList])

    #Initialize the matrix
    n = inputs.dofPerNode * inputs.barElem
    M_b_tilde = np.zeros((n, n))

    #Loop over the bars
    for i in range(len(inputs.elemListBars)):
        n1, n2 = inputs.elemListBars[i][:]

        M_b_e = elementary.Me_bar(i+1, A_b, rho(x[i]))

        if n1 in active_nodes and n2 in active_nodes:
            raise ValueError("Problem in 'M_bars_tilde")

        elif n1 in active_nodes:
            I = np.where(active_nodes == n1)[0][0]
            J = I+1

            M_b_tilde[3*I:3*J, 3*I:3*J] += M_b_e[:3,:3]

        elif n2 in active_nodes:
            I = np.where(active_nodes == n2)[0][0]
            J = I+1

            M_b_tilde[3*I:3*J, 3*I:3*J] += M_b_e[3:,3:]

        else:
            raise ValueError("Problem in 'M_bars_tilde")
```

```
    return M_b_tilde

#Reduced stiffness matrix of the workpiece
def K_workpiece_tilde():
    #Structural stiffness matrix
    K_w = applyBC(K_workpiece())

    #Reduction
    retained_dofs = reduction.get_retained_dofs()
    retained_indices = reduction.get_retained_indices(retained_dofs)
    R = reduction.R_gi(K_w, retained_indices)

    K_w_tilde = reduction.K_gi(K_w, R)

    return K_w_tilde, R

#Reduced mass matrix of the workpiece
def M_workpiece_tilde(R):
    #Structural mass matrix
    M_w = applyBC(M_workpiece())

    return reduction.M_gi(M_w, R)


#Reduced force vector
def f_tilde(R):
    return R.T @ applyBC(f(), dim=1)

#Reduced force vector for multiple load cases
def f_individual_tilde(R):
    return R.T @ applyBC(f_individual(), dim=1)

display.plot(inputs.nodeList,
             inputs.elemListWorkpiece,
             inputs.elemListBars,
             inputs.fixedList,
             inputs.loadList,
             force_scale=5e-3)
```

File *sensitivity.py*

```
"""
SENSITIVITY

Description: This code is aimed at computing the different sensitivity values of the optimization problem.

Author: J. Adler

Academic year: 2024-2025
"""

import numpy as np
import inputs
import elementary
import assembly

#Define a function that returns the sensitivity of the elementary stiffness matrix
def dKe_dx(i, x):
    if i<1 or i>inputs.barElem:
        raise ValueError("Invalid index")

    #Parameters
    p = assembly.p_SIMP
    E0 = assembly.E_b
    A = assembly.A_b

    #Elementary stiffness matrix for x=1
    Ke = elementary.Ke_bar(i, E0, A)

    #Sensitivity of the elementary stiffness matrix
    dKedx = p * np.power(x[i-1], p-1) * Ke

    return dKedx

#Define a function that returns the sensitivity of the structural stiffness matrix
def dK_dx(i, x):
    if i<1 or i>inputs.barElem:
        raise ValueError("Invalid index")

    #Sensitivity of the elementary stiffness matrix
    dKedx = dKe_dx(i, x)

    #Initialize stiffness sensitivity
    dKdx = np.zeros((inputs.dofs, inputs.dofs))

    #Vector containing the dofs of the i-th bar
    loc = inputs.locelBars[i-1] - 1

    for I in range(2 * inputs.dofPerNode):
        for J in range(2 * inputs.dofPerNode):
            ii, jj = int(loc[I]), int(loc[J])
            dKdx[ii, jj] += dKedx[I][J]

    #Apply the boundary conditions
    dKdx = assembly.applyBC(dKdx)

    return dKdx

#Define a function that returns the sensitivity of the elementary mass matrix
def dMe_dx(i):
    if i<1 or i>inputs.barElem:
        raise ValueError("Invalid index")

    #Parameters
    A = assembly.A_b
    rho = assembly.rho_b

    #Elementary mass matrix for x=1
    Me = elementary.Me_bar(i, A, rho)

    #Sensitivity of the elementary mass matrix
    dMedx = Me

    return dMedx

#Define a function that returns the sensitivity of the structural mass matrix
def dM_dx(i):
    if i<1 or i>inputs.barElem:
        raise ValueError("Invalid index")

    #Sensitivity of the elementary mass matrix
    dMedx = dMe_dx(i)

    #Initialize mass sensitivity
    dMdx = np.zeros((inputs.dofs, inputs.dofs))

    #Vector containing the dofs of the i-th bar
    loc = inputs.locelBars[i-1] - 1

    for I in range(2 * inputs.dofPerNode):
        for J in range(2 * inputs.dofPerNode):
            ii, jj = int(loc[I]), int(loc[J])
```

```
                dMdx[ii, jj] += dMedx[I][J]

        #Apply the boundary conditions
        dMdx = assembly.applyBC(dMdx)

        return dMdx

#Sensitivity of reduced stiffness matrix
def dKtilde_dx(i, x):
        if i<1 or i>inputs.barElem:
                raise ValueError("Invalid index")

        #List of active bar nodes
        elemListBars = inputs.elemListBars.flatten()
        fixedList = inputs.fixedList
        active_nodes = np.sort([item for item in elemListBars if item not in fixedList])

        #Sensitivity of the elementary stiffness matrix
        dKedx = dKe_dx(i, x)

        #Initialize stiffness sensitivity
        n = inputs.barElem * inputs.dofPerNode
        dKtildedx = np.zeros((n, n))

        #Fill the matrix
        n1, n2 = inputs.elemListBars[i-1][:]

        if n1 in active_nodes and n2 in active_nodes:
                raise ValueError("Problem in 'dKtilde_dx'")

        elif n1 in active_nodes:
                I = np.where(active_nodes == n1)[0][0]
                J = I+1
                dKtildedx[3*I:3*J, 3*I:3*J] += dKedx[:3,:3]

        elif n2 in active_nodes:
                I = np.where(active_nodes == n2)[0][0]
                J = I+1
                dKtildedx[3*I:3*J, 3*I:3*J] += dKedx[3:,3:]

        else:
                raise ValueError("Problem in 'dKtilde_dx'")

        return dKtildedx

#Sensitivity of reduced mass matrix
def dMtilde_dx(i):
        if i<1 or i>inputs.barElem:
                raise ValueError("Invalid index")

        #List of active bar nodes
        elemListBars = inputs.elemListBars.flatten()
        fixedList = inputs.fixedList
        active_nodes = np.sort([item for item in elemListBars if item not in fixedList])

        #Sensitivity of the elementary mass matrix
        dMedx = dMe_dx(i)

        #Initialize mass sensitivity
        n = inputs.barElem * inputs.dofPerNode
        dMtildedx = np.zeros((n, n))

        #Fill the matrix
        n1, n2 = inputs.elemListBars[i-1][:]

        if n1 in active_nodes and n2 in active_nodes:
                raise ValueError("Problem in 'dMtilde_dx'")

        elif n1 in active_nodes:
                I = np.where(active_nodes == n1)[0][0]
                J = I+1
                dMtildedx[3*I:3*J, 3*I:3*J] += dMedx[:3,:3]

        elif n2 in active_nodes:
                I = np.where(active_nodes == n2)[0][0]
                J = I+1
                dMtildedx[3*I:3*J, 3*I:3*J] += dMedx[3:,3:]

        else:
                raise ValueError("Problem in 'dMtilde_dx'")

        return dMtildedx

#Define a function that returns the vector dc0_dq, i.e. the derivative of the objective function with respect to q_k
def dc0_dq(q):
        # Find the largest value in the matrix
        max_value = np.max(q)

        # Create a new matrix with the same shape as the original, filled with zeros
        dc0dq = np.zeros_like(q)

        dc0dq[q == max_value] = 2 * max_value

        return dc0dq
```

```
#Define a function that returns the sensitivity of the objective function
def dc0_dx(x, K, q):

    if q.ndim==1:
        q = q.reshape(-1, 1)

    bars = inputs.barElem

    #Initialization
    dc0dx = np.zeros(bars)

    b = dc0_dq(q)
    non_zero_columns = np.any(b != 0, axis=0)
    column_indices = np.where(non_zero_columns)[0]

    #Iterate
    for i in range(1,bars+1,1):
        dKdx = dK_dx(i, x)

        for s in column_indices:
            lamda = np.linalg.solve(K, b[:,s]).flatten()
            dc0dx[i-1] += -lamda @ dKdx @ q[:,s]

    return dc0dx

#Define derivative of the objective function with respect to the design variable
#in the reduced case
def dc0_dx_tilde(x, R, q, q_tilde, K_tilde):

    if q.ndim==1:
        q = q.reshape(-1,1)
        q_tilde = q_tilde.reshape(-1,1)

    bars = inputs.barElem
    dc0dx = np.zeros(bars)

    b = dc0_dq(q)
    non_zero_columns = np.any(b != 0, axis=0)
    column_indices = np.where(non_zero_columns)[0]

    for i in range(bars):
        dKtildedx = dKtilde_dx(i+1, x)

        for s in column_indices:
            lamda = np.linalg.solve(K_tilde, R.T @ b[:,s]).flatten()
            dc0dx[i] += - lamda @ dKtildedx @ q_tilde[:,s]

    return dc0dx

#Define the sensitivity of the first constraint function
def dc1_dx():
    return np.ones(inputs.barElem)

#Define the sensitivity of the second constraint function
def dc2_dx(x, w1, q1, reduced=False):
    #w1 is the 1st eigenfrequency squared !!
    #q1 is the 1st mode shape, which must be mass normalized (q1.T @ M @ q1 = 1) !!

    #Initialization
    dw1dx = np.zeros(inputs.barElem)

    if not reduced:
        #Iterate
        for i in range(inputs.barElem):
            dKdx = dK_dx(i+1, x)
            dMdx = dM_dx(i+1)
            #Derivative of w_1^2, where w_1 is the 1st eigenfrequency
            dw1dx[i] = q1 @ (dKdx - w1 * dMdx) @ q1

    else:
        #Iterate
        for i in range(inputs.barElem):
            dKtildedx = dKtilde_dx(i+1, x)
            dMtildedx = dMtilde_dx(i+1)
            #Derivative of w_1^2, where w_1 is the 1st eigenfrequency
            dw1dx[i] = q1 @ (dKtildedx - w1 * dMtildedx) @ q1

    return dw1dx
```

File *reduction.py*

```
"""
REDUCTION

Description: This file contains code for Guyan-Irons reduction method.

Author: J. Adler

Academic year: 2024-2025
"""

import numpy as np
import inputs

#Define a function to get the retained and condensed dofs
def get_retained_dofs():

    elemListBars = inputs.elemListBars.flatten()
    fixedList = inputs.fixedList

    retained_nodes = np.sort([item for item in elemListBars if item not in fixedList]).astype(int)

    retained_dofs = np.sort([inputs.dofList[item-1] for item in retained_nodes]).flatten().astype(int)

    return retained_dofs


def get_retained_indices(retained_dofs):
    '''
    Parameters

    retained_dofs : numpy array (n)
        Global DOFs that are retained for the Guyan-Irons reduction.

    Returns

    Returns the indices of the global DOFs in the structural matrices
    (BCs have been imposed).

    '''
    dofs = inputs.dofList.flatten().astype(int)
    fixed_dofs = inputs.dofList[inputs.fixedList.astype(int) - 1].flatten().astype(int)

    active_dofs = np.sort([item for item in dofs if item not in fixed_dofs])

    indices = np.sort([np.where(active_dofs == item)[0][0] for item in retained_dofs])

    return indices

def K_bloc(K, retained):

    # Sort retained DOFs
    x_R = np.sort(retained).astype(int)

    # Find condensed DOFs
    n = K.shape[0]
    x_C = np.setdiff1d(np.arange(n), x_R)

    # Block decomposition
    K_RR = K[np.ix_(x_R, x_R)]
    K_CC = K[np.ix_(x_C, x_C)]
    K_RC = K[np.ix_(x_R, x_C)]
    K_CR = K[np.ix_(x_C, x_R)]

    return K_RR, K_CC, K_RC, K_CR


def R_from_K_bloc(K_CC_inv, K_CR):

    I = np.eye(K_CR.shape[1])
    R_bottom = - K_CC_inv @ K_CR

    return np.vstack([I, R_bottom])

#Define Guyan-Irons reduction matrix
def R_gi(K, retained):
    '''
    Parameters

    K : numpy array (n x n)
        Structural stiffness matrix.
    retained : numpy array (m)
        Retained DOF indices.

    Returns

    R : numpy array (n x m)
        Guyan-Irons reduction matrix.
    '''
    # Sort retained DOFs
    x_R = np.sort(retained).astype(int)
```

```
    # Find condensed DOFs
    n = K.shape[0]
    x_C = np.setdiff1d(np.arange(n), x_R)

    #Bloc decomposition
    K_RR, K_CC, K_RC, K_CR = K_bloc(K, retained)

    # Build R (in compressed form first)
    I = np.eye(len(x_R))
    R_bottom = -np.linalg.solve(K_CC, K_CR)
    R_compact = np.vstack([I, R_bottom])

    # Now expand R_compact into full-size R by inserting rows in correct positions
    full_indices = np.concatenate([x_R, x_C])
    R = np.zeros((n, len(x_R)))
    for i, global_row in enumerate(full_indices):
        R[global_row, :] = R_compact[i, :]

    return R

#Define function to compute the reduced stiffness matrix
def K_gi(K, R):
    return R.T @ K @ R

#Define funuction to compute the reduced mass matrix
def M_gi(M, R):
    return R.T @ M @ R

#Define fuunction to compute the reduced force vector
def f_gi(f, R):
    return R.T @ f
```

File *display.py*

```
"""
DISPLAY

Description: This code contains functions to display the results.

Author: J. Adler

Academic year: 2024-2025
"""


import numpy as np
import matplotlib.pyplot as plt
import inputs
from mpl_toolkits.mplot3d import Axes3D
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
from matplotlib.animation import FuncAnimation, PillowWriter

#Define a function to display meshes
def plot_mesh(nodes, fixed=np.array([]), force_nodes=np.array([]), force_vectors=np.array([]), force_scale=1,
values=np.array([])):
    '''
    Parameters

    nodes : array
        Array containing the x, y and z coordinates of the nodes in the mesh (n,3).
    fixed : array, optional
        Array containing the x, y and z coordinates of the fixed nodes (n,3). The default is np.array([]).
    force_nodes : array, optional
        Array containing the x, y and z coordinates of the nodes on which the forces are applied (n,3).
        The default is np.array([]).
    force_vectors : array, optional
        Array containing the x, y and z coordinates of the force vectors (n,3). The default is np.array([]).
    force_scale: float
        Scale factor for the force vector.

    Raises

    ValueError
        Error when the shapes of the 'force_nodes'-array and 'force_vectors'-array are different.

    Returns

    Plots the mesh.

    '''

    if force_nodes.shape != force_vectors.shape:
        raise ValueError("Force arrays in 'plot_mesh' must have the same shape")

    #Nodes coordinates
    x = nodes[:,0] * 1000
    y = nodes[:,1] * 1000
    z = nodes[:,2] * 1000

    #Initialize figure
```

```
    fig = plt.figure()
    ax = plt.axes(projection ='3d')

    # Create cubic bounding box to simulate equal aspect ratio
    max_range = np.array([x.max()-x.min(), y.max()-y.min(), z.max()-z.min()]).max()
    Xb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][0].flatten() + 0.5*(x.max()+x.min())
    Yb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][1].flatten() + 0.5*(y.max()+y.min())
    Zb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][2].flatten() + 0.5*(z.max()+z.min())
    # Comment or uncomment following both lines to test the fake bounding box:
    for xb, yb, zb in zip(Xb, Yb, Zb):
       ax.plot([xb], [yb], [zb], 'w')

    #Plot the nodes
    ax.scatter(x, y, z, c = 'green', s=1)
    sc = ax.scatter(x, y, z, c = values, s=5, cmap="Spectral_r")
    plt.colorbar(sc, pad=0.1)

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    if len(fixed)!=0:

        #Fixed nodes coordinates
        x_f = fixed[:,0] * 1000
        y_f = fixed[:,1] * 1000
        z_f = fixed[:,2] * 1000

        #Plot the fixed nodes
        ax.scatter(x_f, y_f, z_f, c = 'black', s=10)


    if len(force_vectors)!=0:
        #Forces
        x_o = force_nodes[:,0] * 1000
        y_o = force_nodes[:,1] * 1000
        z_o = force_nodes[:,2] * 1000

        x_vec = force_vectors[:,0] * force_scale
        y_vec = force_vectors[:,1] * force_scale
        z_vec = force_vectors[:,2] * force_scale

        #Plot the forces
        for i in range(len(force_vectors)):
            ax.quiver(x_o[i], y_o[i], z_o[i], x_vec[i], y_vec[i], z_vec[i], color='red')

    #plt.savefig("Figures/impeller_no_clamps.pdf", format="pdf", bbox_inches="tight")
    plt.show()

    return


#Define a function to display bars
def plot_bars(nodes, bars, fixed=np.array([]), force_nodes=np.array([]), force_vectors=np.array([]), force_scale=1,
values=np.array([])):

    #Nodes coordinates
    x = nodes[:,0] * 1000
    y = nodes[:,1] * 1000
    z = nodes[:,2] * 1000

    #Initialize the plot
    fig = plt.figure()
    ax = plt.axes(projection ='3d')

    # Create cubic bounding box to simulate equal aspect ratio
    max_range = np.array([x.max()-x.min(), y.max()-y.min(), z.max()-z.min()]).max()
    Xb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][0].flatten() + 0.5*(x.max()+x.min())
    Yb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][1].flatten() + 0.5*(y.max()+y.min())
    Zb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][2].flatten() + 0.5*(z.max()+z.min())
    # Comment or uncomment following both lines to test the fake bounding box:
    for xb, yb, zb in zip(Xb, Yb, Zb):
       ax.plot([xb], [yb], [zb], 'w')

    for bar in bars:
        n1_index = int(bar[0]-1)
        n2_index = int(bar[1]-1)

        x1, y1, z1 = x[n1_index], y[n1_index], z[n1_index]
        x2, y2, z2 = x[n2_index], y[n2_index], z[n2_index]

        xs = [x1, x2]
        ys = [y1, y2]
        zs = [z1, z2]

        ax.scatter(x1, y1, z1, c = 'green', s=1)
        ax.scatter(x2, y2, z2, c = 'green', s=1)
        ax.plot3D(xs, ys, zs, c='gray')

    if len(fixed)!=0:

        #Fixed nodes coordinates
        x_f = fixed[:,0] * 1000
        y_f = fixed[:,1] * 1000
        z_f = fixed[:,2] * 1000
```

```
        #Plot the fixed nodes
        ax.scatter(x_f, y_f, z_f, c = 'black', s=10)

    if len(force_vectors)!=0:
        #Forces
        x_o = force_nodes[:,0] * 1000
        y_o = force_nodes[:,1] * 1000
        z_o = force_nodes[:,2] * 1000

        x_vec = force_vectors[:,0] * force_scale
        y_vec = force_vectors[:,1] * force_scale
        z_vec = force_vectors[:,2] * force_scale

        #Plot the forces
        for i in range(len(force_vectors)):
            ax.quiver(x_o[i], y_o[i], z_o[i], x_vec[i], y_vec[i], z_vec[i], color='red')

    #plt.savefig("Figures/bar_truss.pdf", format="pdf", bbox_inches="tight")
    plt.show()

    return

#Define a function to display the optimization solution
def plot_solution(nodeList, barNodes, sol):
    '''

    Parameters
    ----------
    nodeList : numpy array (n x 3)
        Contains the coordinates of the nodes.
    barNodes : numpy array (n x 2)
        Contains the nodes of the bars.
    sol : numpy array (n)
        Contains the solution for each bar.

    Returns
    -------
    Plots the solution.

    '''

    #Nodes coordinates
    x = nodeList[:,0] * 1000
    y = nodeList[:,1] * 1000
    z = nodeList[:,2] * 1000

    #Concerned nodes
    indices = (barNodes[:,0] - 1).astype(int)

    x_s = x[indices]
    y_s = y[indices]
    z_s = z[indices]

    #Initialize the plot
    fig = plt.figure()
    ax = plt.axes(projection ='3d')

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    #Plot the nodes
    #'inferno_r'
    sc = ax.scatter(x_s, y_s, z_s, c=sol, s=50, cmap='binary', alpha=1, depthshade=False)
    plt.colorbar(sc, pad=0.1)
    #ax.set_zlim3d([-50,100])

    #Show
    #plt.savefig("Figures/SLSQP_2.pdf", format="pdf", bbox_inches="tight")
    plt.show()

#Define a function the solution + the mesh
def plot_solution_mesh(nodeList, workpieceNodes, barNodes, forceNodes, sol, generateGIF=False):

    workpieceCoordinates = nodeList[np.sort(np.unique(workpieceNodes)).astype(int) - 1]
    barCoordinates = nodeList[np.sort(np.unique(barNodes)).astype(int) - 1]
    forceCoordinates = nodeList[np.sort(np.unique(forceNodes)).astype(int) - 1]

    #Workpiece nodes coordinates
    x = workpieceCoordinates[:,0] * 1000
    y = workpieceCoordinates[:,1] * 1000
    z = workpieceCoordinates[:,2] * 1000

    #Solution nodes coordinates
    x_s = barCoordinates[:,0] * 1000
    y_s = barCoordinates[:,1] * 1000
    z_s = barCoordinates[:,2] * 1000

    #Force nodes coordniates
    x_f = forceCoordinates[:,0] * 1000
    y_f = forceCoordinates[:,1] * 1000
    z_f = forceCoordinates[:,2] * 1000

    #Initialize figure
```

```
    fig = plt.figure(dpi=200)
    ax = plt.axes(projection ='3d')

    # Create cubic bounding box to simulate equal aspect ratio
    max_range = np.array([x.max()-x.min(), y.max()-y.min(), z.max()-z.min()]).max()
    Xb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][0].flatten() + 0.5*(x.max()+x.min())
    Yb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][1].flatten() + 0.5*(y.max()+y.min())
    Zb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][2].flatten() + 0.5*(z.max()+z.min())
    # Comment or uncomment following both lines to test the fake bounding box:
    for xb, yb, zb in zip(Xb, Yb, Zb):
       ax.plot([xb], [yb], [zb], 'w')

    #Plot the nodes
    #cmap: 'plasma', 'viridis', 'binary', 'inferno'
    ax.scatter(x, y, z, c = 'green', s=0.5, marker="8")
    ax.scatter(x_f, y_f, z_f, c = 'red', s=5, zorder=1)
    sc = ax.scatter(x_s, y_s, z_s, c=sol, s=12, cmap='binary')
    ax.quiver(x_f[0], y_f[0], z_f[0], 50, 50, 0, color='red')
    plt.colorbar(sc, pad=0.1)

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    #elevation = 30
    #alpha = -60 + 3 * 90
    #ax.view_init(elev=elevation, azim=alpha)
    #plt.savefig("Figures/impeller_post-process.pdf", format="pdf", bbox_inches="tight")
    plt.show()


    if generateGIF:
        def update(frame):
            ax.view_init(elev=30, azim=frame)
            return sc,

        # Create the animation
        ani = FuncAnimation(fig, update, frames=np.arange(0, 360, 2), interval=50)

        # Save to GIF (requires Pillow)
        ani.save("Figures/gif.gif", writer=PillowWriter(fps=15))

    return

#Define a function to display the entire mesh (workpiece + bars)
def plot(nodeList, workpieceNodes, barNodes, fixedList, loadList, force_scale=1):
    '''

    Parameters
    ----------
    nodeList : numpy array (n x 3)
        Contains the coordinates of each node.
    workpieceNodes : numpy array
        Contains the nodes of the elements.
    barNodes : numpy array (n x 2)
        Contains the nodes of the bars.
    fixedList : numpy array
        Contains the fixed nodes.
    loadList : numpy array (n x 4)
        Contains the loads applied on the workpiece.
    force_scale : float, optional
        Can be tuned to change the force vector scale. The default is 1.

    Returns
    -------
    Plots the workpiece, the bars, the fixed nodes and the applied loads.

    '''

    #Nodes coordinates
    x = nodeList[:,0] * 1000
    y = nodeList[:,1] * 1000
    z = nodeList[:,2] * 1000

    #Initialize the plot
    fig = plt.figure()
    ax = plt.axes(projection='3d')
    #ax.set_xlabel("x")
    #ax.set_ylabel("y")
    #ax.set_zlabel("z")

    # Create cubic bounding box to simulate equal aspect ratio
    max_range = np.array([x.max()-x.min(), y.max()-y.min(), z.max()-z.min()]).max()
    Xb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][0].flatten() + 0.5*(x.max()+x.min())
    Yb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][1].flatten() + 0.5*(y.max()+y.min())
    Zb = 0.5*max_range*np.mgrid[-1:2:2,-1:2:2,-1:2:2][2].flatten() + 0.5*(z.max()+z.min())
    # Comment or uncomment following both lines to test the fake bounding box:
    for xb, yb, zb in zip(Xb, Yb, Zb):
       ax.plot([xb], [yb], [zb], 'w')

    #Plot the bars
    for bar in barNodes:
        n1_index = int(bar[0]-1)
        n2_index = int(bar[1]-1)
```

```
        x1, y1, z1 = x[n1_index], y[n1_index], z[n1_index]
        x2, y2, z2 = x[n2_index], y[n2_index], z[n2_index]

        xs = [x1, x2]
        ys = [y1, y2]
        zs = [z1, z2]

        #ax.scatter(x1, y1, z1, c = 'green', s=1)
        #ax.scatter(x2, y2, z2, c = 'green', s=1)
        ax.plot3D(xs, ys, zs, c='blue')

    #Plot the workpiece
    indices = (np.unique(workpieceNodes) - 1).astype(int)

    x_w = x[indices]
    y_w = y[indices]
    z_w = z[indices]

    ax.scatter(x_w, y_w, z_w, c = 'green', s=1)

    #Plot fixed nodes
    indices = (fixedList - 1).astype(int)

    x_f = x[indices]
    y_f = y[indices]
    z_f = z[indices]

    ax.scatter(x_f, y_f, z_f, c = 'black', s=5)

    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')

    #Plot loads
    for load in loadList:
        index = (load[0] - 1).astype(int)

        x_o = x[index]
        y_o = y[index]
        z_o = z[index]

        x_vec, y_vec, z_vec = load[1:] * force_scale

        ax.quiver(x_o, y_o, z_o, x_vec, y_vec, z_vec, color='red')

    #Show
    #plt.savefig("Figures/cylinder_1.pdf", format="pdf", bbox_inches="tight")
    plt.show()
```

File *validation.py*

```
"""
VALIDATION
----------

Description: This code is aimed at validation the FEM code from the main files.

Author: J. Adler

Academic year: 2024-2025
"""

import numpy as np
import scipy
import inputs
import assembly
import display
import reduction


if False:
    #VALIDATION (CANTILEVER STIFFNESS)
    #---------

    #Compute stiffness matrix
    K_w = assembly.K_workpiece()
    #Compute force vector
    F = assembly.f()

    #Apply boundary conditions
    K_w = assembly.applyBC(K_w)
    F = assembly.applyBC(F, dim=1)

    #Solve the system
    q = np.linalg.solve(K_w, F)

    #Compute the new node coordinates
    Q = assembly.q_full(q)
    Q = Q.reshape(-1,3)

    newNodeList = inputs.nodeList + Q
```

```
    #Display the deformed mesh
    fixed = inputs.fixedList - 1
    fixed = inputs.nodeList[fixed.astype(int)]

    force_nodes = np.array(inputs.loadList[:,0] - 1, dtype=int)
    force_nodes = inputs.nodeList[force_nodes]

    force_vectors = inputs.loadList[:,1:]

    displacement_values = np.sqrt(np.sum(Q**2, axis=1))

    display.plot_mesh(newNodeList,
                      fixed=fixed,
                      force_nodes=force_nodes,
                      force_vectors=force_vectors,
                      force_scale=1,
                      values=displacement_values*1000)

    print("Max-displacement:", np.round(displacement_values.max()*1000, 3), "[mm]")


if False:
    #VALIDATION (BARS STIFFNESS)
    #-----------

    #Compute stiffness matrix
    K_b = assembly.K_bars(np.ones(3))
    #Compute force vector
    F = assembly.f()

    #Apply boundary conditions
    K_b = assembly.applyBC(K_b)
    F = assembly.applyBC(F, dim=1)

    #Solve the system
    q = np.linalg.solve(K_b, F)

    #Plot the initial bars
    bars = inputs.elemListBars

    fixed = np.array(inputs.fixedList - 1, dtype=int)
    fixed = inputs.nodeList[fixed]

    force_nodes = np.array(inputs.loadList[:,0]-1, dtype=int)
    force_nodes = inputs.nodeList[force_nodes]

    force_vectors = inputs.loadList[:,1:]

    display.plot_bars(inputs.nodeList,
                      bars,
                      fixed=fixed,
                      force_nodes=force_nodes,
                      force_vectors=force_vectors,
                      force_scale=5e-3)

    #Compute the new node coordinates
    Q = assembly.q_full(q)
    Q = Q.reshape(-1,3)

    newNodeList = inputs.nodeList + Q

    #Plot the deformed bars
    force_nodes = np.array(inputs.loadList[:,0]-1, dtype=int)
    force_nodes = newNodeList[force_nodes]

    display.plot_bars(newNodeList,
                      bars,
                      fixed=fixed,
                      force_nodes=force_nodes,
                      force_vectors=force_vectors,
                      force_scale=5e-2)

    #Node displacement
    node_displacement = np.sqrt(np.sum(q**2))

    print("Node-displacement:", np.round(node_displacement*1000,3))

if False:
    #VALIDATION (BARS MASS)
    #-----------

    #Compute stiffness matrix
    K_b = assembly.K_bars(np.ones(3))
    #Compute mass matrix
    M_b = assembly.M_bars(np.ones(3))

    #Apply boundary conditions
    K_b = assembly.applyBC(K_b)
    M_b = assembly.applyBC(M_b)

    # K and M should be symmetric and positive definite
    eigenvals, eigenvecs = scipy.linalg.eigh(K_b, M_b)

    # Eigenfrequencies (rad/s)
```

```
    omega = np.sqrt(eigenvals)

    # Frequencies in Hz
    freqs = omega / (2 * np.pi)

    print("Frequencies (Hz):", freqs)


if False:
    #VALIDATION (CANTILEVER MASS)
    #------------

    #Compute stiffness matrix
    K_w = assembly.K_workpiece()
    #Compute mass matrix
    M_w = assembly.M_workpiece()

    #Apply boundary conditions
    K_w = assembly.applyBC(K_w)
    M_w = assembly.applyBC(M_w)

    # K and M should be symmetric and positive definite
    eigenvals, eigenvecs = scipy.linalg.eigh(K_w, M_w)

    # Eigenfrequencies (rad/s)
    omega = np.sqrt(eigenvals)

    # Frequencies in Hz
    freqs = omega / (2 * np.pi)

    print("Frequencies (Hz):", freqs)

    #Display the deformed mesh
    Q = assembly.q_full(eigenvecs[:,2])
    Q = Q.reshape(-1,3)

    newNodeList = inputs.nodeList + Q/2

    fixed = inputs.fixedList - 1
    fixed = inputs.nodeList[fixed.astype(int)]

    displacement_values = np.sqrt(np.sum(Q**2, axis=1))

    display.plot_mesh(newNodeList,
                      fixed=fixed,
                      values=displacement_values*10/3)


if True:
    #VALIDATION (GUYAN-IRONS, CANTILEVER)
    #------------

    #Compute stifness matrix
    K_w = assembly.K_workpiece()
    M_w = assembly.M_workpiece()
    #Compute force vector
    f = assembly.f()

    #Apply boundary conditions
    K_w = assembly.applyBC(K_w)
    M_w = assembly.applyBC(M_w)
    f = assembly.applyBC(f, dim=1)

    #Guyan-Irons
    retained_dofs, condensed_dofs = reduction.get_relevant_dofs()
    retained = reduction.get_retained_indices(retained_dofs)
    R = reduction.R_gi(K_w, retained)
    K_red = reduction.K_gi(K_w, R)
    M_red = reduction.M_gi(M_w, R)
    f_red = reduction.f_gi(f, R)

    q_red = np.linalg.solve(K_red, f_red)
    q = R @ q_red
    Q = assembly.q_full(q)
    Q = Q.reshape(-1,3)

    newNodeList = inputs.nodeList + Q

    #Display the deformed mesh
    fixed = inputs.fixedList - 1
    fixed = inputs.nodeList[fixed.astype(int)]

    force_nodes = np.array(inputs.loadList[:,0] - 1, dtype=int)
    force_nodes = inputs.nodeList[force_nodes]

    force_vectors = inputs.loadList[:,1:]

    displacement_values = np.sqrt(np.sum(Q**2, axis=1))

    display.plot_mesh(newNodeList,
                      fixed=fixed,
                      force_nodes=force_nodes,
                      force_vectors=force_vectors,
                      force_scale=1,
                      values=displacement_values*1000)
```

```
    #Eigenvalue problem
    eigenvals, eigenvecs = scipy.linalg.eigh(K_red, M_red)
    eigenvecs = R @ eigenvecs

    # Eigenfrequencies (rad/s)
    omega = np.sqrt(eigenvals)

    # Frequencies in Hz
    freqs = omega / (2 * np.pi)

    #Display the deformed mesh
    Q = assembly.q_full(eigenvecs[:,1])
    Q = Q.reshape(-1,3)

    newNodeList = inputs.nodeList + Q/2

    fixed = inputs.fixedList - 1
    fixed = inputs.nodeList[fixed.astype(int)]

    displacement_values = np.sqrt(np.sum(Q**2, axis=1))

    display.plot_mesh(newNodeList,
                      fixed=fixed,
                      values=displacement_values*10/3)

    print("Maximum displacement =", np.round(displacement_values.max()*1000, 3), "[mm]")
    print("Eigenfrequencies =", freqs, "[Hz]")
```

File *optimization_scipy.py*

```
"""
OPTIMIZATION (SCIPY)

Description: This code runs the optimization process.

Author: J. Adler

Academic year: 2024-2025
"""


import numpy as np
from scipy.optimize import minimize
import inputs
import assembly
import display
import time

#Define the number of bars in the solution
V = 16

#Define the objective function to minimize
def objective(x):

    K_w = assembly.K_workpiece()
    K_b = assembly.K_bars(x)
    K = K_w + K_b

    F = assembly.f()

    K = assembly.applyBC(K)
    F = assembly.applyBC(F, dim=1)

    q = np.linalg.solve(K, F) * 1e6
    q_sq = q**2

    return q_sq.max()

#Define the inequality constraints
def ineq_constraints(x):

    return V - x.sum()

# Define the callback function
def callback(xk):
    print(f"Current iteration result: {xk}\n")

#Define the initial guess
initial_guess = np.ones(inputs.barElem)*V/inputs.barElem

#Define the bounds
bounds = [(0, 1) for _ in range(inputs.barElem)]

#Define the constraints
constraints = ({'type': 'ineq', 'fun': ineq_constraints})

#Start timer
start = time.time()
```

```
#Minimize the objective function with the imposed constraints
opt = minimize(objective,
               initial_guess,
               method='SLSQP',
               bounds=bounds,
               constraints=constraints,
               callback=callback)

#End timer
end = time.time()
elapsed_time = np.round(end - start, 1)
print("Elapsed time:", elapsed_time, "sec")

#Print the results
print(opt)

#Plot the solution
sol = opt['x']
display.plot_solution(inputs.nodeList, inputs.elemListBars, sol)
```

File *optimization_MMA.py*

```
"""
OPTIMIZATION (MMA)
-------------------

Description: This code applies the MMA code from Arjen Deetman to the optimization of clamping locations.
             Note: the original code for the MMA solver + examples has been written by Krister Svanberg in MATLAB
             and has been translated to Python by Arjen Deetman.

Author: J. Adler

Academic year: 2024-2025
"""

from __future__ import division

import sys
import os

# Manually specify the absolute path to the mmapy/src folder
mma_src_path = "MMA/arjendeetman-GCMMA-MMA-Python-ce86d94/src"

# Add the path to Python's module search path
if mma_src_path not in sys.path:
    sys.path.append(mma_src_path)

from mmapy import mmasub, kktcheck
from util import setup_logger
from typing import Tuple
import numpy as np
import scipy
import inputs
import assembly
import sensitivity
import display
import time

#Start timer
start = time.time()

#Number of bars remaining at the end
V = 10
Omega = 4000 #[Hz]

def solve(m, n, optimization_problem, formulation='standard', q=1, log_name='default'):
    '''
    Parameters
    ----------
    m : int
        Number of constraint functions.
    n : int
        Number of design variables.
    optimization_problem : function (xval: np.ndarray) -> Tuple[float, np.ndarray, np.ndarray, np.ndarray]
        Description of the optimization problem. This function return the value of the objective and constraint
        functions at the current point as well as the values of the derivatives at the current design point.
    log_name : string, optional
        Name of the logfile that is generated when calling the function. The default is 'default'.

    Returns
    -------
    The solution to the structural optimization problem.

    '''
    # Logger
    path = os.path.dirname(os.path.realpath(__file__))
    file = os.path.join(path, log_name+".log")
    logger = setup_logger(file)
    logger.info("Started\n")

    # Set numpy print options
```

```
    np.set_printoptions(precision=4, formatter={'float': '{:-0.4f}'.format})

    # Initial settings
    #m, n = 2, 3
    eeen = np.ones((n, 1))
    eeem = np.ones((m, 1))
    zeron = np.zeros((n, 1))
    zerom = np.zeros((m, 1))
    xval = np.ones(n).reshape(-1,1) *V/n
    xold1 = xval.copy()
    xold2 = xval.copy()
    xmin = zeron.copy()
    xmax = 1 * eeen
    low = xmin.copy()
    upp = xmax.copy()
    move = 1.0
    c = 1000 * eeem
    d = eeem.copy()
    a0 = 1

    if formulation=='standard':
        a = zerom.copy()
    elif formulation=='min-max':
        a = eeem.copy()
        a[m-q:] = np.zeros((2,1))

    innerit = 0
    outeriter = 0
    maxoutit = 99
    kkttol = 1e-3

    # Calculate function values and gradients of the objective and constraints functions
    if outeriter == 0:
        f0val, df0dx, fval, dfdx = optimization_problem(xval)
        outvector1 = np.concatenate((np.array([outeriter, innerit, f0val]), fval.flatten()))
        outvector2 = xval.flatten()

        # Log
        logger.info("outvector1 = {}".format(outvector1))
        logger.info("outvector2 = {}\n".format(outvector2))

    # The iterations start
    kktnorm = kkttol + 10
    outit = 0

    while kktnorm > kkttol and outit < maxoutit:
        outit += 1
        outeriter += 1

        # The MMA subproblem is solved at the point xval:
        xmma, ymma, zmma, lam, xsi, eta, mu, zet, s, low, upp = mmasub(
            m, n, outeriter, xval, xmin, xmax, xold1, xold2, f0val, df0dx, fval, dfdx, low, upp, a0, a, c, d, move)

        # Some vectors are updated:
        xold2 = xold1.copy()
        xold1 = xval.copy()
        xval = xmma.copy()

        # Re-calculate function values and gradients of the objective and constraints functions
        f0val, df0dx, fval, dfdx = optimization_problem(xval)

        # The residual vector of the KKT conditions is calculated
        residu, kktnorm, residumax = kktcheck(
            m, n, xmma, ymma, zmma, lam, xsi, eta, mu, zet, s, xmin, xmax, df0dx, fval, dfdx, a0, a, c, d)
        #print("Z=",zmma)
        #print("Y=",ymma)
        outvector1 = np.concatenate((np.array([outeriter, innerit, f0val]), fval.flatten()))
        outvector2 = xval.flatten()

        # Log
        logger.info("outvector1 = {}".format(outvector1))
        logger.info("outvector2 = {}".format(outvector2))
        logger.info("kktnorm.... = {}\n".format(kktnorm))

    # Final log
    logger.info("Finished")
    return outvector2, outit

#Number of function evaluations
nfev = np.array([0])
fvals = []

def increment(nfev, fval0, fval1, fval2):
    nfev[0] += 1
    fvals.append([fval0, fval1, fval2])



'''Optimization problem without reduction'''
if True:
    #Number of constraints functions
    m = 1
    q = 1

    #Mesh data
```

```
    K_w = assembly.applyBC(assembly.K_workpiece())
    F = assembly.applyBC(assembly.f_individual(), dim=1)

    #Optimization problem with only the 1st constraint and for a single load (no reduction !)
    def clamploc(xval: np.ndarray) -> Tuple[float, np.ndarray, np.ndarray, np.ndarray]:
        #Objective function value:
        K_b = assembly.applyBC(assembly.K_bars(xval.flatten()))
        K = K_w + K_b

        #Solve the system
        q = np.linalg.solve(K, F) * 1e6
        q_sq = q**2
        f0val = q_sq.max()

        #Sensitivity of the objective function:
        df0dx = sensitivity.dc0_dx(xval.flatten(), K, q).reshape(-1, 1)

        #Constraint function values:
        fval1 = xval.sum() - V
        fval = np.array([[fval1]])

        #Derivative of constraint functions:
        dfdx1 = sensitivity.dc1_dx()
        dfdx = np.array([dfdx1])

        increment(nfev, f0val, fval1, 0)
        return f0val, df0dx, fval, dfdx

    formulation = 'standard'
    opt_prob = clamploc


'''Optimization problem with reduction'''
if False:
    #Number of constraints functions
    m = 1
    q = 1

    #Mesh data
    K_w_tilde, R = assembly.K_workpiece_tilde()
    F_tilde = assembly.f_individual_tilde(R)

    #Optimization problem for multiple load cases (with reduction !)
    def clamplocred(xval: np.ndarray) -> Tuple[float, np.ndarray, np.ndarray, np.ndarray]:
        #Objective function value:
        K_b_tilde = assembly.K_bars_tilde(xval.flatten())
        K_tilde = K_w_tilde + K_b_tilde

        #Solve the system
        q_tilde = np.linalg.solve(K_tilde, F_tilde) * 1e6
        q = R @ q_tilde
        q_sq = q**2
        f0val = q_sq.max()

        #Sensitivity of the objective function:
        df0dx = sensitivity.dc0_dx_tilde(xval.flatten(), R, q, q_tilde, K_tilde).reshape(-1, 1)

        #Constraint function values:
        fval1 = xval.sum() - V
        fval = np.array([[fval1]])

        #Derivative of constraint functions:
        dfdx1 = sensitivity.dc1_dx()
        dfdx = np.array([dfdx1])

        increment(nfev, f0val, fval1, 0)
        return f0val, df0dx, fval, dfdx

    formulation = 'standard'
    opt_prob = clamplocred


'''Optimization problem with reduction and lower bound
   on the 1st eigenfrequency'''
if False:
    #Number of constraints functions
    m = 2
    q = 2

    #Mesh data
    K_w_tilde, R = assembly.K_workpiece_tilde()
    M_w_tilde = assembly.M_workpiece_tilde(R)
    F_tilde = assembly.f_individual_tilde(R)

    #Optimization problem for multiple load cases (with reduction !) and
    #with constraint on the 1st eigenfrequency
    def clamplocred_freq(xval: np.ndarray) -> Tuple[float, np.ndarray, np.ndarray, np.ndarray]:
        #Objective function value:
        K_b_tilde = assembly.K_bars_tilde(xval.flatten())
        K_tilde = K_w_tilde + K_b_tilde

        #Solve the system
        q_tilde = np.linalg.solve(K_tilde, F_tilde) * 1e6
        q = R @ q_tilde
        q_sq = q**2
```

```
        f0val = q_sq.max()

        #Sensitivity of the objective function:
        df0dx = sensitivity.dc0_dx_tilde(xval.flatten(), R, q, q_tilde, K_tilde).reshape(-1, 1)

        #Constraint function values:
        fval1 = xval.sum() - V

        M_b_tilde = assembly.M_bars_tilde(xval.flatten())
        M_tilde = M_w_tilde + M_b_tilde
        eigenvals, eigenvecs = scipy.linalg.eigh(K_tilde, M_tilde)
        w1 = eigenvals[0]
        freq = np.sqrt(w1)/(2*np.pi)
        q1 = eigenvecs[:,0]

        fval2 = Omega - freq
        fval = np.array([[fval1],
                         [fval2]])

        #Derivative of constraint functions:
        dfdx1 = sensitivity.dc1_dx()
        dfdx2 = sensitivity.dc2_dx(xval.flatten(), w1, q1, reduced=True) / (8*np.pi**2 * freq)
        dfdx = np.array([dfdx1, dfdx2])

        increment(nfev, f0val, fval1, fval2)
        return f0val, df0dx, fval, dfdx

    formulation = 'standard'
    opt_prob = clamplocred_freq


#Solve the optimization problem
opt = solve(m, inputs.barElem, opt_prob, formulation=formulation, q=q)
x_star = opt[0]
nit = opt[1]

#End timer
end = time.time()
elapsed_time = np.round(end - start, 1)
print("Elapsed time:", elapsed_time, "sec")

#Plot the solution
display.plot_solution(inputs.nodeList, inputs.elemListBars, x_star)
print(inputs.dofs)
```