Density based topology optimization with the Null Space Optimizer: a tutorial and a comparison

F Feppon

Density based Topology Optimization with the Null Space Optimizer: a tutorial and a comparison

Florian Feppon[1*]

[1*]NUMA Unit, Department of Computer Science, KU Leuven, Belgium.

Corresponding author(s). E-mail(s): florian.feppon@kuleuven.be;

Abstract

The Null Space Optimizer is a constrained optimization solver that has been developed in the context of level-set based Topology Optimization. One of its appealing aspects comes from its relative independence to the need for tuning unintuitive algorithm parameters. The first contribution of this paper is to introduce an upgrade of the Null Space Optimizer that enables to solve optimization problems featuring a large number of constraints with sparse Jacobian matrix. This allows to include in particular bound constraints, making it possible to use the Null Space Optimizer for solving density based Topology Optimization problems.

The second contribution of the paper is to present three tutorials giving an educational view on how to use the open source Python implementation of the Null Space Optimizer for solving Topology Optimization problems in structural mechanics and conductive heat transfer, on structured and unstructured meshes. Elegant Python programming features are used for automating the implementation of density filters and the assembly of sparse Jacobian matrices.

Numerical results are presented on three design problems and compared to those obtained with the Method of Moving Asymptotes (MMA), the Interior Point Optimizer IPOPT and the Optimality Criteria (OC) method. We found that on the situations considered, (i) the Null Space optimizer is able to compute optimized designs with performances comparable to its competitors with very little parameter tuning, (ii) the OC method or IPOPT with default parameters sometimes converge to non-optimal designs, (iii) MMA sometimes converge to slightly better design with a faster decay than the Null Space Optimizer during the first iterations, but may also require case dependent fixes to converge to satisfactory solutions.

Keywords: Nonlinear constrained optimization, density based Topology Optimization, null space gradient flows, Python.

1 Introduction

The Null Space optimization algorithm (Feppon et al (2020a)) was introduced in the context of level-set based Topology Optimization (TO) (Allaire et al (2021)) as a convenient alternative to the *Augmented Lagrangian* (AL) (Nocedal and Wright (2006); Wang and Wang (2006); Xia and Wang (2008); Allaire et al (2013); Deng and Suresh (2016); Emmendoerfer Jr et al (2020)), and to the *Sequential Linear Programming* (SLP) methods (Dunning and Kim (2015); Dunning et al (2015); Liu et al (2021, 2016)). One of the motivations for developing the Null Space algorithm stemmed from the need for efficient constrained optimization solvers that work with little to no tuning of case dependent algorithm parameters. Such tuning is indeed associated with a number

of inconveniences: (i) multiple runs of the optimization are needed in order to find the right parameters leading to a satisfactory optimized design, which is unaffordable for large-scale applications; (ii) the tuning may be quite complex to achieve in situations involving a large number of design constraints; (iii) it limits TO to a set of predefined applications for its use by non-experts in industrial contexts.

The Null Space algorithm is a first-order constrained optimization method that evolves the design variable by solving an Ordinary Differential Equation (ODE). The trajectories of this ODE minimize the objective function while smoothly correcting the violation of the constraints. As a consequence, the sole true algorithm parameter is the time step, which only needs to be taken sufficiently small for ensuring the minimization property along the integrated trajectories. The Null Space algorithm has proved to be quite effective for solving a number of large-scale multiphysics design problems involving multiple physical constraints with level-set based methods (see e.g. Feppon (2019); Feppon et al (2020b, 2021); Salazar De Troya et al (2021)). It allows to deal with both inequality and equality constraints and corrects unfeasible initializations. Its implementation has been released as a freely available open-source Python package called "Null Space Optimizer"[1].

When it was introduced in Feppon et al (2020a), the Null Space Optimizer did not allow to solve optimization problems featuring a very large number of constraints: indeed the original implementation required to explicitly compute the inverse of a matrix of size the number of active constraints. In the Appendix C, we detail a revised implementation that takes advantage of sparsity to avoid this explicit inversion. This allows to extend the Null Space algorithm to constrained optimization problems with a large number of constraints provided these have a sparse Jacobian matrix.

As a straightforward by-product of this upgrade, the Null Space Optimizer now handles *bound-constraints*, which makes it applicable for solving density based Topology Optimization problems. In the density-based TO framework (Bendsoe and Sigmund (2003)), the design is parameterized with density variables $(\rho_i)_{1\leq i\leq n}$ which need to satisfy $0 \leq \rho_i \leq 1$ for every $1 \leq i \leq n$, where n is the number of elements discretizing the computational domain. In contrast with the level-set framework where these bound constraints do not exist, this entails the need to deal with twice as many additional inequality constraints as the number n of design variables.

With this extension at hand, this paper provides a tutorial view on how to use the Python implementation of the Null Space Optimizer for conveniently solving Topology Optimization problems in the density framework. The Null Space Optimizer offers an elegant paradigm for implementing Topology Optimization problems. It automates the use of filters for the design variable, the memoization of the physical solver and the assembly of the Jacobian matrices of the bound constraints. Moreover, it is interfaced with Python implementations of the popular Method of Moving Asymptotes (MMA) (Svanberg (1987); Deetman (2020)), of the Optimality Criteria (OC) method (Bendsoe and Sigmund (2003)), as well as with the primal-dual interior point method optimizer IPOPT (Curtis et al (2012)), which allows us to conveniently provide comparisons with these algorithms.

So far, MMA has been considered as the "gold standard" algorithm for solving density based TO problems with arbitrary inequality constraints (Gersborg-Hansen et al (2005); Wang et al (2011); Sigmund and Maute (2013); Jensen and Sigmund (2011); Liang et al (2019); Zheng et al (2023); Liang et al (2023)). Alternatively, IPOPT has also been used in a number of applications (Swartz et al (2021); Alonso et al (2018); Alonso and Silva (2021); Sá et al (2021)). Although these methods have proved effective in many situations, both MMA and IPOPT remain sensitive to algorithm parameter tuning, requiring sometimes case-dependent fixes to obtain satisfactory optimized designs. For instance, MMA requires the objective and the constraint functions to be rescaled in order to map their values to a bounded interval (Svanberg (2014)), which can be challenging when these functions encounter large variations. In some situations, it may additionally require structural changes of the moving asymptote rules to lead to satisfactory convergence. IPOPT, on the other hand, may require

[1] https://gitlab.com/florian.feppon/null-space-optimizer/

expert tuning of algorithm parameters (such as the strategy for updating the barrier parameter) to obtain good performances (Rojas-Labanda and Stolpe (2015)). The OC method, is often favored in situations featuring a single equality constraint (Andreassen et al (2011); Aage et al (2017)), although some generalizations to multiple constraints have been proposed (Yin and Yang (2001); Kim et al (2021)).

In the present paper, we consider four benchmark cases in which, the Null Space Optimizer appears to be competitive with standard methods with little to no parameter tuning. Depending on the situations, we found that OC and IPOPT converge to a less performant design, while MMA may require some case dependent tuning to achieve performances similar or slightly better to that of the Null Space algorithm. Our numerical results suggest that (i) the Null Space Optimizer can be a suitable alternative to MMA and IPOPT for density based Topology Optimization, and (ii) it allows to compute designs that have comparable performance to its competitors without the need for any particular structural change to the algorithm, rescaling of the constraints or the tuning of nonphysical parameters.

Please note that this paper does not aim to compare more extensively various optimization solvers: we do not claim the superiority of the Null Space Optimizer in every scenario since we have considered only a limited range test cases and optimization solvers, and we have not invested significant effort in fine-tuning the parameters. Due to these limitations, we have chosen not to provide runtime comparisons, since they can be heavily influenced by parameter tuning and other factors, such as the use of precompiled routines. Actually, our numerical results of Sections 4 and 5 suggest a superiority of MMA to find more efficient initial design updates, but this advantage seems to be lost after 70 to 80 iterations. The reader interested in further comparisons between various optimizers on benchmark test cases in density based TO may refer to Fanni et al (2013); Rojas-Labanda and Stolpe (2015).

An overview on the principles of the Null Space optimization algorithm is given in Section 2. The necessary technical updates to account for optimization constraints with sparse Jacobian, including bound constraints are included in the Appendix C. This appendix also presents a first benchmark numerical example challenging the enforcement of bound constraints by the Null Space Optimizer and compares the results with MMA and IPOPT.

The next three sections provide tutorials for implementing three 2D Topology Optimization test cases with the Null Space Optimizer as well as comparisons with MMA, IPOPT and the OC method.

Section 3 details how to implement the classical MBB beam compliance minimization problem with a single volume constraint from the popular 88 lines Matlab code (Andreassen et al (2011); Aage and Johansen (2016)). The complete code for solving this example with the Null Space Optimizer is provided in Appendix A. Then, we test the different optimizers on a more challenging version of this problem obtained by imposing a very small volume fraction. This example is interesting in that the MMA algorithm needs a major structural fix to obtain a satisfactory convergence, while the IPOPT optimizer struggles to find a performant optimum. We find that the Null Space Optimizer only requires a sufficiently small time step to find a satisfactory solution.

Section 4 considers the design optimization of a bridge featuring multiple loads as more complex test case that precludes the use the OC method. This test case is implemented by adding very minor modifications to the code of Appendix A. On this example, we find that without extensive tuning, MMA converges to a solution that is slightly better than the one computed by the Null Space Optimizer, while the one computed by IPOPT is slightly worse.

Section 5 provides an additional tutorial describing the implementation of the Topology Optimization of a heat sink on unstructured meshes, involving thus a different physical model and a different finite element discretization. The complete source code for this example is reproduced in Appendix B; it is constituted of a main Python file interfaced with two scripts written in the FreeFEM language (Hecht (2012)), which is especially convenient for dealing with the finite element method on unstructured meshes. For this particular example that involves a single volume constraint, we find that the Null Space algorithm,

MMA and IPOPT converge to designs with similar performances, while the popular Optimality Criteria method converges to a local minimum with worse performance.

Throughout the paper, we relied on the following implementations of MMA, IPOPT and OC for the purpose of our numerical comparisons:

(i) we use the Python implementation of the *Method of Moving Asymptotes* (MMA) (Svanberg (1993)) provided by Arjen Deetman[2] that closely matches the original Matlab implementation distributed by K. Svanberg (Svanberg (2009, 2014)). In order to improve MMA's performance, we systematically rescale the objective function so that the initial value is equal to 10. Since MMA does not handle by default equality constraints, a volume equality constraint is treated as an inequality constraint when using this optimizer in Sections 3 and 4.

(ii) We rely on the Python interface cyipopt[3] in order to use the primal-dual Interior Point Optimizer IPOPT (Wächter and Biegler (2006); Curtis et al (2012)). Since the optimizer involves many parameters that are not straightforward to tune, we systematically run the optimizer with the default options.

(iii) The *Optimality Criteria* (OC) method (Bendsoe and Sigmund (2003)) is implemented following the source code of Aage and Johansen (2016).

2 Presentation of the Null Space Optimizer

The Null Space algorithm is a numerical method for solving generic nonlinear constrained optimization problems of the form

$$\min_{x\in\mathcal{X}} \quad J(x) \qquad s.t. \begin{cases} g_i(x) = 0 \text{ for } 1 \le i \le p, \\ h_j(x) \le 0 \text{ for } 1 \le j \le q. \end{cases} \tag{1}$$

Here, $\mathcal{X}$ denotes the design set, x is the design variable and the functions J, $\boldsymbol{g} := (g_i)_{1\le i\le p}$ and $\boldsymbol{h} := (h_j)_{1\le j\le q}$ are respectively the objective function, p equality constraints and q inequality constraints. For density based Topology Optimization, the design set is the finite-dimensional space $\mathcal{X} = \mathbb{R}^n$, where n is the number of cells or mesh elements discretizing the design domain.

2.1 Null Space gradient flows

The basis of the Null Space algorithm for solving the optimization problem (1) is to numerically integrate the following Ordinary Differential Equation (ODE), or so-called "null space gradient flow":

$$\dot{x}(t) = -\alpha_J \xi_J(x(t)) - \alpha_C \xi_C(x(t)). \tag{2}$$

The trajectories $x(t)$ are determined by two orthogonal vector fields ξ_J and ξ_C referred to as the "null space" and "range space" directions, following denominations of null space optimization methods for equality constrained problems, see e.g. Nocedal and Wright (2006); Nie (2004); Liu and Yuan (2010). The null space step $-\xi_J$ is the best locally feasible ascent direction. The range space step $-\xi_C$ is a Gauss-Newton direction that gradually corrects the violation of the constraints. The precise definitions of ξ_J and ξ_C are reviewed in Sections 2.2 and 2.3 below. All in all, (2) is a generalization of the standard gradient flow $\dot{x} = -\nabla J(x)$ minimizing J without constraints. Similarly to the unconstrained case, the trajectories $x(t)$ of (2) tend to converge to feasible designs x^* that are locally optimal in the sense that they satisfy the first order Karush-Kuhn-Tucker (KKT) conditions, namely x^* is feasible and

$$\exists \boldsymbol{\lambda} \in \mathbb{R}^p,\ \exists \boldsymbol{\mu} \in \mathbb{R}^q_+, \quad \begin{cases} \nabla J(x^*) + \mathrm{D}\boldsymbol{g}(x^*)^T \boldsymbol{\lambda} + \mathrm{D}\boldsymbol{h}(x^*)^T \boldsymbol{\mu} = 0, \\ \mu_j > 0 \Rightarrow h_j(x^*) = 0 \text{ for all } 1 \le j \le q, \end{cases} \tag{3}$$

where $\mathbb{R}^q_+$ denotes the q-dimensional nonnegative orthant and $\nabla J(x^*)$, $\mathrm{D}\boldsymbol{g}(x^*)$ and $\mathrm{D}\boldsymbol{h}(x^*)$ are respectively the gradient of J and the Jacobian matrices of $\boldsymbol{g}$ and $\boldsymbol{h}$ at x^*.

The ODE (2) is parameterized by two positive constants $\alpha_J > 0$ and $\alpha_C > 0$. The parameter α_J tunes the rate at which the objective function values decrease while not worsening the constraints.

[2] https://github.com/arjendeetman/GCMMA-MMA-Python

[3] https://github.com/mechmotum/cyipopt

The parameter α_C tunes the pace at which the violation of the constraints decreases (the violation decreases precisely at the exponential rate $e^{-\alpha_C t}$). In principle, the success of the method to find a local minimizer does not depend on the choice of α_J and α_C: it depends only on the selection of a sufficiently small time step. However, setting the ratio α_J/α_C to a custom value sometimes helps to find a better optimization path because it tunes the trade-off between guiding quickly the design to a region with small objective function values and driving it quickly to the feasible set. In any case, we emphasize that these parameters have a clear intuitive meaning and in many situations (including those considered in this paper), it is possible to set $\alpha_J/\alpha_C = 1$. For a more flexible usage, the Python implementation of the Null Space Optimizer implements automatic rescaling rules on the parameters α_J and α_C, see Section 2.5.

Remark 1. The Null Space algorithm is not restricted to optimization on finite-dimensional spaces but also allows to consider much more general sets equipped with a kind of manifold structure, see Feppon (2019), chapter 6. This feature is essential for level-set based Topology Optimization, where the design set is the set of shapes $\mathcal{X} = \{\Omega \subset D\}$ for a given computational domain D.

Remark 2. In (3) and throughout the paper, we denote by $\mathrm{D}J(x)$ and $\mathrm{D}\boldsymbol{g}(x)$ the Fréchet derivatives of a function J and a vector $\boldsymbol{g}$, where $\mathrm{D}J(x) := \begin{bmatrix}\partial_{x_1}J(x) & \dots & \partial_{x_n}J(x)\end{bmatrix}$ is understood as a *row* vector, and where $\mathrm{D}\boldsymbol{g}(x) := (\partial_j g_i(x))$ is the Jacobian matrix of $\boldsymbol{g}$. We use the gradient notation to refer to the transpose $\nabla J(x) := \mathrm{D}J(x)^T$. This distinction between Fréchet derivative and gradient is needed in the Hilbertian framework of level-set based Topology Optimization; see Feppon et al (2020a), Definition 2.1.

2.2 Definition of the null space direction ξ_J

In what follows, $||\cdot||$ denotes the standard Euclidean norm. For a given design $x \in \mathcal{X}$, we consider the set $\widetilde{I}(x)$ of inequality constraints that are active or violated, and we denote respectively by $\widetilde{q}(x)$ and $\boldsymbol{h}_{\widetilde{I}(x)}$ their number and the associated vectors of constraints:

$$\widetilde{I}(x) = \{i\,|h_i(x) \geq 0\}, \tag{4}$$

$$\widetilde{q}(x) := \mathrm{Card}(\widetilde{I}(x)), \tag{5}$$

$$\boldsymbol{h}_{\widetilde{I}(x)} := (h_i(x))_{i\in\widetilde{I}(x)}. \tag{6}$$

The *null space direction* $\xi_J(x(t))$ is defined by the formula

$$\xi_J(x) := \nabla J(x) + \mathrm{D}\boldsymbol{g}(x)^T\boldsymbol{\lambda}^*(x) + \mathrm{D}\boldsymbol{h}_{\widetilde{I}(x)}(x)^T\boldsymbol{\mu}^*(x), \tag{7}$$

where $(\boldsymbol{\lambda}^*(x), \boldsymbol{\mu}^*(x)) \in \mathbb{R}^p \times \mathbb{R}_+^{\widetilde{q}(x)}$ are optimal Lagrange multipliers obtained by solving the "dual" problem

$$\min_{(\boldsymbol{\lambda},\boldsymbol{\mu})\in\mathbb{R}^p\times\mathbb{R}_+^{\widetilde{q}(x)}} ||\nabla J(x) + \mathrm{D}\boldsymbol{g}(x)^T\boldsymbol{\lambda} + \mathrm{D}\boldsymbol{h}_{\widetilde{I}(x)}(x)^T\boldsymbol{\mu}||, \tag{8}$$

In practice, (8) is a nonnegative least-squares problem which can be solved by using any standard quadratic programming solver; the current Python implementation of the Null Space Optimizer allows to choose between the CVXOPT (Vandenberghe (2010)) and the OSQP (Stellato et al (2020)) solvers.

Remark 3. The problem (8) has a clear and intuitive interpretation. Either its minimum value is zero and the KKT optimality conditions (3) are satisfied; or it is not zero and the vector $\xi_J(x)$ obtained from (7) with the dual variables $(\boldsymbol{\lambda}^*(x), \boldsymbol{\mu}^*(x))$ is the "best" locally feasible descent direction. More precisely, $\xi^*(x) := -\xi_J(x)/||\xi_J(x)||$ minimizes the linearized objective function while not worsening the first order variation of the constraints among all possible unit directions:

$$-\frac{\xi_J(x)}{||\xi_J(x)||} = \arg\min_{\xi\in\mathbb{R}^n} \quad \mathrm{D}J(x)\cdot\xi \qquad s.t. \begin{cases} \mathrm{D}\boldsymbol{g}(x)\cdot\xi = 0, \\ \mathrm{D}\boldsymbol{h}_{\widetilde{I}(x)}(x)\cdot\xi \leq 0, \\ ||\xi_J(x)|| \leq 1. \end{cases} \tag{9}$$

This alternative is the generalization to the well-known fact that in the unconstrained case, $\nabla J(x)$ is either zero at a local minimum or $-\nabla J(x)/||\nabla J(x)||$ is the best unit descent direction. The problem (8) is called "dual" because it is the dual of (9) (Nocedal and Wright (2006)).

2.3 Definition of the range space direction ξ_C

The *range space direction* $\xi_C(x(t))$ is a Gauss-Newton direction for reducing the violation of the constraints, thus gradually correcting unfeasible initializations. It is mathematically defined as

$$\xi_C(x) := \mathrm{D}C_{\widetilde{I}(x)}^T(\mathrm{D}C_{\widetilde{I}(x)}\mathrm{D}C_{\widetilde{I}(x)}^T)^{-1}C_{\widetilde{I}(x)} \quad (10)$$

where $C_{\widetilde{I}(x)} := \begin{bmatrix} \boldsymbol{g}(x) \\ \boldsymbol{h}_{\widetilde{I}(x)} \end{bmatrix}$ is the column vector gathering the equality and active inequality constraints. The definition (10) ensures that $\mathrm{D}C_{\widetilde{I}(x)}\xi_C(x) = C_{\widetilde{I}(x)}$. By using the properties $\mathrm{D}\boldsymbol{g}(x)\cdot\xi_J(x) = 0$ and $\mathrm{D}h_{\widetilde{I}(x)}\cdot\xi_J(x) \leq 0$, it is found that the violation of the constraints decreases exponentially as

$$\boldsymbol{g}(x(t)) = e^{-\alpha_C t}\boldsymbol{g}(x(0)),$$
$$\max(\boldsymbol{h}(x(t)), 0) \leq \max(\boldsymbol{h}(x(0)), 0)e^{-\alpha_C t},$$

see Feppon et al (2020a).

Remark 4. Formula (10) assumes $\mathrm{D}C_{\widetilde{I}(x)}$ to be full rank for the matrix $\mathrm{D}C_{\widetilde{I}(x)}\mathrm{D}C_{\widetilde{I}(x)}^T$ to be invertible. In order to address the issue of rank degeneracy, the Python implementation of the Null Space Optimizer uses the SciPy routine `lsqr` to compute the right pseudo inverse of $\mathrm{D}C_{\widetilde{I}(x)}$ in place of $\mathrm{D}C_{\widetilde{I}(x)}^T(\mathrm{D}C_{\widetilde{I}(x)}\mathrm{D}C_{\widetilde{I}(x)}^T)^{-1}$.

2.4 Treatment of constraints with sparse Jacobian and bound constraints

A naive implementation of the quadratic norm in (8) requires to compute the large matrix $\mathrm{D}C_{\widetilde{I}(x)}\mathrm{D}C_{\widetilde{I}(x)}^T$. This matrix needs also to be inverted in the formula (10). When the total number $p + \widetilde{q}(x)$ of active or violated inequality constraints is large, performing these operations explicitly may be computationally intractable. However, it is possible to reformulate (8) and (10) in terms of matrix-vector products that lend themselves to iterative methods when the constraints have a sparse Jacobian matrix. The necessary adaptations are detailed in Appendix C.

Bound constraints are a special instance of constraints with sparse Jacobian matrix. With the upgrade of Appendix C, the Null Space Optimizer treats them as any other constraints, by simply including their Jacobian matrix in the assembly of $\mathrm{D}C_{\widetilde{I}(x)}$. For instance, bound constraints of the form $l_i \leq x_i \leq u_i$ with $1 \leq i \leq n$ can be reformulated into $2n$ inequality constraints $l_i - x_i \leq 0$ and $x_i - u_i \leq 0$. The associated Jacobian is the block matrix

$$\mathrm{D}C = \begin{bmatrix} I \\ -I \end{bmatrix},$$

where I is the $n \times n$ identity matrix. For the ease of the user, the Nullspace Optimizer Python package provides a decorator `@bound_constraints_optimizable` for automating this assembly which is described in more details in Section 3.2.4. Additionally, the decorator ensures that the design variable is projected to the admissible set at every iteration by using the min/max operator

$$x \mapsto \min(u_i, \max(l_i, x)). \quad (11)$$

This operation ensures a more conservative enforcement of the bound constraints.

2.5 Implementation aspects and variations of the algorithm

The gradient flow (2) is discretized by using a Forward Euler scheme with time step Δt. The Python implementation of the Null Space Optimizer introduces further adaptations for improved numerical stability and usage flexibility.

Automatic rescaling of α_J and α_C

The parameters $\alpha_J \equiv \alpha_J^n$ and $\alpha_C \equiv \alpha_C^n$ are automatically scaled at every iteration n in order to ensure

$$||\alpha_J^n \xi_J(x^n)||_\infty \Delta t \leq A_J \Delta t, \quad (12)$$
$$||\alpha_C^n \xi_C(x^n)||_\infty \Delta t \leq A_C \Delta t, \quad (13)$$

where x^n is the iterate after n time steps, A_J and A_C are two fixed constants set by the user and $||\cdot||_\infty$ is the supremum or L^∞-norm (thus not the Euclidean norm $||\cdot||$ of Section 2.2). Furthermore, the parameter α_J^n is chosen such that (12) holds as an equality during for the first N_{it} iterations:

$$||\alpha_J^n \xi_J(x^n)||_\infty \Delta t = A_J \Delta t, \text{ for all } 0 \leq n \leq N_{\mathrm{it}}. \quad (14)$$

The number N_{it} is fixed by the user (see the correspondence table at the end of this subsection). The equality is not enforced afterwards in order to let $\xi_J(x_n) \to 0$ as x_n gets closer to a local minimum.

These rescalings are quite convenient since A_J and A_C can be understood as fractions of the time step. They have therefore a clear intuitive meaning for the user who does not need to worry about the magnitudes of $\xi_J(x_n)$ and $\xi_C(x_n)$.

Stabilization near active inequality constraints

Eq. (2) is an ODE with discontinuous right-hand side, which may introduce numerical instabilities near the boundaries of inequality constraints. When a novel constraint becomes active in the set $\widetilde{I}(x)$, the gradient of the objective function is suddenly projected on a smaller set when crossing the constraint barrier, resulting in a discontinuous change in the orientation vector of the trajectory. These instabilities can be addressed by enlarging the set $\widetilde{I}(x)$ to include inequality constraints $\{0 \leq j \leq q \,|\, h_j(x) \leq \epsilon_j\}$ that are not yet but likely to become active when computing the null space direction $\xi_C(x)$ with (7) and (8). The parameter ϵ_j are small threshold values that are automatically computed to detect constraints located at a distance less than $K\Delta t$, where K is a parameter set by default to 0.1 (which is working well in practice independently of the optimization problem). Then, the set $\widetilde{I}(x)$ considered in (10) for computing the range step $\xi_C(x)$ is also enlarged with constraints that are likely to remain active based on the information given by the optimal multiplier $\boldsymbol{\mu}^*$. We refer the reader to Feppon et al (2020a) for the detailed implementation of these rules.

Weights for the violated constraints

The direction $\xi_C(x)$ in (10) is designed to ensure that the violation of the constraints decays as $e^{-\alpha_C t}$. In practice, the user may desire different decay rates for each of the constraints for a more flexible control of the optimization path. This is possible by defining $\xi_C(x)$ as

$$\begin{aligned}\xi_C(x) := &\mathrm{D}C_{\widetilde{I}(x)}^T(\mathrm{D}C_{\widetilde{I}(x)}\mathrm{D}C_{\widetilde{I}(x)}^T)^{-1}\\ &\times \mathrm{diag}(\alpha_1,\dots,\alpha_p,\alpha_{p+1},\dots,\alpha_{p+q})C_{\widetilde{I}(x)} \end{aligned} \quad (15)$$

where $(\alpha_i)_{1\leq i\leq p+q}$ are $p+q$ weights ensuring that the violation of the corresponding equality or inequality constraint i decreases faster than $e^{-\alpha_i\alpha_C t}$.

Correspondence table for the algorithm parameters

The Null Space Optimizer accepts various algorithm parameters in the form of a Python dictionary provided by the optimization routine (see e.g. Section 3.3 below). The following table provides the correspondence between the mathematical notation of the parameters mentioned in this section and their name (as dictionary keys) in the Python implementation.

Parameter	Implementation	Default
Δt	`dt`	0.1
A_J	`alphaJ`	1
A_C	`alphaC`	1
N_{it}	`itnormalisation`	1
K	`K`	0.1
$(\alpha_i)_{1\leq i\leq p+q}$	`alphas`	$(1)_{1\leq i\leq p+q}$

Note that these algorithm parameters are not tuning parameters, they are rather used to provide an increase level of flexibility allowing for faster convergence towards an optimum. They have a clear physical meaning and many practical situations do not require a custom value. In the numerical examples considered in this paper, we leave the parameters to the default values unless otherwise specified.

3 Topology Optimization of the 88 lines Matlab MBB beam example with the Null Space Optimizer

This section and the next to follows provide three tutorials giving an in-depth view on the use of the Null Space Optimizer for density based Topology Optimization. In this section, we revisit the classical MBB beam example from the popular 99 lines Matlab Topology Optimization code originally introduced in Sigmund (2001), which was refactored in a more efficient 88 lines version in Andreassen et al (2011).

We provide a step-by-step tutorial for writing an elegant implementation of this classical example with the Null Space optimizer, without bringing substantial modifications to the state

solver and the sensitivities. The proposed implementation is claimed "elegant" because it uses elegant paradigms of the Python language such as object-oriented programming and decorators to isolate the various building blocks of the Topology Optimization code. As the following examples demonstrate, we obtain as such succinct and flexible implementations that closely matches the mathematical formulation (1) of the optimization problem, sets a clear separation between the implementation of the state solver, the filtering of the design variable and the optimization solver. This programming paradigm is easily reproducible to more general situations.

This section is organized as follows. The physical setting of the MBB beam test case and the formulation of the associated design optimization problem is given in Section 3.1. Section 3.2 details how to implement this test case in Python and to solve it with the Null Space Optimizer. Numerical results and comparisons with the Optimality Criteria, MMA and IPOPT solvers are presented in Section 3.4 for two different benchmark situations: one with a rather large target volume fraction corresponding to the setting of the 88 line Matlab code paper of Andreassen et al (2011), and a second example imposing a tiny target volume fraction that is more challenging.

3.1 The classical MBB beam problem of the 88 lines Matlab Topology Optimization code.

The classical MBB beam problem setting as introduced in Sigmund (2001); Andreassen et al (2011) is depicted on Fig. 1. The design domain is a rectangle discretized with $n := \texttt{nelx} \times \texttt{nely}$ square elements. Each element $1 \leq e \leq \texttt{nelx} \times \texttt{nely}$ is characterized by a Young modulus $E_e(x_e)$ that depends on the local fraction of material x_e according to the SIMP interpolation law:

$$E_e(x_e) := \texttt{Emin} + x_e^p(\texttt{Emax} - \texttt{Emin}).$$

Throughout the paper, the SIMP exponent is set to $p := 3$, $\texttt{Emax} := 1$ is the Young modulus of the solid material and $\texttt{Emin} := 10^{-9}$ is the Young modulus of an ersatz material representing void. The left-side boundary is subject to no horizontal displacement and the bottom right corner to no vertical displacement. A unit downward force $\boldsymbol{g}$ is applied at the top left corner.

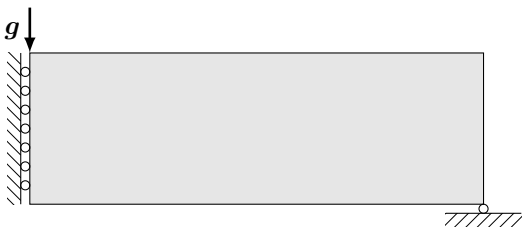


Fig. 1: Setting of the MBB beam Topology Optimization problem of Section 3.

The structural design optimization problem for minimizing the compliance of the structure subject to a single volume equality constraint reads:

$$\begin{aligned} \min_{\boldsymbol{x}=(x_e)_{1\leq e\leq n}} \quad & J(\boldsymbol{x}) = \boldsymbol{U}^T \boldsymbol{K}(\boldsymbol{x}) \boldsymbol{U} \\ & = \sum_{e=1}^{n} E_e(x_e) \boldsymbol{u}_e^T \texttt{KE} \boldsymbol{u}_e \\ s.t. \quad & \begin{cases} V(\boldsymbol{x})/V_0 := \dfrac{1}{n}\displaystyle\sum_{e=1}^{n} x_e = \texttt{volfrac}, \\ \boldsymbol{K}(\boldsymbol{x})\boldsymbol{U} = \boldsymbol{F}, \\ 0 \leq x_e \leq 1, \text{ for all } 1 \leq e \leq n, \end{cases} \end{aligned} \tag{16}$$

where $\boldsymbol{K}(\boldsymbol{x})$ is the global stiffness matrix, $\boldsymbol{U}$ and $\boldsymbol{F}$ are the global displacement and force vectors, $\boldsymbol{u}_e$ is the displacement vector in the element e and $\texttt{KE}$ is the element stiffness matrix for an element with unit Young's modulus. $V(\boldsymbol{x})$ refers to the volume of the optimized design, V_0 is the total volume of the design domain and $\texttt{volfrac}$ is the target volume fraction.

A classical issue in density based TO when solving the minimization problem (16) is the arising of checkerboard patterns in the design variable (Sigmund and Petersson (1998)). In order to eliminate these nonphysical solutions from the design space, the original 88 lines Matlab Topology Optimization code implements a density filter on the optimization variable. More precisely, the final

optimization problem considered reads

$$\min_{\boldsymbol{x}=(x_e)_{1\leq e\leq n}} \quad J(\widetilde{\boldsymbol{x}}) \qquad s.t. \begin{cases} V(\widetilde{\boldsymbol{x}})/V_0 = \texttt{volfrac}, \\ \boldsymbol{K}(\widetilde{\boldsymbol{x}})\boldsymbol{U} = \boldsymbol{F}, \\ 0 \leq x_e \leq 1 \text{ for all } 1 \leq e \leq n, \end{cases} \tag{17}$$

where the filtered density $\widetilde{\boldsymbol{x}}$ is a local averaging of elements that are neighbors by a distance smaller than a prescribed filter radius $r_{\min}$:

$$\widetilde{x}_e := \frac{1}{\sum_{i=1}^n H_{ei}} \sum_{i=1}^n H_{ei} x_i \text{ for all } 1 \leq e \leq n, \tag{18}$$

with $H_{ei} = \max(0, r_{\min} - \Delta(e,i))$, where $\Delta(e,i)$ is the distance between the centers of elements e and i.

3.2 Python implementation of the MBB beam problem with the Null Space Optimizer

In order to closely match the original Matlab implementation of this test case, we reshape a Python translation of the 88 lines code of Andreassen et al (2011) that is provided by Aage and Johansen (2016). In the subsequent subsections, we provide a comprehensive description of the implementation using the Null Space Optimizer. The full source code is provided in Appendix A and we refer to the associated line numbering throughout the discussion.

The code can be comprehended in the following parts:

- lines 1–9: Python modules imports;
- lines 11–109: initializations of the element stiffness and density filter matrices;
- lines 111–142: declaration of the state solver;
- lines 144–152: declaration of the filtering functions;
- lines 154–191: definition of the optimization problem
- lines 199–203: setting of optimization parameters and running the optimization.

3.2.1 Declaration of the optimization problem

The overall code is best understood by looking first at the declaration of the class `TO_problem` in lines 154–191. The class `TO_problem` inherits the `EuclideanOptimizable` class from the Null Space Optimizer Python package, which indicates to the optimization routine that the design domain is a finite dimensional design space $\mathbb{R}^n$ (represented by NumPy arrays).

The declaration of this class closely follows the mathematical definition of the Topology Optimization problem (17) by implementing the following methods (required by the Null Space Optimizer):

- `x0` returns the initial design;
- `J` returns the objective function computed on the design variable `x`;
- `G` returns the array of equality constraints, here the single volume constraint $V(\widetilde{x})/V_0 -$ `volfrac`;
- `dJ` returns the sensitivity of the objective function (a row vector);
- `dG` returns the Jacobian matrix of the equality constraints (here a row vector);

Arbitrary inequality constraints could be added similarly by implementing methods `H` and `dH`. Additionally, the class implements a function `accept` for initializing and updating plots of the design. The function `accept` is called at the beginning of every iteration by the Null Space Optimizer, it takes as inputs a dictionary `params` containing the optimization algorithm parameters and a dictionary `results` storing the optimization histories. The optimized design from the current iteration is accessed as `results['x'][-1]`.

The implementation of the Topology Optimization problem relies then on three key ingredients:

- `solve_state` (lines 111-142): a function solving the linear elasticity problem and returning the compliance `obj` and its sensitivity `dc`,
- `@filtered_optimizable` (line 156): a decorator which implements the filtering of the design variable,
- `@bound_constraints_optimizable` (line 155): a decorator which supplements the optimization problem with the bound constraints $0 \leq x_e \leq 1$ for all $1 \leq e \leq n$.

These three ingredients are described in the next Sections 3.2.2 to 3.2.4.

3.2.2 Computations of the state solution and the sensitivities

The computation of the compliance and its sensitivity is achieved by the function `solve_state` at lines 111–142. The function depends on a preliminary part of the code (lines 11–109) which precomputes the element stiffness matrix `KE`, the index pointers `iK`, `jK` as well as the element stiffness matrix `edofMat` describing the degrees of freedom attached to every element. The array `dofs` references the finite element degrees of freedom attached to every vertex. The total number of degrees of freedom is stored in the variable `ndof`, the degrees of freedom associated to the Dirichlet and the free boundary are stored in the arrays named `fixed` and `free`, and the downward unit load is stored in the finite element vector `f` (which implements the right-hand side $\boldsymbol{F}$ of the linear elasticity system).

Then, the function `solve_state` takes as an input an array `x` representing the density variables $\boldsymbol{x} = (x_e)_{1\leq e\leq n}$. It internally assembles the finite element matrix and solves the linear system $\boldsymbol{K}(\boldsymbol{x})\boldsymbol{U} = \boldsymbol{F}$ by applying a sparse Cholesky factorization provided by the CVXOPT package. This part follows almost textually the Python translation of the 88-lines Matlab code of Aage and Johansen (2016) and the reader is referred to Andreassen et al (2011) for more details.

However, a few minor useful changes have been implemented. First, the function `solve_state` allows the force vector `f` to be a matrix with N_{loads} columns corresponding to different loads. This parameter is useful for the multiple load case of Section 4 below. In the present situation, `f` is a single column vector. The function returns an array `obj` for multiple loads $N_{\text{loads}} > 1$, and the real number `obj`[0] when $N_{\text{loads}} = 1$ (the present case).

Second, the NumPy function `np.einsum` is used at line 131 for efficient and vectorized computation of the terms $\boldsymbol{u}_e^T \texttt{KE} \boldsymbol{u}_e$ in (16).

Third, the definition of the function `solve_state` is preceded by the decorator `@memoize()` provided by the Null Space Optimizer package. This decorator caches the function's results, effectively returning a stored value if the computationally intensive `solve_state` function is invoked multiple times with the same argument. This feature is particularly advantageous in the declaration of the optimization problem which requires to separation of the objective function and the sensitivities into distinct functions, resulting in separate calls at lines 162 and 166: this syntax ensures that the linear elasticity system is solved only one time per iteration. The decorator can be understood as equivalent to appending the following lines of codes after the definition of `solve_state`:

```
old_solve_state = solve_state
solve_state = \
    memoize()(old_solve_state)
```

The function `memoize()` relies on a hashing algorithm that determines whether the function `solve_state` has already been called on the argument `x`, and in that case, returns a stored value.

3.2.3 Filtering of the density variable

In the proposed implementation, the density filter $\boldsymbol{x} \mapsto \widetilde{\boldsymbol{x}}$ of (18) is conveniently implemented at line 156 by the decorator

```
@filtered_optimizable(filter,
   diff_filter)
```

Its arguments `filter` and `diff_filter` respectively implement the filtering function (18) and its sensitivity.

The decorator `@filtered_optimizable` converts the problem (16) into the filtered problem (17). It automatically creates a copy of the class `TO_problem`, substituting the methods `J(self,x)` and `G(self,x)` with `J(self,filter(x))` and `G(self,filter(x))`.

The filtering function `filter` is implemented at lines 144–152 with the following lines of code:

```
@memoize()
def filter(x):
    return Hs @ (H @ x)
```

where `H` is the matrix (H_{ei}) of (18) and `Hs` is the sum of its elements; `H` and `Hs` are precomputed at lines 80–105 reproducing the implementation of Aage and Johansen (2016); Andreassen et al (2011). Memoization is used once again for efficiency, which allows for the proposed implementation paradigm without the need to worry about redundant calls to the `filter` function.

To obtain fully consistent sensitivities, the derivatives need to be updated according to the chain rule

$$\frac{\mathrm{d}}{\mathrm{dx}}[\mathtt{J}(\widetilde{\mathtt{x}})] \cdot \mathtt{dx} = \mathtt{dJ}(\widetilde{\mathtt{x}}) \cdot \frac{\mathrm{d}\widetilde{\mathtt{x}}}{\mathrm{dx}} \cdot \mathtt{dx}.$$

The decorator `@filtered_optimizable` automatically applies this transformation to the methods `dJ` and `dG` by exploiting the second argument `diff_filter`, which is required to be a Python function implementing the left vector-matrix product:

$$(\mathtt{x}, \mathtt{v}) \mapsto \mathtt{v} \cdot \frac{\mathrm{d}\widetilde{\mathtt{x}}}{\mathrm{dx}}, \tag{19}$$

where `v` is an input row vector (or a $p \times n$ matrix if G is a function returning p constraints) that is to be replaced with $\mathtt{dJ}(\widetilde{\mathtt{x}})$ or $\mathtt{dG}(\widetilde{\mathtt{x}})$ by the optimization routine. In the present case, the filter is linear which makes the implementation of `diff_filter` straightforward (lines 143–146):

```
@memoize()
def diff_filter(x,v):
    return (H @ (Hs @ v.T)).T
```

3.2.4 Bound constraints

The $2n$ bound constraints $0 \leq x_e \leq 1$ for $1 \leq e \leq n$ are implemented with the decorator `@bound_constraints_optimizable(l=0,u=1)` at line 155. This decorator returns a copy of the filtered class `TO_problem`, by:

(i) appending a method `H(self,x)` implementing the $2n$ bound constraints

$$\begin{cases} -x_e \leq 0, \\ x_e \leq 1, \end{cases} \quad \text{for } 1 \leq e \leq n.$$

This operation is equivalent to adding a method `H(self,x)` in the definition of the class `TO_problem`[4]:

```
def H(self,x):
    # Returns the vector
    # of constraints
    # [x_0-1,x_1-1,...,x_n-1,
    #  -x_0,-x_1,...,-x_n]
    return np.hstack((x-1,-x))
```

The use of the decorator allows to simplify this process for the user's convenience;

(ii) appending a method `dH(self,x)` returning the constraints sensitivities, that is the block diagonal matrix

$$\mathtt{dH}(\mathtt{x}) = \begin{bmatrix} I \\ -I \end{bmatrix}$$

where I is the $n \times n$ identity matrix, This function returns this Jacobian matrix in a sparse format. This operation is equivalent to adding the following method in the class `TO_problem`:

```
def dH(self,x):
    I = sp.eye(n,format="csc")
    return sp.vstack((I,-I))
```

(iii) adding a method `retract(self,x,dx)` implementing the projection rule (11) that is applied after every update by the Null Space Optimizer. We refer to Absil and Malick (2012) and to the documentation of the package regarding the definition of retractions and their use in optimization.

Notice that unlike MMA or IPOPT, the Null Space optimizer treats bound constraints almost as any other inequality constraints. The only difference in this treatment with other constraints is the use of the projection rule (11) that enables a stricter enforcement than the range space step ξ_C.

3.3 Running the optimizer

The optimization problem is solved with the Null Space Optimizer at lines 193–203. Line 200 initializes finite element matrices by calling the function `init()`. The class `TO_problem` is instantiated at line 195. Then, the optimization problem is solved at line 201 by the routine `nlspace_solve` by discretizing the null space gradient flow (2). The arguments of the routine are an instance of the optimization problem and the dictionary of algorithm parameters `optimization_params`. Here, we specify a time step `dt` = 0.3. The parameter `itnormalisation`=50 ensures that the null space step $\alpha_J \xi_J(x)$ is rescaled to unity during the first 50 iterations, before being allowed to

[4] Note that `H(self,x)` is a method of `TO_problem` which is not to be confused with the filtering matrix also denoted by `H` (a global variable). We use `H` for the filtering matrix to match the notation of Andreassen et al (2011).

smoothly decrease (see Section 2.5). The parameter `maxit` stops the optimization after 150 iterations. The parameters `save_only_N_iterations` and `save_only_Q_constraints` are used for memory efficiency: by default, the Null Space Optimizer stores the values of all constraints and designs of all iterations in the output dictionary `results`. In the present case, we require that only the last iteration and the first 5 constraints (which include the volume constraints and the first 4 bound constraints) are stored.

3.4 Numerical results and comparisons with OC, MMA and IPOPT

We solve the MBB Topology Optimization problem with the Null Space optimizer and we compare the convergence to that of the Optimality Criteria method, MMA and IPOPT. We consider two situations: first, the exact configuration of the MBB problem considered in the 88 lines Matlab code (Section 3.4.1), then a more challenging version where the volume fraction target is set to a very small value (Section 3.4.2).

3.4.1 MBB beam test case with medium volume fraction

In this first situation corresponding to the setting of Andreassen et al (2011), the volume fraction target is set to `volfrac`:=0.4 and the filter radius to $r_{\min} := 5.4$. The number of elements in the x and y directions are set to `nelx` := 180 and `nely` := 60. The Null Space optimizer time step `dt`, the move limit m_{OC} of the Optimality Criteria method and the move limit m_{MMA} of MMA are set to the same value $\texttt{dt} = m_{\text{OC}} = m_{\text{MMA}} = 0.3$. The number of normalisation iterations for the null space direction ξ_J is set to `itnormalisation`=50. We keep the default settings for IPOPT. The optimization is stopped for every algorithm after 150 iterations.

The final objective and constraint values are reported in Table 1. Convergence histories and intermediate designs at identical iterations are displayed respectively on Figs. 2 and 4. A close-up on the convergence histories for the first 30 iterations is shown on Fig. 3.

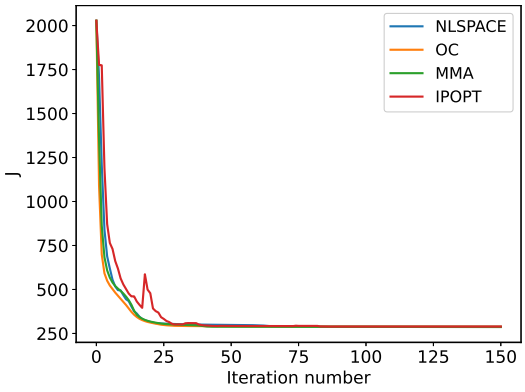
On this basic example, all the algorithms obtain the same final design with very similar performances. The Null Space Optimizer (abbreviated NLSPACE) performs similarly well as its competitors, the Optimality Criteria method, MMA and IPOPT. With the chosen settings, NLSPACE manages to enforce the volume constraint at machine precision. The performance of the design found by MMA is slightly better, while the performance of IPOPT and OC are almost identical to that of the Null Space Optimizer.

IPOPT takes slightly more iterations to converge than the other algorithms and exhibits a rather surprising behavior around iteration 20, it still manages to drive the design to the optimum after a few more iterations. This analysis, however, must be tempered by the fact that we didn't tune IPOPT parameters to find the most efficient convergence setting. It can also be observed that MMA takes a rather large deviation to the volume constraint during the first 20 iterations, while OC maintains it close to zero throughout the iterations. The Null Space Optimizer as well as IPOPT only slightly violate the constraint during the first iterations before correcting it after approximately 20 iterations.

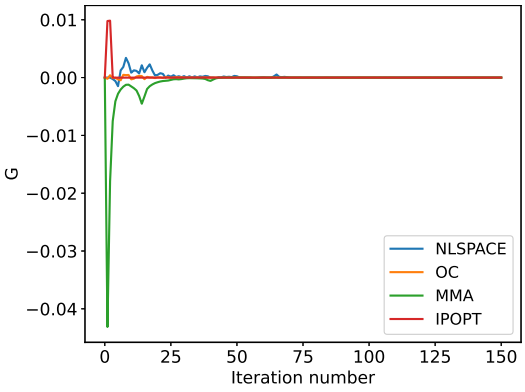
Looking at the first iterations on the convergence histories of Fig. 3, we observe that the Optimality Criteria method, MMA and the Null Space Optimizer have similar initial decays, with a small advantage for OC and MMA. This effect is also visible on the other numerical examples considered in the next sections. This small advantage may be surprising because we used identical `move` and time step `dt` parameters while the definition (9) of the null space step ξ_J should ensure that it produces the best locally feasible descent direction, hence theoretically leading to best update among all possible directions with identical norm. This apparent paradox is explained by the fact that the norm $||\cdot||$ involved in the definition (9) of $\xi_J(x)$ is a quadratic norm, while the implementation normalizes the update steps with the supremum (L^∞) norm $||\cdot||_\infty$ (see Section 2.5). Here, MMA and OC use descent directions that are more optimal than ξ_J in the L^∞-norm, but we checked that these descent directions are of strictly larger L^2-norm than ξ_J. Still, this advantage remains negligible on this test case as NLSPACE catches up MMA and OC after about 7 iterations.

Algorithm	J	$G = V(\boldsymbol{x})/V_0 - \texttt{volfrac}$
NLSPACE	288.702	0
OC	288.830	$-1.30 \cdot 10^{-5}$
MMA	286.383	$-2.20 \cdot 10^{-6}$
IPOPT	288.801	$-9.30 \cdot 10^{-8}$

Table 1: Final objective and constraint values for the MBB example for the different optimization algorithms.



(a) Objective function J.

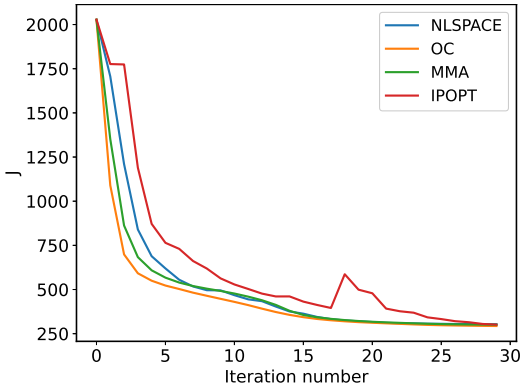


(b) Volume constraint $G(\boldsymbol{x}) = V(\boldsymbol{x})/V_0 - \texttt{volfrac}$.

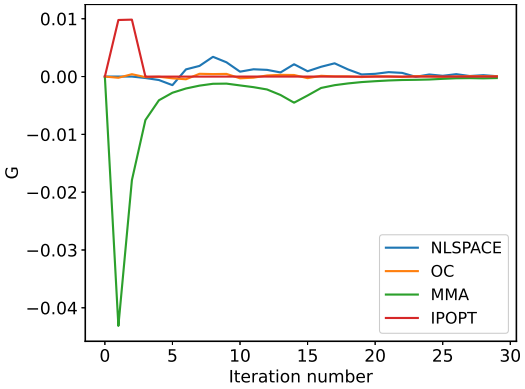
Fig. 2: Convergence history of the different optimizers on the MBB beam example of Section 3.4.1.

3.4.2 MBB beam test case with a tiny volume fraction

We now investigate a more challenging example where the volume fraction is set to the small value $\texttt{volfrac} := 0.01$. For this test case, the resolution parameters of the computational mesh are set to $\texttt{nelx}:=150$ and $\texttt{nely}:=50$, and the filter radius to $r_{\min} := 1.9$. As the numerical results



(a) Objective function J.

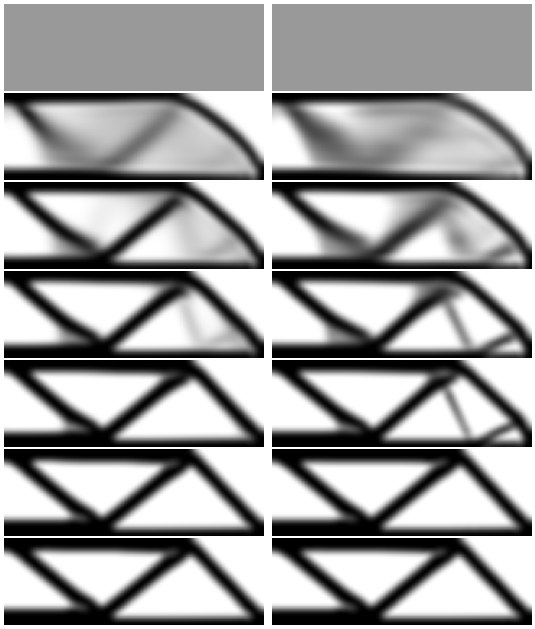


(b) Volume constraint $G(\boldsymbol{x}) = V(\boldsymbol{x})/V_0 - \texttt{volfrac}$.

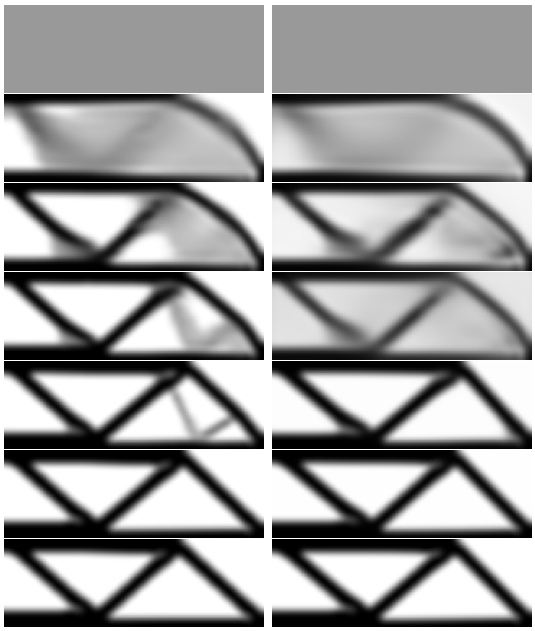
Fig. 3: Convergence history of the different optimizers on the MBB beam example of Section 3.4.1 (zoom on the first 30 iterations).

demonstrate below, the test case is somehow unrealistic since a completely black and white design cannot be achieved for such a low amount of material given the resolution. Still, the test case is an interesting challenging benchmark for testing the performance of optimization methods.

For such a small value of the volume fraction target, the classical implementation of MMA based on the Matlab code released by Svanberg (2014) fails both at finding an optimum value and ensuring the correct volume fraction, see Fig. 5 and Table 2. Rescaling the objective function is not sufficient for ensuring convergence towards a satisfactory design. The issue comes from what seems to us a limitation of the heuristic update rules of the moving asymptotes. Indeed, a suitable convergence can be obtained by the following



(a) Optimality Criteria (b) Null Space



(c) MMA (d) IPOPT

Fig. 4: Comparison of the designs obtained by the different optimizers for the MBB beam test case of Section 3.4.1 at iterations 0, 10, 15, 20, 30, 80 and 150.

three modifications to the standard implementation described in Svanberg (2014):

- using a different initialization for the asymptotes. The default initialization of the asymptotes reads (eq. (3.11) in Svanberg (2014)):

$$\begin{cases} l_e^{(k)} = x_e^{(k)} - 0.5(x_e^{\max} - x_e^{\min}), \\ u_e^{(k)} = x_e^{(k)} + 0.5(x_e^{\max} - x_e^{\min}), \end{cases}$$

  for $k = 0, 1$. Here, $x_e^{(k)}$ is the e-th coordinate of the design variable at iteration k, $x_e^{\max}$ and $x_e^{\min}$ are the lower and upper bound limits and $l_e^{(k)}$ and $u_e^{(k)}$ denote respectively the lower and upper asymptotes. A better convergence is obtained by setting instead:

$$\begin{cases} l_e^{(k)} = \max(x_e^{(k)} - 0.5(x_e^{\max} - x_e^{\min}), 0), \\ u_e^{(k)} = \max(x_e^{(k)} + 0.5(x_e^{\max} - x_e^{\min}), 10), \end{cases}$$

  for $k = 0, 1$. This setting avoids that the lower asymptotes to become lower than 0, and it "pushes" the upper asymptotes to 10 making the initial approximation of the objective and constraint functions flatter;
- discarding the moving asymptote rule, which reads (eq. (3.12) in Svanberg (2014)):

$$\begin{cases} l_e^{(k)} = x_e^{(k)} - \gamma_e^{(k)}(x_e^{(k-1)} - l_e^{(k-1)}), \\ u_e^{(k)} = x_e^{(k)} + \gamma_e^{(k)}(u_e^{(k-1)} - x_e^{(k-1)}), \end{cases}$$

  where $\gamma_e^{(k)}$ is a number set to 0.7, 1.2 and 1 depending on the values of $x_e^{(k)}$, $x_e^{(k-1)}$ and $x_e^{(k-2)}$. In our "fixed" implementation of MMA, we manually disable this rule and we leave the remainder of the code provided by Deetman (2020) unchanged.
- using tighter moves between iterations. Our default implementation of MMA sets $x_e^{\min} = 0$ and $x_e^{\max} = 1$ for all $1 \leq e \leq n$. The fixed implementation forbids updates larger than the parameter `move` by setting

$$\begin{aligned} x_e^{\min} &= \max(0, x^{(k)} - \texttt{move}), \\ x_e^{\max} &= \min(1, x^{(k)} + \texttt{move}). \end{aligned}$$

  at every iteration.

Fig. 5: Convergence history of the standard implementation of MMA on the MBB beam example of Section 3.4.2 with volume fraction target `volfrac`=0.01 (iterations 0, 20, 30, 80, 100, 150).

We note that the use of the line-search provided by the Globally Convergent MMA (GCMMA) algorithm of Svanberg (2014) would also be an appropriate fix for this problem. Still, even GCMMA would require the smart initialization of the asymptotes to lead to satisfactory designs.

We run the different optimizers with `dt`=0.02 for the Null Space Optimizer, `move`=0.2 for MMA and `move`=0.05 for the OC solver. These move parameters yield satisfactory results for MMA and OC. The NLSPACE time step is the sole parameter to tune, it needs to be taken sufficiently small to accommodate the small volume fraction and the initial design (a uniform gray design $\boldsymbol{x}^0 = (x_e^0)$ with $x_e^0 = 0.01/(\texttt{nelx} \times \texttt{nely})$ for $1 \leq e \leq n$) close to 0. IPOPT is executed with default options. The numerical values of the objective function and volume constraint of the final designs are listed in Table 2. Convergence histories are shown on Fig. 6 as well as a close-up on the first 30 iterations on Fig. 7. Figure 8 displays a sequence of intermediate designs for each optimizer.

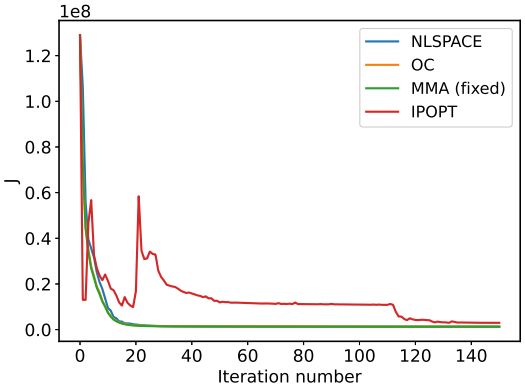
As it is visible on Fig. 8, the target volume fraction could not be achieved with completely black and white designs, leading "gray" solid bars having a density that do not reach the maximum value $x_e = 1$. The numerically computed optimized designs suggest that this benchmark example seems to admit several locally minimizing designs with different topologies achieving comparable performances. The values of Table 2 indicate that the fixed version of MMA happens to find a design with the best performance among the tested algorithms. The Optimality Criteria solver does not need any fix and is able to find a slightly different design with similar performances. However, this solver is less accurate regarding the enforcement of the volume constraint as illustrated on Fig. 6b. The Null Space algorithm converges to a different design with a performance very similar to the one found by the fixed version of MMA, albeit slightly worse and slightly better than OC. Noticeably, it enforces the volume constraint at machine precision at convergence. On this example, the IPOPT solver with default parameters converges to a design that satisfies the constraint, but that is much less performant than the solutions provided by the other successful algorithms.

Algorithm	J	$G = V(\boldsymbol{x})/V_0 - \texttt{volfrac}$
NLSPACE	$1.27371 \cdot 10^6$	0
OC	$1.28758 \cdot 10^6$	$2.10 \cdot 10^{-4}$
MMA	$3.72097 \cdot 10^6$	$-3.10 \cdot 10^{-5}$
MMA (fixed)	$1.25473 \cdot 10^6$	$-8.70 \cdot 10^{-9}$
IPOPT	$2.97208 \cdot 10^6$	$-3.40 \cdot 10^{-7}$

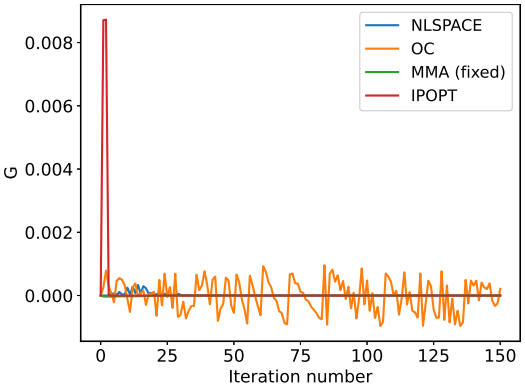
Table 2: Final objective and constraint values for the MBB example of Section 3.4.2 with tiny volume fraction target for the different optimization algorithms.

4 Implementing Robust Topology Optimization for multiple load cases

In this section, we describe the implementation of robust design optimization of a bridge subject to multiple loads using a min/max formulation. We demonstrate the suitability of the Null Space Optimizer for solving this problem featuring multiple inequality constraints, and we compare the numerical results to those of MMA and IPOPT (for this example, it is not possible to use the standard Optimality Criteria method which is adapted to optimization problems featuring only a single

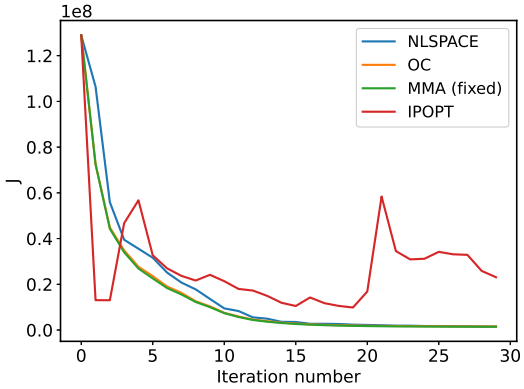


(a) Objective function J.

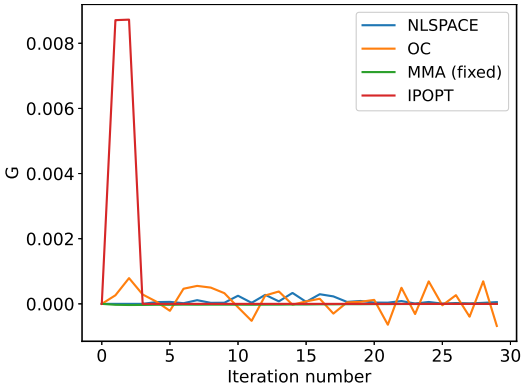


(b) Volume constraint $G(\boldsymbol{x}) = V(\boldsymbol{x})/V_0 -$ `volfrac`.

Fig. 6: Convergence history of the different optimizers on the MBB beam example of Section 3.4.2 with tiny volume fraction target.



(a) Objective function J.



(b) Volume constraint $G(\boldsymbol{x}) = V(\boldsymbol{x})/V_0 -$ `volfrac`.

Fig. 7: Convergence history of the different optimizers on the MBB beam example of Section 3.4.2 with tiny volume fraction target (close-up on the first 30 iterations).

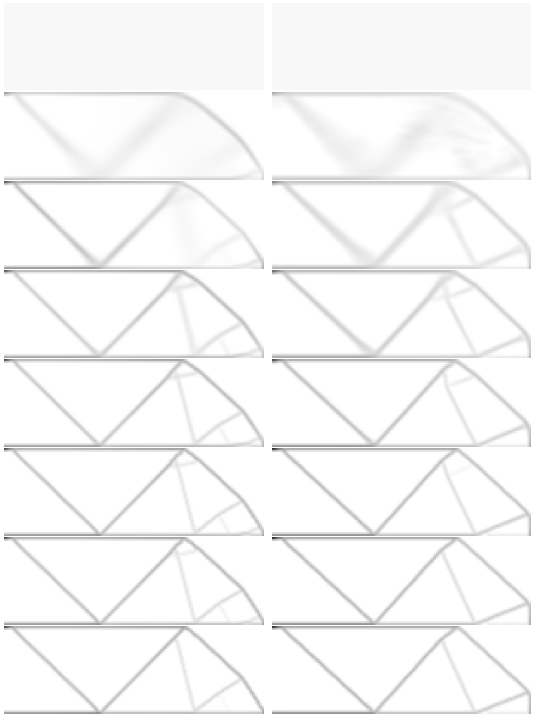
equality constraint). The Null Space Optimizer has already proven to be effective for multiple load case Topology Optimization with the level-set method: a similar bridge test case has been considered in Feppon et al (2020a). We find that the performance of the method is not altered by adding the bound constraints that are peculiar to density-based Topology Optimization.

The physical setting and the formulation of the optimization problem are outlined in Section 4.1. Section 4.2 describes the Python implementation of the multiple load case with the Null Space Optimizer, that is obtained after only a few modifications of the MBB beam test case of Section 3. A decorator `@minmax_optimizable` is provided by the Python package to conveniently implement the robust min/max formulation of the optimization problem. Finally, numerical results comparing the Null Space Optimizer, MMA and IPOPT are given in Section 4.3.

4.1 Setting of the bridge Topology Optimization problem with multiple loads

The bridge test case that we consider is identical to that of the MBB problem illustrated on Fig. 1, up to the following changes:

- the design domain features now `nelx` $\times$ `nely` elements with `nelx`=180 and `nely`=90;
- the bottom right corner is now fixed both in the horizontal vertical direction;



(a) Optimality Criteria (b) Null Space

(c) MMA (with fix) (d) IPOPT

Fig. 8: Intermediate designs obtained with the different optimizers at iterations 0, 10, 15, 20, 40 and 80 for the MBB beam example of Section 3.4.2 with tiny volume fraction target.

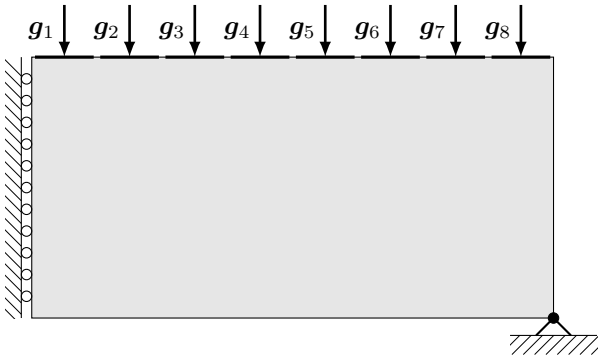


Fig. 9: Setting of the bridge design problem of Section 4.

- that eight different unit loads $(\boldsymbol{g}_i)_{1\leq i\leq 8}$ are independently applied on the top boundary.

Here, the optimized domain actually represents half of the final structure, assuming symmetry with respect to the y-axis; the whole structure is thus subject to 8 loads consisting of two vertical forces distributed symmetrically. Each load spreads continuously over one eighth of the top boundary with a magnitude set to $-1/\texttt{nelx}$. The setting of the multiple load case is illustrated on Fig. 9.

The goal is to minimize the maximum of the compliances of the structure subject to the 8 possible loads and a prescribed volume fraction of material. This robust design formulation corresponds mathematically to solving the min/max optimization problem

$$\min_{\boldsymbol{x}=(x_e)_{1\leq e\leq n}} \max_{1\leq i\leq 8} J_i(\widetilde{\boldsymbol{x}}) := \boldsymbol{U}_i^T \boldsymbol{K}(x) \boldsymbol{U}_i$$
$$s.t. \begin{cases} V(\widetilde{\boldsymbol{x}})/V_0 = \texttt{volfrac}, \\ \boldsymbol{K}(\widetilde{\boldsymbol{x}})\boldsymbol{U}_i = \boldsymbol{F}_i, \text{ for all } 1 \leq i \leq 8, \\ 0 \leq x_e \leq 1 \text{ for all } 1 \leq e \leq n, \end{cases} \quad (20)$$

where $\widetilde{\boldsymbol{x}}$ is the filtered density (18), $\boldsymbol{F}_i$ is a column vector corresponding to the finite element representation of the i-th load $\boldsymbol{g}_i$, and $\boldsymbol{U}_i$ is the finite element representation of the associated displacement vector.

In order to treat (20) without the need for differentiating the maximum function, we solve the classical equivalent formulation that introduces a

slack variable $m \in \mathbb{R}$:

$$\min_{\substack{\boldsymbol{x}=(x_e)_{1\le e\le n},\\ m\in\mathbb{R}}} \quad m$$
$$s.t. \begin{cases} J_i(\widetilde{\boldsymbol{x}}) - m \le 0, \text{ for all } 1 \le i \le 8, \\ V(\widetilde{\boldsymbol{x}})/V_0 = \texttt{volfrac}, \\ \boldsymbol{K}(\widetilde{\boldsymbol{x}})\boldsymbol{U}_i = \boldsymbol{F}_i, \text{ for all } 1 \le i \le 8, \\ 0 \le x_e \le 1 \text{ for all } 1 \le e \le n. \end{cases} \tag{21}$$

4.2 Python implementation with the Null Space Optimizer

Thanks to a few automations provided by the Null Space optimizer Python package, the implementation of the problem (21) requires only minor updates of the code of the MBB beam test case of Appendix A.

First, the variable `nely` can now be assigned by default to 90 by updating line 50:

```
nely = kwargs.get('nely', 90)
```

Then, the vector `fixed` containing the finite element degrees of freedom corresponding to zero Dirichlet boundary conditions needs to be updated to match the situation of Fig. 9. This is done by updating line 77 as follows:

```
fixed = np.union1d(dofs[0,:,0],
                   dofs[-1,-1,:])
```

Second, we assemble the matrix `f`$=\begin{bmatrix}\boldsymbol{F}_1 & \dots & \boldsymbol{F}_8\end{bmatrix}$ whose columns contain the finite element representation of the 8 different loads. This can be achieved by substituting lines 108–109 with the following code which determines the degrees of freedom at the top boundary corresponding to each load $\boldsymbol{g}_i$ and assigns them the force $-1/$`nelx`:

```
nforces = 8
n = np.ceil(nelx/nforces)
positions = np.array(\
  [k*n for k in range(nforces)]\
    +[nelx], dtype=int)
f = np.zeros((ndof,nforces))
for i in range(nforces):
    start = positions[i]
    end = positions[i+1]
    f[dofs[start:end,0,1],i] \
        = -1.0/nelx
```

Since the function `solve_state` of lines 111–142 is designed to allow for such matrix argument `f`, the (unfiltered) method `J(self, x)` at line 161 returns without any further change the vector

$$(J_1(\boldsymbol{x}), J_2(\boldsymbol{x}), \dots, J_8(\boldsymbol{x})),$$

and the (unfiltered) method `dJ(self, x)` at line 165 returns the $8 \times n$ matrix

$$\begin{bmatrix} \mathrm{d}J_1(\boldsymbol{x}) \\ \mathrm{d}J_2(\boldsymbol{x}) \\ \vdots \\ \mathrm{d}J_8(\boldsymbol{x}) \end{bmatrix}, \tag{22}$$

where $\mathrm{d}J_i(\boldsymbol{x})$ is the sensitivity of the i-th compliance.

Third, the multi-objective optimization problem needs to be converted into the min/max formulation (21). This can be conveniently achieved using the decorator `@minmax_optimizable` provided by the package, which needs to be imported at the beginning of the code as follows:

```
from nullspace_optimizer import
    minmax_optimizable
```

Then, the conversion of the multi-objective problem to the min/max formulation is implemented with a single line of code by inserting the decorator before the definition of the class `TO_problem` just before line 155:

```
@minmax_optimizable
@bound_constraints_optimizable(l=0,
                               u=1)
@filtered_optimizable(filter,
                      diff_filter)
class TO_problem(
    EuclideanOptimizable):
    # ...
    # identical to lines 152-185
```

The decorator `@minmax_optimizable` provided by the Null Space Optimizer Python allows thus the user to implement the robust formulation (20) in a multi-objective fashion with the multivalued objective function `J(self,x)`. This paradigm could be easily extended to further multi-objective formulations, for instance it would not be difficult to implement a decorator `@weighted_optimizable` converting `TO_problem` into a class corresponding to the minimization of a weighted average of the objective functions.

In the present case, the decorator `@minmax_optimizable` returns a copy of the class `TO_problem` with the following changes:

- the method `x0(self)` returns $(\boldsymbol{x}^0, \max_i(J_i(\boldsymbol{x}^0)))$ where $\boldsymbol{x}^0$ is the output of the original method `x0` of `TO_problem`;
- the method `J(self,x)` returns $m \equiv$ `x[1]` since the optimization variable is now a tuple `x` $\equiv (\boldsymbol{x}, m)$;
- a method `H(self,x)` is added to return the inequality constraints $(\texttt{J}_i(\boldsymbol{x}) - m)_{1\leq i\leq 8}$;
- methods `dJ(self,x)` and `dH(self,x)` are added for automatically returning sensitivity Jacobian matrices updated with respect to the variable m. `dJ(self,x)` simply returns the $n+1$ dimensional row vector $\begin{bmatrix}0 & \dots & 0 & 1\end{bmatrix}$. The method `dH(self,x)` returns the $8\times(n+1)$ matrix

$$\begin{bmatrix} \mathrm{d}J_1(\boldsymbol{x}) & -1 \\ \mathrm{d}J_2(\boldsymbol{x}) & -1 \\ \vdots & \vdots \\ \mathrm{d}J_7(\boldsymbol{x}) & -1 \end{bmatrix}. \tag{23}$$

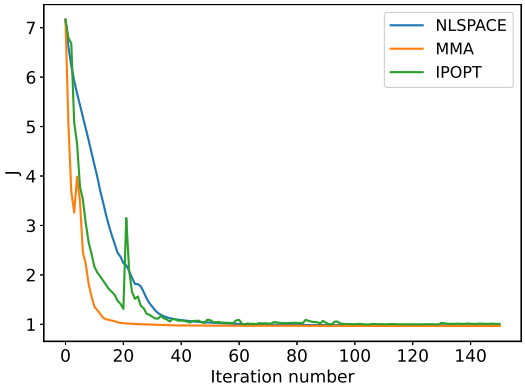
- finally, a method `retract(self,x,dx)` is added, which implements the projection rule

$$m \leftarrow \max_{1\leq i\leq 8} J_i(\boldsymbol{x}^{n+1}).$$

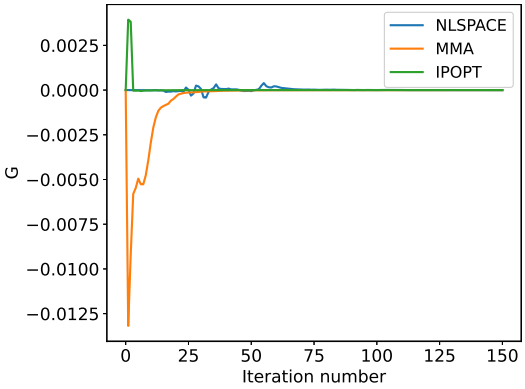
This projection is executed by the Null Space Optimizer after each update of the design variable. Although not fully necessary, this ensures a more conservative enforcement of the inequality constraints $J_i(\boldsymbol{x}) \leq m$ than the gradual correction of the range step ξ_C.

4.3 Numerical results and comparisons with MMA and IPOPT

We run the Null Space Optimizer on the bridge test case with the time step `dt`=0.1 and the setting `itnormalisation`=50. We also solve the design test case with the standard implementation of MMA (with `move`=0.1) and IPOPT (with default parameters). The final values of the compliance for all the load cases and of the volume constraint are reported in Table 3. Convergence histories of the objective and volume constraint are shown on Fig. 10. Intermediate designs obtained during the iterations of the three algorithms are displayed on



(a) Objective function J.



(b) Volume constraint $G(\boldsymbol{x}) = V(\boldsymbol{x})/(V_0\texttt{volfrac}) - 1$.

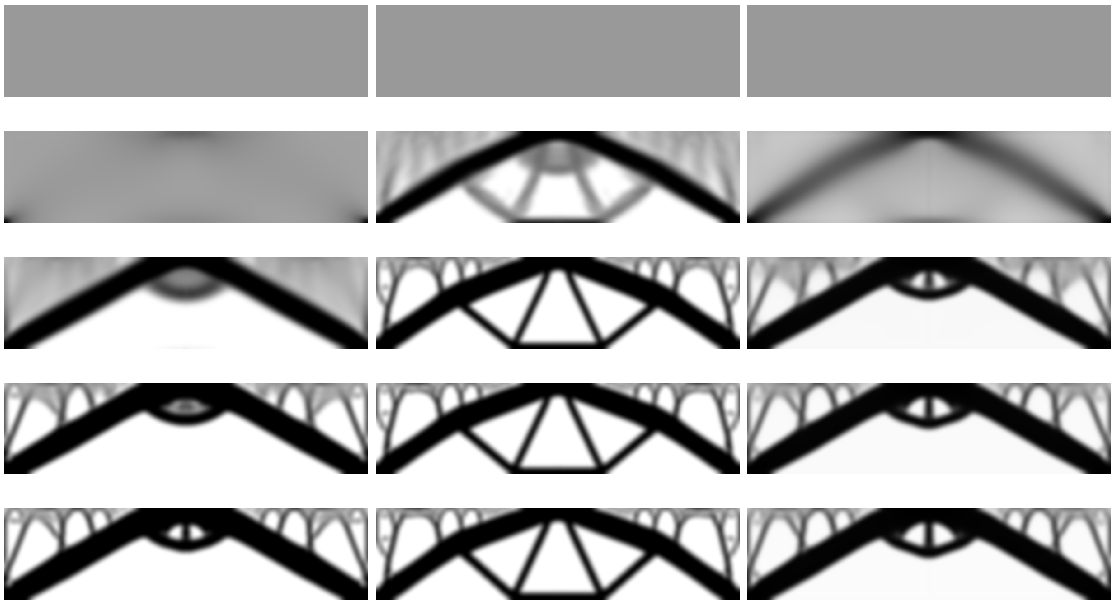
Fig. 10: Convergence history of the various optimizers for the multiple load bridge test case of Section 4.

Fig. 11; the whole bridge is represented by symmetrizing the numerically computed design for a better visualization.

On this example, the three optimizers tested converge to three different designs with similar performances. Noticeably, MMA takes a significantly distinct optimization path leading to a very different geometry while IPOPT and NLSPACE find designs that are similar up to some small differences at the middle of the structure. Here, the optimum found by MMA is slightly better than the one found by the Null Space Optimizer, which is itself slightly better than the one computed by IPOPT. It seems that there is a bifurcation between two possible local minimizers and that for some unclear reason, MMA selects the path leading to a better optimum. We observed that MMA

	NLSPACE	MMA	IPOPT
J_1	$9.77864 \cdot 10^{-1}$	$9.65891 \cdot 10^{-1}$	$1.00824 \cdot 10^{0}$
J_2	$8.30432 \cdot 10^{-1}$	$8.95300 \cdot 10^{-1}$	$8.61833 \cdot 10^{-1}$
J_3	$9.72427 \cdot 10^{-1}$	$9.65886 \cdot 10^{-1}$	$1.00589 \cdot 10^{0}$
J_4	$9.71707 \cdot 10^{-1}$	$9.65887 \cdot 10^{-1}$	$1.00830 \cdot 10^{0}$
J_5	$9.70662 \cdot 10^{-1}$	$9.65884 \cdot 10^{-1}$	$1.00822 \cdot 10^{0}$
J_6	$9.71344 \cdot 10^{-1}$	$9.65884 \cdot 10^{-1}$	$1.00821 \cdot 10^{0}$
J_7	$9.71813 \cdot 10^{-1}$	$9.65887 \cdot 10^{-1}$	$1.00823 \cdot 10^{0}$
J_8	$9.71894 \cdot 10^{-1}$	$9.65887 \cdot 10^{-1}$	$1.00825 \cdot 10^{0}$
$\max_{1 \leq i \leq 8} J_i$	$\mathbf{9.77864 \cdot 10^{-1}}$	$\mathbf{9.65891 \cdot 10^{-1}}$	$\mathbf{1.00830 \cdot 10^{0}}$
$G = V(x)/V_0 - \texttt{volfrac}$	$\mathbf{-3.10 \cdot 10^{-6}}$	$\mathbf{-1.60 \cdot 10^{-6}}$	$\mathbf{-9.80 \cdot 10^{-7}}$

Table 3: Final objective and constraint values obtained for the multiple load case of Section 4 with the Null Space Optimizer, MMA and IPOPT.



(a) NLSPACE (b) MMA (c) IPOPT

Fig. 11: Intermediate designs obtained by the different optimizers at iterations 0, 10, 40, 80, and 150 for the bridge test case of Section 4.

slightly violates the constraints $J_i(x) - m$ during the first iterations, which might help in finding afterwards a better optimization path.

5 Heat sink Topology Optimization on unstructured meshes

This section presents an additional tutorial for implementing Topology Optimization of 2D heat sinks. This classical test case was initially considered by Bejan (1997), and then solved with density based Topology Optimization in Gao et al (2008);

Zhang and Liu (2008); Marck and Privat (2014); Yan et al (2018).

The test case substantially deviates from the setting of the 88 lines Matlab TO paper of Andreassen et al (2011) considered in the previous parts, because we consider a different physics, unstructured mesh discretization and the use of continuous sensitivities rather than discrete ones.

The physical setting of the heat sink design problem is described in Section 5.1. A step by step description of the implementation of this test case within the paradigm of the Null Space Optimizer Python package is outlined in Section 5.2. We describe how to implement the physical state solver with FreeFEM (Hecht (2012)) and a Laplacian smoothing density filter is implemented to avoid the checkerboard effect on triangular meshes. The interfacing of the optimizer with the Finite Element software FreeFEM is especially advantageous as it could be used to deal with more challenging physics or for implementing density based TO on computational domains with arbitrary geometries. Finally, Section 5.3 presents numerical results of the proposed implementation including a comparison with the OC, MMA and IPOPT optimizers.

The content of this section partly follows the work of Krasniqi (2023) where the same test case was solved using a simpler extension of the Null Space algorithm for dealing with bound constraints. (namely, which did not consider an arbitrary inner product A as described in Appendix C).

5.1 Setting of the heat Topology Optimization problem

We consider a square domain $D = (0,1) \times (0,1)$ constituted of a conductive material with temperature field T subjected to a uniform heat source Q. A prescribed temperature $T = 0$ is applied on a part Γ_D of the left boundary. The normal heat flux $\partial T/\partial n$ vanishes on the remaining boundary $\Gamma_N := \partial D \backslash \overline{\Gamma_D}$, which is adiabatic. The length of Γ_D is $|\Gamma_D| = 0.2$. The computational domain D and its boundary conditions are illustrated on Fig. 12.

The optimization problem at hand is to find a distribution of two materials with constant conductivities $\kappa_f \gg \kappa_s$ that minimizes the average

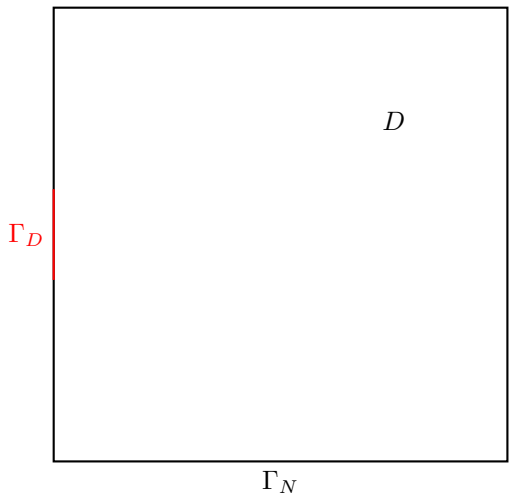


Fig. 12: Geometry for the heat sink test case of Section 5.

temperature on D, with a prescribed maximum volume fraction `volfrac` for the most conductive material κ_f. We use an "optimize-then-discretize" density based Topology Optimization approach, whereby the design variable is a density function $\rho : D \to (0,1)$ such that $\rho(x)$ corresponds to the local volume fraction of material of conductivity κ_f in an infinitesimal cell centered around x. The conductivity $\kappa(\rho)$ inside the whole domain D is determined by the SIMP interpolation law with penalization exponent $p = 3$:

$$\kappa(\rho) = \rho^p(\kappa_f - \kappa_s) + \kappa_s. \tag{24}$$

Of course, the purpose of the exponent p is to penalize intermediate densities so that numerically optimized designs ideally satisfy $\rho(x) = 0$ or $\rho(x) = 1$ in the whole domain.

The mathematical formulation of the heat sink design problem reads

$$\min_{\rho} \quad J(\rho) = \int_D T \,\mathrm{d}x$$
$$s.t. \begin{cases} V(\rho)/V_0 := \dfrac{1}{|D|}\displaystyle\int_D \rho \,\mathrm{d}x \le \texttt{volfrac} \\ 0 \le \rho(x) \le 1, \quad \text{for all } x \in D; \end{cases} \tag{25}$$

where $T \equiv T(\rho)$ is the solution to the heat equation

$$\begin{cases} -\mathrm{div}(\kappa(\rho)\nabla T) = Q \text{ in } D, \\ T = 0 \text{ on } \Gamma_D, \\ \dfrac{\partial T}{\partial n} = 0 \text{ on } \Gamma_N. \end{cases} \tag{26}$$

Note that in this test case, we treat the volume constraint as an inequality constraint.

In the following numerical applications, the numerical values of the physical parameters are set to

$$\kappa_f = 401,\ \kappa_s = 1,\ Q = 10^4, \texttt{volfrac} = 0.1.$$

Equation (26) is solved with the finite element method by assembling the weak formulation

$$\text{find } T \in V \text{ such that } \forall S \in V, \quad \int_D \kappa(\rho)\nabla T \cdot \nabla S \,\mathrm{d}x = \int_D QS \,\mathrm{d}x, \tag{27}$$

where $V = \{v \in H^1(D) \,|\, v_{|\Gamma_D} = 0\}$ is the subspace of H^1 functions vanishing on Γ_D (Brezis (2011)).

Following the "optimize-then-discretize" approach, we discretize the Fréchet derivatives of $J(\rho)$ and $V(\rho)$ rather than computing the derivatives of the discretized finite element functionals. For this classical self-adjoint problem, the Fréchet derivatives (Allaire, 2007, Definition 10.1.1) of the objective function and volume constraint respectively read:

$$\begin{aligned} \mathrm{D}J(\rho)\cdot\delta\rho &= \int_D \mathrm{D}T(\rho)\cdot\delta\rho \,\mathrm{d}x \\ &= \frac{1}{Q}\int_D \kappa(\rho)\nabla T \cdot \nabla(\mathrm{D}T(\rho)\cdot\delta\rho)\,\mathrm{d}x \\ &= -\frac{1}{Q}\int_D [\kappa'(\rho)\nabla T\cdot\nabla T]\delta\rho\,\mathrm{d}x, \end{aligned} \tag{28}$$

$$\mathrm{D}[V(\rho)/V_0]\cdot\delta\rho = \frac{1}{|D|}\int_D \delta\rho\,\mathrm{d}x, \tag{29}$$

where (28) is obtained by differentiating the variational equation (27) with respect to ρ.

Finally, in order to avoid checkerboard effects on the unstructured discretization mesh, we actually solve the following filtered version of (25):

$$\min_{\rho} \quad J(\widetilde{\rho}) = \int_D T(\widetilde{\rho})\,\mathrm{d}x \qquad s.t. \begin{cases} V(\widetilde{\rho})/V_0 \le \texttt{volfrac}, \\ 0 \le \rho(x) \le 1, \quad \text{for all } x \in D, \end{cases} \tag{30}$$

where $\widetilde{\rho}$ is a regularization of the design variable ρ obtained by solving the variational problem

$$\text{Find } \widetilde{\rho} \in H^1(D) \text{ such that } \forall v \in H^1(D), \quad \int_D (\gamma^2\nabla\widetilde{\rho}\cdot\nabla v + \widetilde{\rho}v)\,\mathrm{d}x = \int_D \rho v\,\mathrm{d}x, \tag{31}$$

with γ a smoothing parameter set to the minimum edge length. The variational problem (31) is the weak formulation of the equation

$$-\gamma^2\Delta\widetilde{\rho} + \widetilde{\rho} = \rho$$

with Neumann boundary conditions on ∂D. For this reason, the mapping $\rho \mapsto \widetilde{\rho}$ is sometimes called "Helmholtz type filter" despite (31) is not a wave equation. The density filter (31) is often preferred over (18) due to its ease of implementation on unstructured meshes. We refer the reader to Lazarov and Sigmund (2011) for an extensive comparison of filters in density based Topology Optimization.

5.2 Python implementation with the Null Space Optimizer and FreeFEM

The complete implementation of the heat Topology Optimization test case is provided in Appendix B. The code is divided into a main Python file (Appendix B.1) and two FreeFEM scripts `filter_matrices.edp` and `solve_state.edp` (respectively Appendix B.2 and Appendix B.3). The first one performs finite element operations for the filtering of the design variable, and the second one solves the state equations before computing the objective and constraint functions and their sensitivities. These two scripts are written in the FreeFEM language (Hecht (2012)) for the convenience of the implementation, and they are interfaced with the main

Python script through the library PyFreeFEM[5] importing the class `FreeFemRunner` at line 2 of the main script. In what follows, we systematically refer to the line numbering of the main script of Appendix B.1 unless otherwise specified.

5.2.1 Declaration of the optimization problem

Similarly to the MBB beam test case, the code is best understood by looking first at the declaration of the class `Heat_TO` at lines 61–95. The implementation carefully follows the mathematical definition of the design optimization problem (25): the class implements methods `x0`, `J`, `H` returning respectively the initial design, the values of the objective function and of the volume inequality constraint.

Similarly to the implementation of the MBB test case of Section 3, the method `x0` returns a constant array of length the number of mesh triangles corresponding to the initial design density $\rho(x) =$ `volfrac`. In the present implementation, the design variable is called `rho` in the Python script of Appendix B.1; it is a P0 discretization of the continuous variable ρ, that is constant on every triangle of the discretization mesh.

The implementation of the optimization class `Heat_TO` at lines 61–95 requires several ingredients which are described in the next subsections:

- a computational mesh `Th` with `Th.nt` triangles (Section 5.2.2);
- a state solver `solve_state` that returns a dictionary with the objective and constraint values as well as their sensitivities (Section 5.2.3);
- the declaration of an appropriate filter function `filter` as well as its derivative `diff_filter` on the triangular mesh `Th` (Section 5.2.4).

5.2.2 Construction of design domain mesh

The design domain D is discretized with a triangular finite element mesh `Th` that is generated at lines 15-19. The mesh build in FreeFEM with (line 17)

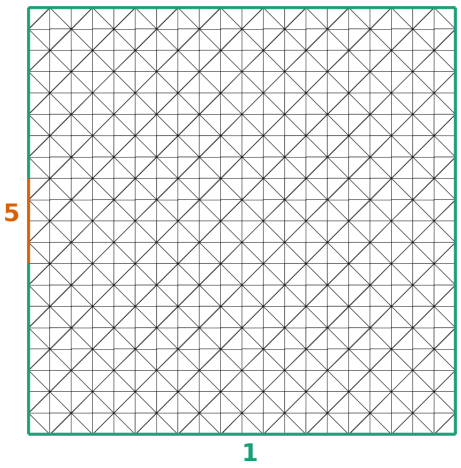


Fig. 13: Finite element mesh for the heat Topology Optimization problem of Section 5 with boundary labels corresponding to Γ_D and Γ_N.

```
mesh Th=square($n, $n, flags=1);
```

where `$N` is a prescribed number of edges on each side and `flags=1` allows to obtain a symmetric mesh. The FreeFEM code is executed at line (19) which returns a Python variable `Th` of the `Mesh` type defined the library PyMedit[6] (line 20). The `Mesh` type complies with the INRIA mesh format and could be easily exported to or imported from a `.mesh` or `.meshb` file. The variable `Th` is then manually processed to change the boundary labels, so that Γ_D is assigned to the label 5 and Γ_N to the label 1 (lines 21–30).

The mesh obtained for `N=10` is represented on Fig. 13. It would be straightforward to run the test case on any other triangular mesh by updating the variable `Th`.

5.2.3 Implementation of the physical state solver

The Python function `solve_state` is implemented at lines 49–59 of the main script. It executes the FreeFEM script `solve_state.edp` of Appendix B.3 through the PyFreeFEM interface (line 54). The instructions `importArray`, `exportArray` in the FreeFEM script allow to conveniently import and export FreeFEM data structures from and to Python. The dollar prefixed variables `$p`, `$kappaf`, `$kappas`, `$volfrac`

[5] https://gitlab.com/florian.feppon/pyfreefem

[6] https://gitlab.com/florian.feppon/pymedit

are input parameters assigned by the main Python script at lines 54–57.

The method `import_variables` of the library PyFreeFEM automatically stores the Python array `x` and the mesh `Th` in a temporary directory, which are then made accessible in the FreeFEM script `solve_state.edp` of Appendix B.3 through the instructions `importMesh("Th")` and `importArray("rho")` (lines 3 and 9 of this script). The script solve the heat equation (lines 21–24) and compute the objective function (line 26), the volume constraint (line 29) and their sensitivities (lines 32-33 and 35-36). As these lines demonstrate, the use of FreeFEM is especially convenient for implementing (25) and (27) to (29) with a language very close to the continuous mathematical formulation.

The objective function, volume constraints and their sensitivities are then exported towards the Python script at lines 38–42 with the instructions `exportVar` and `exportArray`. Then, the execution of the FreeFEM script occurs at line 54 of the main script of Appendix B.1 with the method `execute` which returns a Python dictionary `exports` gathering the exported data structures: for instance `exports['J']` and `exports['dJ']` are respectively a floating number and a NumPy array containing the objective function value and its sensitivity.

5.2.4 Implementation of a smoothing density filter

The implementation of the density filter $\rho \mapsto \widetilde{\rho}$ defined by (31) is automated by the decorator

```
@filtered_optimizable(filter,
                      diff_filter)
```

at line 63 of the main Python script. Here, the implementation of the filtering procedure requires a little caution because ρ is implemented as a P0 function `rho` $\equiv \rho_{\mathrm{P0}}$ while the variational problem (31) solved with P1 finite elements returns a P1 function $\widetilde{\rho}_{\mathrm{P1}}$. To address this issue, the proposed implementation converts this P1 function $\widetilde{\rho}_{\mathrm{P1}}$ to a P0 function $\widetilde{\rho}_{\mathrm{P0}}$ by solving the variational problem: find a P0 function $\widetilde{\rho}_{\mathrm{P0}}$ such that for any P0 function v_{P0},

$$\int_D \widetilde{\rho}_{\mathrm{P0}} v_{\mathrm{P0}}\,\mathrm{d}x = \int_D \widetilde{\rho}_{\mathrm{P1}} v_{\mathrm{P0}}\,\mathrm{d}x. \qquad (32)$$

In a shorthand matrix notation, the numerical filtering procedure mapping P0 to P0 densities reads

$$\widetilde{\rho}_{\mathrm{P0}} := M_{\mathrm{P0P0}}^{-1} M_{\mathrm{P1P0}}^{T} A^{-1} M_{\mathrm{P1P0}}\, \rho_{\mathrm{P0}}. \qquad (33)$$

where we have denoted by M_{P1P0} and M_{P0P0} the mass matrices discretizing the bilinear form $(u, v) \mapsto \int_D uv\,\mathrm{d}x$ with v a P0 function and u respectively a P1 or a P0 function.

The assembly of the finite element matrices A, M_{P0P0} and M_{P1P0} is performed by the FreeFEM script `filter_matrices.edp` of Appendix B.2. This script is executed between lines 32–37 of the main Python script of Appendix B.1. The matrices are exported to sparse Scipy matrices and a factorization of A and M_{P0P0} is computed at lines 37-38 in order to compute efficiently the mapping (33).

Finally, the filtering map `filter` and its Fréchet derivative `diff_filter` that are required by the decorator `@filtered_optimizable` are implemented at lines 41–47 of the main script by copying the formula (33).

5.2.5 Running the optimizer

The test case is instantiated at line 100 with the function `init(100)` which constructs a computational mesh `Th` with `N`=100 edges on each boundary. The optimization parameters are set at lines 100–104, setting a time step `dt`=0.3, 50 normalization iterations and stopping the optimization after 150 iterations. The design optimization problem is solved with the Null Space Optimizer at line 105 by calling the function `nlspace_solve`. The execution of the code generates plots of the intermediate densities thanks to the instructions given in the method `accept` at lines 87–95, relying on the plotting features of PyMedit (line 92).

5.3 Numerical results and comparisons with OC, MMA and IPOPT

We solve the heat Topology Optimization problem (30) with the Null Space Optimizer, MMA, IPOPT and the Optimality Criteria method. The `move` parameters of the OC method and MMA are set to 0.1. IPOPT runs with the default settings.

The numerical values of the optimized designs found by the different optimizers are reported in

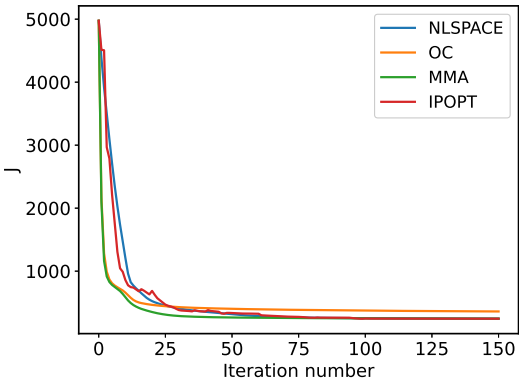
Algorithm	J	$H = V(\boldsymbol{x})/V_0 - \texttt{volfrac}$
NLSPACE	$2.45701 \cdot 10^2$	$9.60 \cdot 10^{-11}$
OC	$3.60279 \cdot 10^2$	$-7.90 \cdot 10^{-6}$
MMA	$2.46293 \cdot 10^2$	$-1.90 \cdot 10^{-5}$
IPOPT	$2.45544 \cdot 10^2$	$-1.90 \cdot 10^{-6}$

Table 4: Final objective and constraint values for the heat Topology Optimization test case of Section 5 obtained for the tested optimization algorithms.

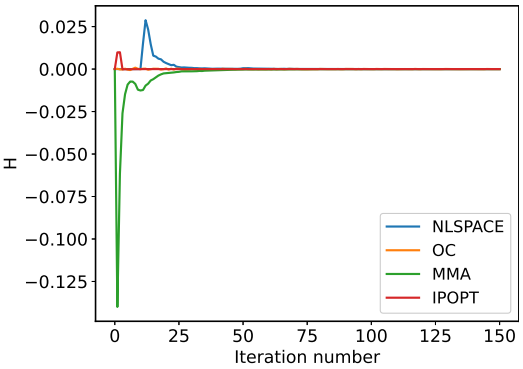
Table 4. Convergence histories of the objective function and volume constraint are reported on Fig. 14. Figure 15 depicts a sequence of intermediate designs obtained for each of the optimizers.

For this example, the OC method converges to a design which performs more poorly (by a factor of approximately 30%) than its competitors. MMA, NLSPACE and IPOPT converge to different designs with very similar performance. Notably, the four optimizers drive the design variable on significantly different optimization paths.

Let us conclude by noticing that none of the optimizers converge to needle like structures that are known to be optimal for this problem Yan et al (2018).

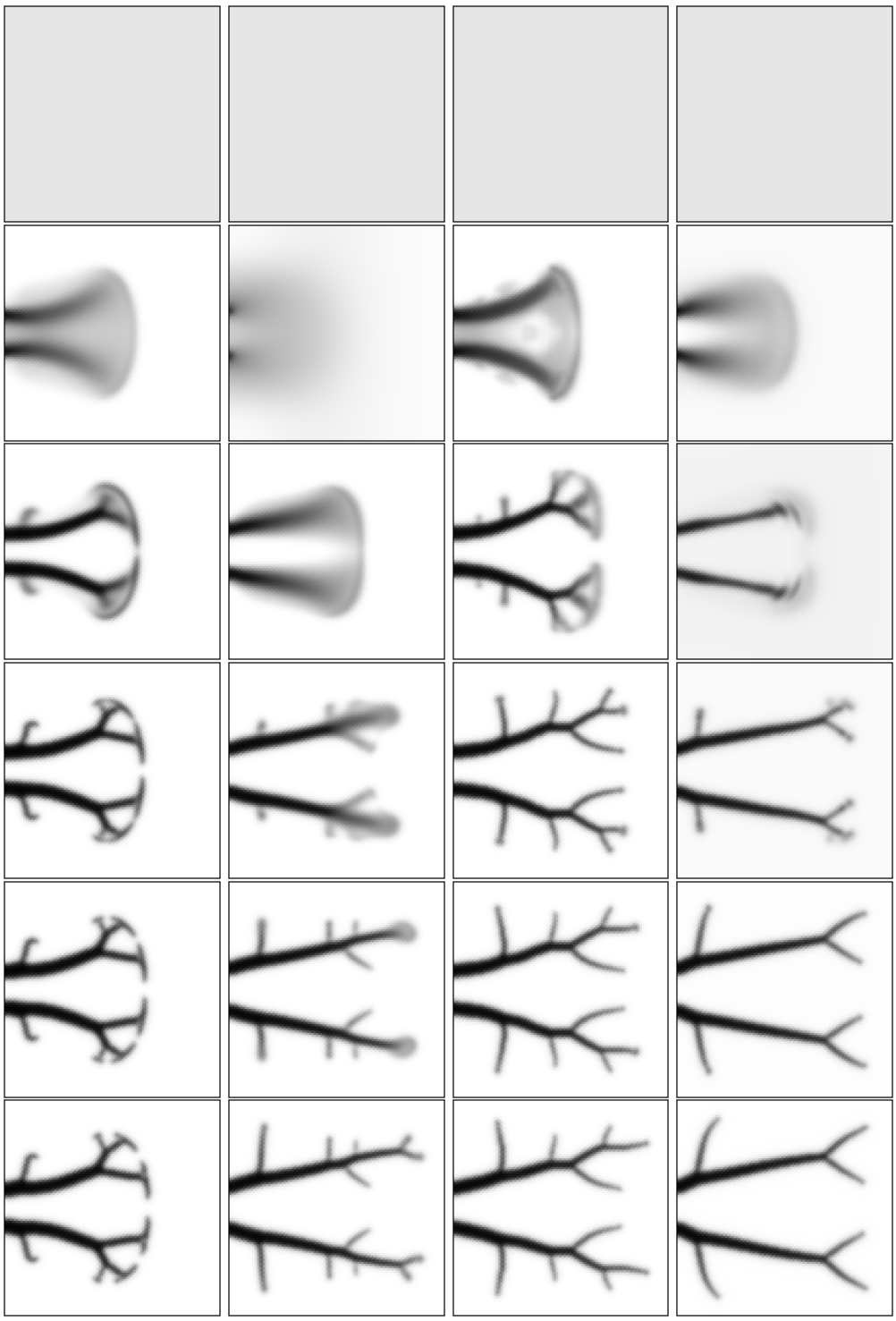


(a) Objective function J.



(b) Volume constraint $G(\boldsymbol{x}) = V(\boldsymbol{x})/(V_0\texttt{volfrac}) - 1$.

Fig. 14: Convergence history of the different optimizers tested on the heat Topology Optimization test case of Section 5.



(a) Optimality Criteria (b) Null Space (c) MMA (d) IPOPT

Fig. 15: Intermediate designs of the heat Topology Optimization test case of Section 5 obtained with the different optimizers at iterations 0, 10, 15, 20, 40 and 80.

Appendix A Source code of the MBB beam example of Section 3

```
from nullspace_optimizer import EuclideanOptimizable,\
    bound_constraints_optimizable, memoize, filtered_optimizable,\
    nlspace_solve
import numpy as np
import cvxopt
import cvxopt.cholmod
import scipy.sparse as sp
import matplotlib.pyplot as plt
from matplotlib import colors

# element stiffness matrix
def lk():
    E = 1
    nu = 0.3
    k = np.array([1/2-nu/6, 1/8+nu/8, -1/4-nu/12, -1/8+3*nu/8,
                  -1/4+nu/12, -1/8-nu/8, nu/6, 1/8-3*nu/8])
    KE = E/(1-nu**2)*np.array([
        [k[0], k[1], k[2], k[3], k[4], k[5], k[6], k[7]],
        [k[1], k[0], k[7], k[6], k[5], k[4], k[3], k[2]],
        [k[2], k[7], k[0], k[5], k[6], k[3], k[4], k[1]],
        [k[3], k[6], k[5], k[0], k[7], k[2], k[1], k[4]],
        [k[4], k[5], k[6], k[7], k[0], k[1], k[2], k[3]],
        [k[5], k[4], k[3], k[2], k[1], k[0], k[7], k[6]],
        [k[6], k[3], k[4], k[1], k[2], k[7], k[0], k[5]],
        [k[7], k[2], k[1], k[4], k[3], k[6], k[5], k[0]]])
    return (KE)

def deleterowcol(A, delrow, delcol):
    # Delete row and columns of a sparse csc matrix A
    m = A.shape[0]
    keep = np.delete(np.arange(0, m), delrow)
    A = A[keep, :]
    keep = np.delete(np.arange(0, m), delcol)
    A = A[:, keep]
    return A

# Max and min stiffness
Emin = 1e-9
Emax = 1.0

# Assemble FEM matrices and filter
def init(**kwargs):
    # Initialized global variables
    global nelx, nely, volfrac, rmin, penal
    global H, Hs, ndof, KE, iK, jK, fixed, free
    global edofMat, f, dofs

```

```
48      # Default input parameters
49      nelx = kwargs.get('nelx', 180)
50      nely = kwargs.get('nely', 60)
51      volfrac = kwargs.get('volfrac', 0.4)
52      rmin = kwargs.get('rmin', 5.4)
53      penal = kwargs.get('penal', 3.0)
54
55      # Degrees of Freedom matrix: edofMat[i,j,:] contains the
56      # DOFs associated to the square element (i,j)
57      ndof = 2*(nelx+1)*(nely+1)
58      dofs = np.arange(ndof).reshape((nelx+1, nely+1, 2))
59      edofMat = np.zeros((nelx, nely, 8), dtype=int)
60      edofMat[:, :, 0] = dofs[:-1, 1:, 0]
61      edofMat[:, :, 1] = dofs[:-1, 1:, 1]
62      edofMat[:, :, 2] = dofs[1:, 1:, 0]
63      edofMat[:, :, 3] = dofs[1:, 1:, 1]
64      edofMat[:, :, 4] = dofs[1:, :-1, 0]
65      edofMat[:, :, 5] = dofs[1:, :-1, 1]
66      edofMat[:, :, 6] = dofs[:-1, :-1, 0]
67      edofMat[:, :, 7] = dofs[:-1, :-1, 1]
68      edofMat = edofMat.reshape((nelx*nely, 8))
69
70      # FE: Build the index vectors for the for coo matrix format.
71      KE = lk()
72      # Construct the index pointers for the coo format
73      iK = np.kron(edofMat, np.ones((8, 1))).flatten()
74      jK = np.kron(edofMat, np.ones((1, 8))).flatten()
75
76      # BC's and support
77      fixed = np.union1d(dofs[0, :, 0], dofs[-1, -1, 1])
78      free = np.setdiff1d(dofs.reshape(ndof), fixed)
79
80      # Filter: Build (and assemble) the index+data vectors
81      # for the coo matrix format
82      nfilter = int(nelx*nely*((2*(np.ceil(rmin)-1)+1)**2))
83      iH = np.zeros(nfilter)
84      jH = np.zeros(nfilter)
85      sH = np.zeros(nfilter)
86      cc = 0
87      for i in range(nelx):
88          for j in range(nely):
89              row = i*nely+j
90              kk1 = int(np.maximum(i-(np.ceil(rmin)-1), 0))
91              kk2 = int(np.minimum(i+np.ceil(rmin), nelx))
92              ll1 = int(np.maximum(j-(np.ceil(rmin)-1), 0))
93              ll2 = int(np.minimum(j+np.ceil(rmin), nely))
94              for k in range(kk1, kk2):
95                  for l in range(ll1, ll2):
96                      col = k*nely+l
97                      fac = rmin-np.sqrt(((i-k)*(i-k)+(j-l)*(j-l)))
```

```
                    iH[cc] = row
                    jH[cc] = col
                    sH[cc] = np.maximum(0.0, fac)
                    cc = cc+1
    # Finalize assembly and convert to csc format
    H = sp.coo_matrix((sH, (iH, jH)),
                      shape=(nelx*nely, nelx*nely)).tocsc()
    Hs = sp.diags(1/np.asarray(H.sum(1)).flatten())

    # f is the unit load vector at the bottom right corner
    f = np.zeros((ndof, 1))
    f[1, 0] = -1

# Compute compliance and its sensitivity
@memoize()
def solve_state(x):
    dc = np.ones(nely*nelx)
    ce = np.ones(nely*nelx)

    # Setup and solve FE problem
    sK = ((KE.flatten()[np.newaxis]).T*(Emin+x**penal*(Emax-Emin)))\
        .flatten(order='F')
    K = sp.coo_matrix((sK, (iK, jK)), shape=(ndof, ndof)).tocsc()
    # Remove constrained dofs from matrix and convert to coo
    K = deleterowcol(K, fixed, fixed).tocoo()
    # Solve system
    K = cvxopt.spmatrix(K.data, K.row.astype(int), K.col.astype(int))
    B = cvxopt.matrix(f[free, :])
    cvxopt.cholmod.linsolve(K, B)
    u = np.zeros((ndof, f.shape[1]))
    u[free, :] = np.array(B)[:, :]

    # compliance
    ce = np.einsum('ija,jk,ika->ia', u[edofMat, :], KE, u[edofMat, :])
    obj = (Emin + (x**penal*(Emax-Emin))).dot(ce)

    # sensitivity of compliance with respect to x
    dc = np.multiply((-penal*x**(penal-1)*(Emax-Emin))[:, np.newaxis],
                     ce).T

    # If not multi-objective
    if len(obj)==1:
        return (obj[0], dc[0,:])
    # else
    return (obj, dc)

# Density filter
@memoize()
def filter(x):
    return Hs @ (H @ x)
```

```
148
149 # Sensitivity of the filter
150 @memoize()
151 def diff_filter(x, v):
152     return (H @ (Hs @ v.T)).T
153
154 # Definition of the optimization problem
155 @bound_constraints_optimizable(l=0, u=1)
156 @filtered_optimizable(filter, diff_filter)
157 class TO_problem(EuclideanOptimizable):
158     def x0(self):
159         return volfrac * np.ones(nely*nelx, dtype=float)
160
161     def J(self, x):
162         (obj, dc) = solve_state(x)
163         return obj
164
165     def dJ(self, x):
166         (obj, dc) = solve_state(x)
167         return dc
168
169     def G(self, x):
170         return [np.sum(x)/(nelx*nely)-volfrac]
171
172     def dG(self, x):
173         dv = np.ones(nelx*nely)/(nelx*nely)
174         return dv
175
176     def accept(self, params, results):
177         # Plot the design at every iteration
178         x = results['x'][-1]
179         if not hasattr(self, 'im'):
180             plt.ion()  # Ensure that redrawing is possible
181             self.fig, self.ax = plt.subplots()
182             self.im = self.ax.imshow(-x.reshape((nelx, nely)).T,\
183                                      cmap='gray',
184                                      interpolation='none',
185                                      norm=colors.Normalize(vmin=-1, vmax=0))
186             self.ax.axis('off')
187             self.fig.show()
188         else:
189             self.im.set_array(-x.reshape((nelx, nely)).T)
190             self.fig.canvas.draw()
191         plt.pause(0.01)
192
193 # Optimization parameters
194 optimization_params = {'dt': 0.3,
195                        'itnormalisation': 50,
196                        'save_only_N_iterations': 1,
197                        'save_only_Q_constraints': 5,
```

```
                        'maxit': 150}
# Initialize and solve the TO problem
init()
case = TO_problem()
results = nlspace_solve(case, optimization_params)
input("Press any key")
```

Appendix B Source code of the Heat Topology Optimization example of Section 5

B.1 Main Python script

```
import numpy as np
from pyfreefem import FreeFemRunner
from pymedit import P0Function
from nullspace_optimizer import EuclideanOptimizable, memoize, \
    nlspace_solve, bound_constraints_optimizable, filtered_optimizable
import matplotlib.pyplot as plt
import scipy.sparse.linalg as lg

def init(n=100, vfrac=0.1):
    global volfrac, Th
    global filter, diff_filter, solveA, MP1P0, solveMP0P0

    volfrac = vfrac

    mesh_script = """
        IMPORT "io.edp"
        mesh Th = square($n, $n, flags=1);
        exportMesh(Th);"""
    Th = FreeFemRunner(mesh_script).execute({'n':n})['Th']

    # Changing boundary label on left hand side
    pts1 = Th.vertices[Th.edges[:,0]-1]
    pts2 = Th.vertices[Th.edges[:,1]-1]
    eps = 1/(10*n)
    indices = np.logical_and(abs(pts1[:,1]-0.5)<=0.1+eps,
                             abs(pts2[:,1]-0.5)<=0.1+eps)
    indices = np.logical_and(indices,Th.edges[:,-1]==4)
    Th.edges[indices,-1]=5
    Th.edges[Th.edges[:,-1]!=5,-1] = 1
    Th._AbstractMesh__updateBoundaries()

    # execute FreeFEM code and retrieves the matrices
    # in Python
    runner = FreeFemRunner("filter_matrices.edp")
    runner.import_variables(Th=Th)
    exports = runner.execute()
    solveA = lg.factorized(exports['A'])
    solveMP0P0 = lg.factorized(exports['MP0P0'])
```

```
39      MP1P0 = exports['MP1P0']
40
41  @memoize()
42  def filter(rho):
43      return solveMP0P0(MP1P0.T @ solveA( MP1P0 @ rho))
44
45  @memoize()
46  def diff_filter(rho, v):
47      return (MP1P0.T @ solveA( MP1P0 @ solveMP0P0(v.T))).T
48
49  @memoize()
50  def solve_state(rho):
51      # Execute FreeFEM script with input density rho
52      runner = FreeFemRunner("solve_state.edp")
53      runner.import_variables(rho = rho, Th=Th)
54      exports = runner.execute({'p':3,'Q':1e4,
55                                'kappaf':401,
56                                'kappas':1,
57                                'volfrac':volfrac})
58      # exports: an array with entries J, H, dJ, dH
59      return exports
60
61  # Declaration of the optimization problem
62  @bound_constraints_optimizable(l=0,u=1)
63  @filtered_optimizable(filter, diff_filter)
64  class Heat_TO(EuclideanOptimizable):
65      def __init__(self, plot=None):
66          self.plot = plot
67
68          if self.plot:
69              plt.ion()
70              self.fig = plt.figure()
71
72      def x0(self):
73          return volfrac * np.ones(Th.nt,dtype=float)
74
75      def J(self, rho):
76          return solve_state(rho)['J']
77
78      def H(self, rho):
79          return [solve_state(rho)['H']]
80
81      def dJ(self, rho):
82          return solve_state(rho)['dJ']
83
84      def dH(self, rho):
85          return solve_state(rho)['dH']
86
87      def accept(self, params, results):
88          # Plotting
```

```
89          if self.plot:
90              rho = results['x'][-1]
91              self.fig.clear()
92              P0Function(Th, rho).plot(fig=self.fig, ax=plt.gca(),\
93                          cmap="gray_r", vmin=0, vmax=1,
94                          title = 'Iteration␣'+str(results['it'][-1]))
95              plt.pause(0.1)
96
97 init(100)
98
99 case = Heat_TO(plot=True)
100 params = dict(dt=0.1,
101               itnormalisation=50,
102               maxit=150,
103               save_only_N_iterations=1,
104               save_only_Q_constraints=5)
105 nlspace_solve(case, params)
106 input("Press␣any␣key")
```

B.2 File filter_matrices.edp

```
1  IMPORT "io.edp"
2  mesh Th = importMesh("Th");
3
4  fespace Fh0(Th,P0);
5  fespace Fh1(Th,P1);
6
7  varf mass(u,v)=int2d(Th)(u*v);
8
9  matrix MP1P0 = mass(Fh0,Fh1);
10 matrix MP0P0 = mass(Fh0,Fh0);
11
12 macro grad(u) [dx(u),dy(u)]//
13
14 real gamma = Th.hmin;
15 varf helmholtz(u,v) = int2d(Th)(gamma^2*grad(u)'*grad(v)+u*v);
16 matrix A = helmholtz(Fh1,Fh1);
17
18 exportMatrix(MP1P0);
19 exportMatrix(MP0P0);
20 exportMatrix(A);
```

B.3 File solve_state.edp

```
1  IMPORT "io.edp"
2
3  mesh Th = importMesh("Th");
4
5  fespace Fh0(Th,P0);
6  fespace Fh1(Th,P1);
```

```
7
8  Fh0 rho;
9  rho[] = importArray("rho");
10
11 real p = $p;
12 real kappaf = $kappaf;
13 real kappas = $kappas;
14 Fh0 kappa = rho^p*(kappaf-kappas)+kappas;
15 Fh0 dkappa = p*rho^(p-1)*(kappaf-kappas);
16 Fh1 T, S;
17 func Q = $Q;
18
19 macro grad(u) [dx(u),dy(u)] //
20
21 solve heat(T,S)=
22     int2d(Th)(kappa*grad(T)'*grad(S))
23     -int2d(Th)(Q*S)
24     +on(5,T=0);
25
26 real J = int2d(Th)(T);
27 real vol0 = int2d(Th)(1.);
28 real volfrac = $volfrac;
29 real H = int2d(Th)(rho/(vol0*volfrac))-1;
30
31 Fh0 drho, dummy;
32 varf DJ(dummy, drho) = -int2d(Th)(dkappa*grad(T)'*grad(T)*drho*1.0/Q);
33 real[int] dJ = DJ(0,Fh0);
34
35 varf DH(dummy, drho) = int2d(Th)(drho/(vol0*volfrac));
36 real[int] dH = DH(0,Fh0);
37
38 exportVar(J);
39 exportVar(H);
40 exportArray(T[]);
41 exportArray(dJ);
42 exportArray(dH);
```

Appendix C Treatment of constraints with sparse Jacobian matrix

In this appendix, we describe the modifications of the Null Space Optimizer that enable to solve optimization programs with constraints with sparse Jacobian matrix, deviating from the original description of this algorithm in Feppon et al (2020a). The theoretical changes for the computation of the null space and range space directions are detailed in Appendix C.1. Then, in Appendix C.2, we challenge the Null Space Optimizer on a benchmark numerical example featuring many bound constraints and we provide qualitative comparisons with the standard IPOPT and MMA optimizers.

C.1 Sparse computations of the null space and range space directions

In this part only, we denote by A a given $n\times n$ symmetric positive definite matrix and we assume that the norm considered in (8) and (9) is the Euclidean norm $||\cdot|| \equiv ||\cdot||_A$ associated to the inner product $\langle x,y\rangle_A := x^TAy$, namely $||x||_A := |x^TAx|^{\frac{1}{2}}$. For many practical situations including density based Topology Optimization, the reader may assume that A is the identity matrix. However, we need to consider an arbitrary inner product A to account for more general contexts such as the Hilbertian setting of level-set based Topology Optimization where such matrix is used to regularize shape derivatives (de Gournay (2006)).

Using an arbitrary inner product has for consequence that the transpose operator $\cdot^T$ must be preceded by A^{-1} in the definitions (7), (8) and (15) of respectively $\xi_J(x)$, $\Lambda^*(x) := (\boldsymbol{\lambda}^*(x),\boldsymbol{\mu}^*(x))$ and $\xi_C(x)$. The null space direction $\xi_J(x)$ now reads

$$\xi_J(x) = A^{-1}\mathrm{D}J(x)^T + A^{-1}\mathrm{D}C_{\widetilde{I}(x)}(x)^T\Lambda^*(x), \tag{C1}$$

where the multiplier $\Lambda^*(x) := (\boldsymbol{\lambda}^*(x),\boldsymbol{\mu}^*(x))$ is the minimizer of the problem

$$\min_{\Lambda=(\boldsymbol{\lambda},\boldsymbol{\mu})\in\mathbb{R}^p\times\mathbb{R}_+^{\widetilde{q}(x)}} ||A^{-1}\mathrm{D}J(x)^T+A^{-1}\mathrm{D}C_{\widetilde{I}(x)}(x)^T\Lambda||_A. \tag{C2}$$

The range space direction is updated as follows:

$$\xi_C(x) = A^{-1}\mathrm{D}C_{\widetilde{I}(x)}^T(\mathrm{D}C_{\widetilde{I}(x)}A^{-1}\mathrm{D}C_{\widetilde{I}(x)}^T)^{-1}C_{\widetilde{I}(x)}. \tag{C3}$$

The reader may verify that these formulas coincide with (7), (8) and (10) when substituting A with the $n\times n$ identity matrix.

We now show that it is possible to rewrite (C1) to (C3) only in terms of $\mathrm{D}C_{\widetilde{I}(x)}$ and A, avoiding altogether the need for assembling $\mathrm{D}C_{\widetilde{I}(x)}A^{-1}\mathrm{D}C_{\widetilde{I}(x)}$ or computing its explicit inverse.

Proposition 1. (i) The null space direction $\xi_J(x)$ defined in (C1) is equivalently given by

$$\xi_J(x) = A^{-1}\mathrm{D}J(x)^T + X^*(x), \tag{C4}$$

where $X^*(x)$ is obtained from the solution $(X^*(x),\Lambda^*(x))$ to the quadratic minimization program

$$\min_{(X,\Lambda)\in\mathbb{R}^n\times\mathbb{R}^{p+\widetilde{q}(x)}} \frac{1}{2}X^TAX + \mathrm{D}J(x)X \quad s.t. \begin{cases} AX - \mathrm{D}C_{\widetilde{I}(x)}^T\Lambda = 0, \\ \Lambda = (\boldsymbol{\lambda},\boldsymbol{\mu})\in\mathbb{R}^p\times\mathbb{R}^{\widetilde{q}(x)} \text{ with } \boldsymbol{\mu}\geq 0. \end{cases} \tag{C5}$$

(ii) The minimizer $\Lambda^*(x)$ of (C5) is also the optimal Lagrange multiplier $\Lambda^*(x) = (\boldsymbol{\lambda}^*(x),\boldsymbol{\mu}^*(x))$ as defined by (C2).

(iii) The range space direction $\xi_C(x)$ defined in (C3) can be obtained from the solution to the symmetric indefinite linear system

$$\begin{bmatrix} A & \mathrm{D}C_{\widetilde{I}(x)}^T \\ \mathrm{D}C_{\widetilde{I}(x)} & 0 \end{bmatrix}\begin{bmatrix}\xi_C(x)\\ \Lambda\end{bmatrix} = \begin{bmatrix}0\\ C_{\widetilde{I}(x)}\end{bmatrix}. \tag{C6}$$

Proof. The square of the right-hand side of (C2) reads

$$\begin{aligned}
&||A^{-1}\mathrm{D}J(x)^T + A^{-1}\mathrm{D}C_{\widetilde{I}(x)}^T\Lambda||_A^2 \\
&= \mathrm{D}J(x)A^{-1}\mathrm{D}J(x)^T + 2\mathrm{D}J(x)A^{-1}\mathrm{D}C_{\widetilde{I}(x)}^T\Lambda \\
&\quad + \Lambda^T\mathrm{D}C_{\widetilde{I}(x)}A^{-1}\mathrm{D}C_{\widetilde{I}(x)}^T\Lambda \\
&= X^TAX + \mathrm{D}J(x)^TX,
\end{aligned}$$

where we have introduced $X := A^{-1}\mathrm{D}C_{\widetilde{I}(x)}^T\Lambda$. Since X is equivalently characterized by $AX - \mathrm{D}C_{\widetilde{I}(x)}^T\Lambda = 0$, this proves the point (ii) and the point (i) follows from (C1). The point (iii) is obtained by observing that $\mathrm{D}C_{\widetilde{I}(x)}\xi_C(x) = C_{\widetilde{I}(x)}$ and $A\xi_C(x) + \mathrm{D}C_{\widetilde{I}(x)}^T\Lambda = 0$ where $\Lambda := -(\mathrm{D}C_{\widetilde{I}(x)}A^{-1}\mathrm{D}C_{\widetilde{I}(x)}^T)^{-1}C_{\widetilde{I}(x)}$. □

When both the matrices A and $\mathrm{D}C_{\widetilde{I}(x)}$ are sparse, Proposition 1 provides a definition of $\xi_J(x)$ and $\xi_C(x)$ that becomes accessible to iterative solvers. A number of strategies can be devised for efficiently solving the symmetric indefinite system (C6), also called KKT system, see e.g. Freund and Nachtigal (1994); Ashcraft et al (1998); Toh et al (2004); Bai et al (2009); Nocedal and Wright (2006). The present implementation of the Null Space Optimizer solves (C6) with Scipy `lsqr` method with machine precision. The quadratic minimization problem (C5) with the nonnegativity constraint $\boldsymbol{\mu} \geq 0$ is solved with OSQP (Stellato et al (2020)) or CVXOPT (Vandenberghe (2010)), which both internally rely on sparse linear solvers. Finally, let us note that the proposed implementation also addresses the issue of degenerate constraints, which are either dealt with the internal iterative solvers of OSQP or CVXOPT, or with Scipy least-squares `lsqr` solver. All the examples of Sections 3 to 5 considered in this paper relied on the OSQP solver as it seems to provide in general better runtime performances than CVXOPT (see Appendix C.2).

Remark 5. The quadratic program (C5) is rather large if the matrix $\mathrm{D}C_{\widetilde{I}(x)}$ features a large number of constraints, resulting in a potentially higher cost per optimization iteration than solving the MMA subproblem (Svanberg (1987)). However, for the test cases considered in this article, we found the OSQP solver quite efficient at solving the quadratic problem (C5) in reasonable time, which is therefore not a bottleneck of the method.

C.2 A benchmark example: generating a checkerboard

In order to illustrate the numerical treatment of bound constraints and the influence of the quadratic programming solver, we consider the benchmark minimization problem of generating a checkerboard on a $n \times n$ grid:

$$\min_{(x_{ij})_{1\leq i,j\leq n}} \quad J((x_{ij})_{1\leq i,j\leq n}) \\ s.t. \quad 0 \leq x_{ij} \leq 1 \text{ for all } 1 \leq i \leq n. \tag{C7}$$

where the function $J((x_{ij})_{1\leq i,j\leq n})$ measures the opposite of the discrepancies of the variables $(x_{ij})_{1\leq i,j\leq n}$ between adjacent cells on the $n \times n$ grid:

$$J((x_{ij})_{1\leq i,j\leq N}) := -\frac{1}{2}\left[\sum_{i=1}^{n-1}\sum_{j=1}^{n}(x_{i+1,j} - x_{i,j})^2 + \sum_{i=1}^{n}\sum_{j=1}^{n-1}(x_{i,j+1} - x_{i,j})^2\right] \tag{C8}$$

The global minimum is clearly attained for a checkerboard, namely for a variable $(x_{ij})_{1\leq i,j\leq n}$ satisfying $|x_{i+1,j} - x_{ij}| = |x_{i,j+1} - x_{ij}| = 1$ for all $1 \leq i, j \leq n$. However, due to the quadratic exponents, the problem (C7) admits many local minima. The minimization is thus very sensitive to numerical noise and to the chosen optimization path, which makes it an interesting benchmark for comparing several optimizers.

We run the Null Space Optimizer with the time step `dt` $= 0.3$, using either OSQP or CVXOPT as the quadratic programming solver for (3). The number of normalization iterations for the null space step is set to `itnormalisation`=10. The tolerance parameters of both the OSQP and CVXOPT solver are set to 10^{-15}. The number of scaling iterations as well as the maximum number of iterations allowed for the OSQP solver is set to 400. The maximum number of iterations for the CVXOPT solver is set to 30 (these are observed to be computationally more costly).

The initialization is defined to be the density

$$x_{ij}^0 := ((i-1)/(n-1)-0.5)((j-1)/(n-1)-0.5), \quad 1 \leq i,j \leq n, \quad \text{(C9)}$$

which is symmetric with respect to the grid medial axes. We also solve (C7) with the optimizers IPOPT and MMA for comparison (we do not use the OC method which is not relevant in this context devoid of equality constraints). The `move` parameter of MMA is set to 0.3 and IPOPT is run with the default parameters.

The numerical values of the computed optima are listed in Table C1 and the convergence histories are shown on Fig. C1. Intermediate design variables are shown on Fig. C2 for the various optimization strategies. On this test case, IPOPT finds the best optimum. The version of MMA provided by Deetman (2020) converges to a very poor solution, probably due to the fact that the default moving asymptotes rules is not suited on this particular example that only features bound constraints.

The CVXOPT variant of the Null Space Optimizer converges to a rather suboptimal local minimum, although it provides a slightly faster decay than its OSQP variant at the first iterations: CVXOPT seems unable to update the design variables in the middle black region (Fig. C2b). The OSQP variant should in theory find the same numerical result than the with the CVXOPT one since both solvers solve the same dual problem (3). The difference between the two can be attributed to the fact that the OSQP solver seems less conservative on the nonnegativity constraint $\boldsymbol{\mu} \geq 0$ of (C5) than CVXOPT, accepting negligible violations. This seems to be advantageous to escape the local minimizer and to help finding a design that has a performance rather comparable to the one found by IPOPT in a similar number of iterations.

The significant discrepancy between the numerically computed designs illustrates the strong dependency on the optimization path chosen by the optimizer and on numerical round off errors. This also suggests some limitations to the standard implementation of the MMA algorithm, which has difficulty to converge on a case devoid of constraints other than the bound constraints on the design variable. It would be tempting to conclude that IPOPT performs better than the Null Space Optimizer, and that the Method of Moving Asymptotes is inappropriate, however, we have seen in the tutorial sections that this is of course very situation specific. Once again, we emphasize that the performance of an algorithm to find an optimum as close as possible to the global optimum is situation dependent and cannot constitute a single selection criterion between several optimizers for their use on more general situations.

Appendix D Statements and declarations

- The author has no relevant financial or non-financial interests to disclose.
- The author has no competing interests to declare that are relevant to the content of this article.
- The author certify that he has no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.
- The author has no financial or proprietary interests in any material discussed in this article.

Appendix E Reproducibility of numerical results

The integral source code corresponding to the examples presented in Sections 3 to 5 are included in the `examples/topopt_examples` directory of the GitLab repository of the Null Space Optimizer. Since the source code published on this repository is subject to future updates, a copy of the version used for the present manuscript is available as a supplementary material to this article.

Appendix F Acknowledgements

I sincerely thank Niels Aage for interesting discussions about the MMA algorithm, for suggesting the test case of Section 3.4.2 and sharing the solution of using the fixed initialization for the

Algorithm	J	Iteration number at convergence
NLSPACE (OSQP)	$-6.27600 \cdot 10^3$	65
NLSPACE (CVXOPT)	$-3.70000 \cdot 10^3$	65
MMA	$-1.07864 \cdot 10^0$	30
IPOPT	$-6.88800 \cdot 10^3$	58

Table C1: Final objective values for the checkerboard example of Appendix C.2. Comparison between the Null Space Optimizer using either CVXOPT or OSQP for solving the quadratic subproblem (3), MMA and IPOPT.

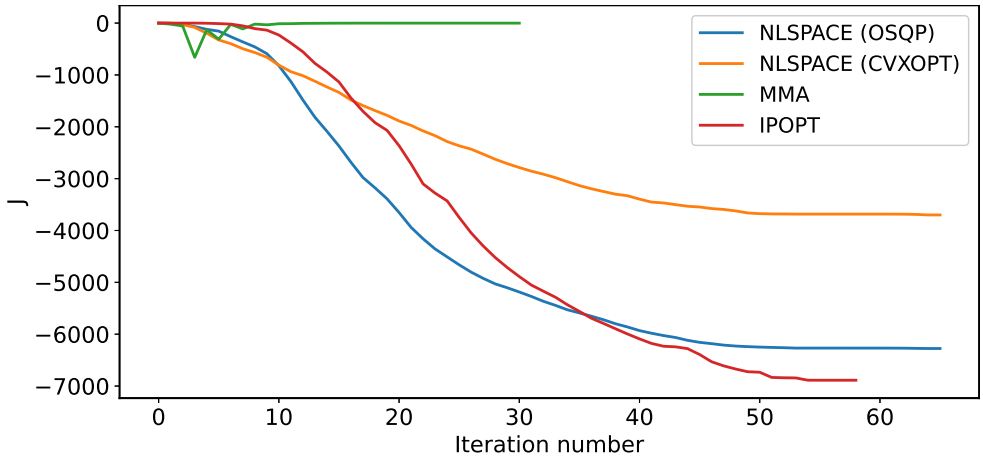


Fig. C1: Convergence history for the checkerboard example of Appendix C.2 for the three optimizers tested.

asymptotes. This solution came from discussions between N. Aage, Michael Stingl and Fabian Wein.

I also thank very much R. Krasniqi for his Master thesis work (Krasniqi (2023)) upon which part this work is built.

References

Aage N, Johansen VE (2016) A 200 line topology optimization code. https://www.topopt.mek.dtu.dk/-/media/subsites/topopt/apps/dokumenter-og-filer-til-apps/topopt_cholmod.py, visited on May 15th 2023

Aage N, Andreassen E, Lazarov BS, et al (2017) Giga-voxel computational morphogenesis for structural design. Nature 550(7674):84–86

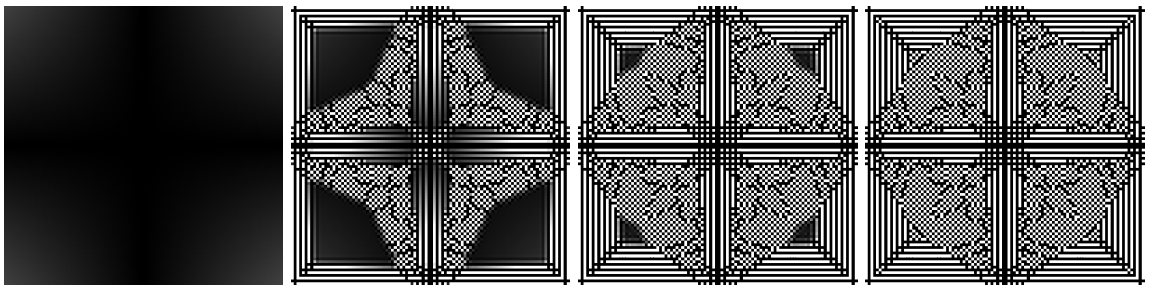
Absil PA, Malick J (2012) Projection-like Retractions on Matrix Manifolds. SIAM Journal on Optimization 22(1):135–158

Allaire G (2007) Numerical Analysis and Optimization. An Introduction to Mathematical Modelling and Numerical Simulation. Translation from the French by Alan Craig. Numer. Math. Sci. Comput., Oxford: Oxford University Press

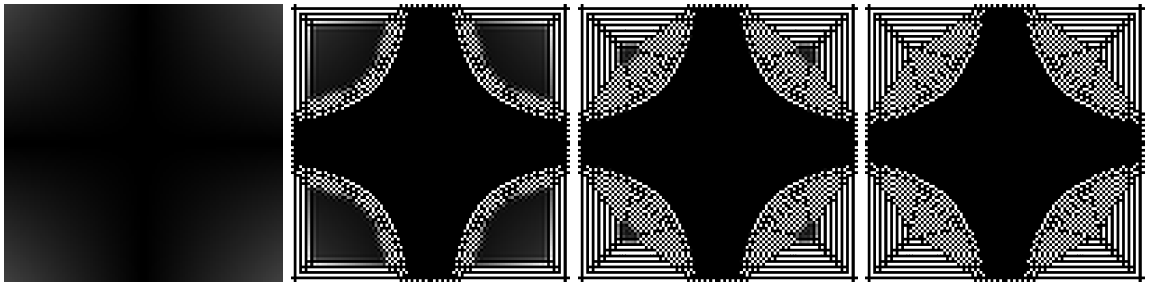
Allaire G, Jouve F, Michailidis G (2013) Casting constraints in structural optimization via a level-set method. In: 10th World Congress on Structural and Multidisciplinary Optimization

Allaire G, Dapogny C, Jouve F (2021) Chapter 1 - Shape and topology optimization. In: Bonito A, Nochetto RH (eds) Handbook of Numerical Analysis, Geometric Partial Differential Equations - Part II, vol 22. Elsevier, p 1–132

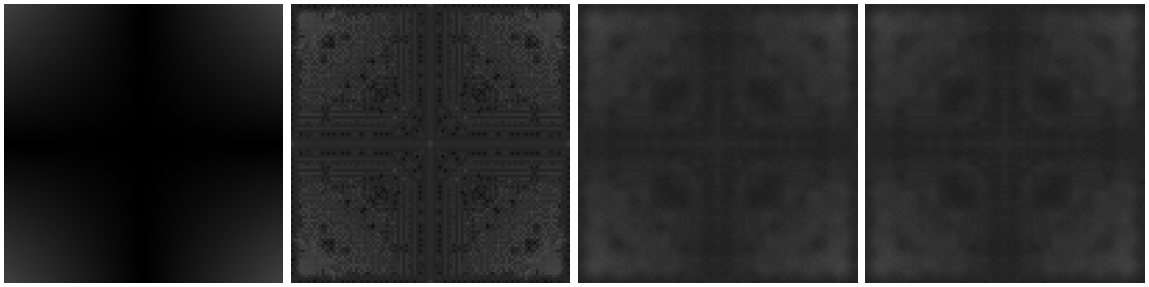
Alonso DH, Silva ECN (2021) Topology optimization for blood flow considering a hemolysis



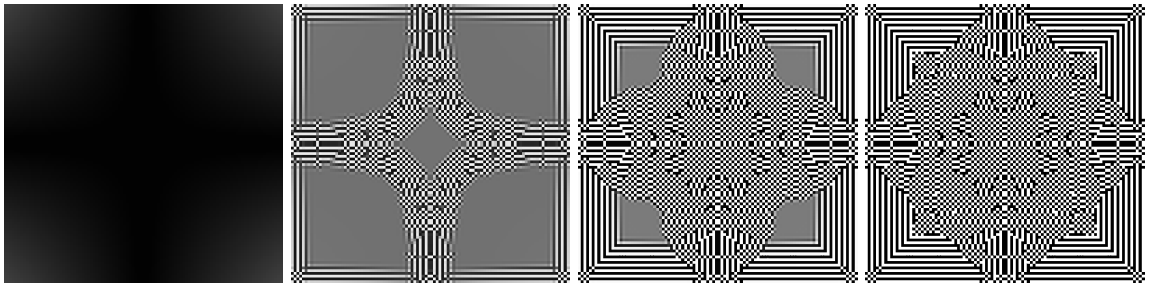
(a) Null Space (OSQP), iterations 0, 22, 45 and 68.



(b) Null Space (CVXOPT), iterations 0, 24, 49 and 74.



(c) MMA, iterations 0, 23, 46 and 70.



(d) IPOPT, iterations 0, 20, 41 and 62.

Fig. C2: Intermediate optimized designs for the various optimizers tested in the checkerboard example of Appendix C.2.

model. Structural and multidisciplinary optimization 63(5):2101–2123

Alonso DH, de Sá LFN, Saenz JSR, et al (2018) Topology optimization applied to the design of 2D swirl flow devices. Structural and Multidisciplinary Optimization 58(6):2341–2364

Andreassen E, Clausen A, Schevenels M, et al (2011) Efficient topology optimization in MATLAB using 88 lines of code. Structural and Multidisciplinary Optimization 43(1):1–16

Ashcraft C, Grimes RG, Lewis JG (1998) Accurate symmetric indefinite linear equation solvers. SIAM Journal on Matrix Analysis and Applications 20(2):513–561

Bai ZZ, Ng MK, Wang ZQ (2009) Constraint preconditioners for symmetric indefinite matrices. SIAM Journal on Matrix Analysis and Applications 31(2):410–433

Bejan A (1997) Constructal-theory network of conducting paths for cooling a heat generating volume. International Journal of Heat and Mass Transfer 40(4):799–816

Bendsoe MP, Sigmund O (2003) Topology Optimization: Theory, Methods, and Applications. Springer Science & Business Media

Brezis H (2011) Functional Analysis, Sobolev Spaces and Partial Differential Equations. Springer, New York, NY

Curtis FE, Huber J, Schenk O, et al (2012) A note on the implementation of an interior-point algorithm for nonlinear optimization with inexact step computations. Mathematical Programming 136(1):209–227

de Gournay F (2006) Velocity Extension for the Level-set Method and Multiple Eigenvalues in Shape Optimization. SIAM Journal on Control and Optimization 45(1):343–367. https://doi.org/10.1137/050624108

Deetman A (2020) GCMMA-MMA-Python. https://github.com/arjendeetman/GCMMA-MMA-Python

Deng S, Suresh K (2016) Multi-constrained 3D topology optimization via augmented topological level-set. Computers & Structures 170:1–12

Dunning PD, Kim HA (2015) Introducing the sequential linear programming level-set method for topology optimization. Structural and Multidisciplinary Optimization 51(3):631–643

Dunning PD, Stanford B, Kim HA (2015) Level-set topology optimization with aeroelastic constraints. In: 56th AIAA/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, p 1128

Emmendoerfer Jr H, Fancello EA, Silva ECN (2020) Stress-constrained level set topology optimization for compliant mechanisms. Computer Methods in Applied Mechanics and Engineering 362:112777

Fanni M, Shabara N, Alkalla M (2013) A Comparison Between Different Topology Optimization Methods. Engineering Journal 38

Feppon F (2019) Shape and topology optimization of multiphysics systems. These de doctorat, Université Paris-Saclay (ComUE)

Feppon F, Allaire G, Dapogny C (2020a) Null space gradient flows for constrained optimization with applications to shape optimization. ESAIM: Control, Optimisation and Calculus of Variations p 90

Feppon F, Allaire G, Dapogny C, et al (2020b) Topology optimization of thermal fluid–structure systems using body-fitted meshes and parallel computing. Journal of Computational Physics 417:109574

Feppon F, Allaire G, Dapogny C, et al (2021) Body-fitted topology optimization of 2D and 3D fluid-to-fluid heat exchangers. Computer Methods in Applied Mechanics and Engineering 376:113638

Freund RW, Nachtigal NM (1994) A new Krylov-subspace method for symmetric indefinite linear systems. Tech. Rep. ORNL/TM-12754, Oak Ridge National Lab. (ORNL), Oak Ridge, TN (United States)

Gao T, Zhang WH, Zhu JH, et al (2008) Topology optimization of heat conduction problem involving design-dependent heat load effect. Finite Elements in Analysis and Design 44(14):805–813

Gersborg-Hansen A, Sigmund O, Haber RB (2005) Topology optimization of channel flow problems. Structural and multidisciplinary optimization 30:181–192

Hecht F (2012) New development in FreeFem++. Journal of numerical mathematics 20(3-4):251–266

Jensen JS, Sigmund O (2011) Topology optimization for nano-photonics. Laser & Photonics Reviews 5(2):308–321

Kim NH, Dong T, Weinberg D, et al (2021) Generalized optimality criteria method for topology optimization. Applied Sciences 11(7):3175

Krasniqi R (2023) Density-based topology optimization with the Null Space optimiser. Master's thesis, KU Leuven

Lazarov BS, Sigmund O (2011) Filters in topology optimization based on Helmholtz-type differential equations. International Journal for Numerical Methods in Engineering 86(6):765–781

Liang J, Zhang X, Zhu B (2019) Nonlinear topology optimization of parallel-grasping microgripper. Precision Engineering 60:152–159

Liang K, Zhu D, Li F (2023) Macro–microscale topological design for compliant mechanisms with special mechanical properties. Computer Methods in Applied Mechanics and Engineering 408:115970

Liu P, Luo Y, Kang Z (2016) Multi-material topology optimization considering interface behavior via XFEM and level set method. Computer methods in applied mechanics and engineering 308:113–133

Liu P, Yan Y, Zhang X, et al (2021) Topological design of microstructures using periodic material-field series-expansion and gradient-free optimization algorithm. Materials & Design 199:109437

Liu X, Yuan Y (2010) A null-space primal-dual interior-point algorithm for nonlinear optimization with nice convergence properties. Mathematical programming 125(1):163–193

Marck G, Privat Y (2014) On some shape and topology optimization problems in conductive and convective heat transfers. In: OPTI 2014, An International Conference on Engineering and Applied Sciences Optimization, pp 1640–1657

Nie Py (2004) A null space method for solving system of equations. Applied Mathematics and computation 149(1):215–226

Nocedal J, Wright SJ (2006) Numerical optimization, 2nd edn. Springer Ser. Oper. Res. Financ. Eng., New York, NY: Springer

Rojas-Labanda S, Stolpe M (2015) Benchmarking optimization solvers for structural topology optimization. Structural and Multidisciplinary Optimization 52(3):527–547

Sá LF, Yamabe PV, Souza BC, et al (2021) Topology optimization of turbulent rotating flows using Spalart–Allmaras model. Computer Methods in Applied Mechanics and Engineering 373:113551

Salazar De Troya MA, Tortorelli DA, Beck VA (2021) Two Dimensional Topology Optimization of Heat Exchangers with the Density and Level-Set Methods. WCCM-ECCOMAS Congress 1300(LLNL-JRNL-816310)

Sigmund O (2001) A 99 line topology optimization code written in Matlab. Structural and Multidisciplinary Optimization 21(2):120–127

Sigmund O, Maute K (2013) Topology optimization approaches: A comparative review. Structural and Multidisciplinary Optimization 48(6):1031–1055

Sigmund O, Petersson J (1998) Numerical instabilities in topology optimization: A survey on

procedures dealing with checkerboards, mesh-dependencies and local minima. Structural optimization 16(1):68–75

Stellato B, Banjac G, Goulart P, et al (2020) OSQP: an operator splitting solver for quadratic programs. Mathematical Programming Computation 12(4):637–672

Svanberg K (1987) The method of moving asymptotes—a new method for structural optimization. International Journal for Numerical Methods in Engineering 24(2):359–373

Svanberg K (1993) The Method of Moving Asymptotes (MMA) with Some Extensions. In: Rozvany GIN (ed) Optimization of Large Structural Systems. NATO ASI Series, Springer Netherlands, Dordrecht, p 555–566

Svanberg K (2009) MMA and GCMMA Matlab code. http://www.smoptit.se/GCMMA-MMA-code-1.5.zip, visited on May 15th 2023

Svanberg K (2014) MMA and GCMMA – two methods for nonlinear optimization. Tech. rep.

Swartz KE, White DA, Tortorelli DA, et al (2021) Topology optimization of 3D photonic crystals with complete bandgaps. Optics Express 29(14):22170–22191

Toh KC, Phoon KK, Chan SH (2004) Block preconditioners for symmetric indefinite linear systems. International Journal for Numerical Methods in Engineering 60(8):1361–1381

Vandenberghe L (2010) The CVXOPT linear and quadratic cone program solvers. Online: http://cvxopt org/documentation/coneprog pdf

Wächter A, Biegler LT (2006) On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Mathematical Programming 106(1):25–57

Wang F, Lazarov BS, Sigmund O (2011) On projection methods, convergence and robust formulations in topology optimization. Structural and multidisciplinary optimization 43:767–784

Wang MY, Wang P (2006) The augmented Lagrangian method in structural shape and topology optimization with RBF based level set method. In: CJK-OSM 4: The Fourth China-Japan-Korea Joint Symposium on Optimization of Structural and Mechanical Systems, p 191

Xia Q, Wang MY (2008) Topology optimization of thermoelastic structures using level set method. Computational Mechanics 42:837–857

Yan S, Wang F, Sigmund O (2018) On the non-optimality of tree structures for heat conduction. International Journal of Heat and Mass Transfer 122:660–680. https://doi.org/10.1016/j.ijheatmasstransfer.2018.01.114

Yin L, Yang W (2001) Optimality criteria method for topology optimization under multiple constraints. Computers & Structures 79(20-21):1839–1850

Zhang Y, Liu S (2008) Design of conducting paths based on topology optimization. Heat and Mass Transfer 44(10):1217–1227

Zheng N, Zhai X, Chen F (2023) Topology Optimization of Self-supporting Porous Structures Based on Triply Periodic Minimal Surfaces. Computer-Aided Design p 103542