Topology optimization using PETSc: a Python wrapper and extended functionality

Smit, Thijs; Aage, Niels; Ferguson, Stephen J.; Helgason, Benedikt

Published in:
Structural and Multidisciplinary Optimization

Link to article, DOI:
10.1007/s00158-021-03018-7

Publication date:
2021

Document Version
Publisher's PDF, also known as Version of record

Structural and Multidisciplinary Optimization (2021) 64:4343–4353
https://doi.org/10.1007/s00158-021-03018-7

EDUCATIONAL PAPER

Topology optimization using PETSc: a Python wrapper and extended functionality

Thijs Smit[1] · **Niels Aage**[2] · **Stephen J. Ferguson**[1] · **Benedikt Helgason**[1]

Received: 9 February 2021 / Revised: 14 June 2021 / Accepted: 12 July 2021 / Published online: 14 September 2021

Abstract
This paper presents a Python wrapper and extended functionality of the parallel topology optimization framework introduced by Aage et al. (Topology optimization using PETSc: an easy-to-use, fully parallel, open source topology optimization framework. Struct Multidiscip Optim 51(3):565–572, 2015). The Python interface, which simplifies the problem definition, is intended to expand the potential user base and to ease the use of large-scale topology optimization for educational purposes. The functionality of the topology optimization framework is extended to include passive domains and local volume constraints among others, which contributes to its usability to real-world design applications. The functionality is demonstrated via the cantilever beam, bracket and torsion ball examples. Several tests are provided which can be used to verify the proper installation and for evaluating the performance of the user's system setup. The open-source code is available at https://github.com/thsmit/, repository `TopOpt_in_PETSc_wrapped_in_Python`.

Keywords Topology optimization · Parallel computing · Large scale · PETSc · Python C-Extension module · Python wrapper · 3D printing · Porous structures · Robust design · Infill

1 Introduction

The aim of this paper is to facilitate the use of large-scale topology optimization (TO) in education and to increase its adaption to real-world design applications. This is done by introducing a Python wrapper and by extending the functionality of the parallel TO framework introduced by Aage et al. (2015). The Python interface simplifies the problem definition and can therefore expand the potential user base to persons with limited, or no, experience with low-level programming and message passing interface (MPI). The Python wrapper also facilitates the use of the framework for educational purposes and the framework's integration into other Python projects. Furthermore, the functionality of the TO framework is extended which contributes to its usability to design real-world objects. For example, defining custom design domains, user-specific objective and constraint functions and multiple load cases is available as additional functionality. Moreover, a local volume constraint (Guest 2009) is included, as implemented by Wu et al. (2018), which allows for the design of porous structures.

Sigmund (2001) introduced a 2D TO code in Matlab, which has been widely adopted for teaching and as a starting point for research projects. Andreassen et al. (2011) improved the code in terms of efficiency and functionality, and recently Ferrari and Sigmund (2020) presented a new 99-line Matlab code with further improvements in speed and extension to 3D. 3D TO codes were introduced by Liu and Tovar (2014) and Amir et al. (2014) and apart from TO codes in Matlab, several Python codes are available, e.g. TUD (2013), Zuo and Xie (2015), Hunter (2017). The latter is increasingly relevant because of Python's open-source nature and the growing adoption of Python in teaching.

More recently, TO was increasingly used for designing real-world structures. Two factors contributing to the increased application of TO to real-world design problems are the development of 3D codes and the increased efficiency and parallelization of codes to enable finer discretizations. Examples of these high-performance computing TO codes

Responsible Editor: Emilio Carlos Nelli Silva

✉ Thijs Smit
thsmit@ethz.ch

[1] Institute for Biomechanics, ETH Zürich, Hönggerbergring 64, 8093 Zürich, Switzerland

[2] Solid Mechanics, Department of Mechanical Engineering, Technical University of Denmark, Nils Koppels Alle, B.404, 2800, Kgs. Lyngby, Denmark

are the 'Topology optimization using PETSc' framework (Aage et al. 2015), the parallel TO frameworks (Kennedy and Martins 2014; Chin et al. 2019; Wu et al. 2016; Liu et al. 2018) and the 'Topology optimization in OpenMDAO' framework (Chung et al. 2019). The framework by Aage et al. (2015) is state of the art, capable of handling ultra high-resolution problems with more than two billion elements (Amir et al. 2017; Baandrup et al. 2020) and uses the PETSc library to handle the parallel linear algebra (Balay et al. 1997, 2019, 2020).

Contributing to the successful widespread adoption of TO frameworks is the open-source policy. This enables others to use codes in their own research and educational projects. However, applying the frameworks to real-world practice requires modifications of the available codes, for example, changing the problem definition, incorporating multi-load cases, defining custom design domains, using passive rigid and solid regions, generating output files for 3D printing and performing body-fitted post-analysis in commercial Finite Element Analysis (FEA) software. These modifications require the knowledge of C++ and parallel computing. The technological threshold for these modifications by the end-user may limit the adoption of these frameworks in both practice and teaching.

In this work, we introduce a Python C-Extension module (Python wrapper), called `topoptlib`, for the 'Topology optimization using PETSC' framework (Aage et al. 2015). Furthermore, we propose a standardized workflow for real-world design applications and demonstrate the codes extended functionality and educational value via the cantilever beam, bracket and torsion ball examples. Several tests are provided which can be used to verify the proper installation of the framework and to check the performance of the user's system setup.

This work's `topoptlib` is available open-source (see Sect. 5). The following section presents the functionality extensions, explains the problem definition structure and the proposed workflow. Code snippets are provided to guide the user through the installation process and to run examples and tests. The functionality extensions are demonstrated by solving the cantilever beam, bracket and torsion ball examples. Finally, the performances of the Python wrapper and the original code are compared.

2 Functionality and usage

The original code, by Aage et al. (2015), is classified as a density approach to topology optimization using SIMP (Solid Isotropic Material with Penalization). The mesh is voxel-based and logical, i.e. structured. The Finite Element Model (FEM) uses a standard 8-node linear brick element. Regularization of the optimization is included through PDE (Lazarov and Sigmund 2010), sensitivity (Sigmund 1997) or density filters (Bourdin 2001). This avoids checkerboard patterns in the final design. The code uses the Method of Moving Asymptotes (MMA) as optimizer. For detailed information on SIMP, filters and MMA, the reader is referred to Sigmund and Bendsøe (2004), Sigmund (2007) and Svanberg (1987). The functionality of the original code is extended with the following features:

- STL (file format for storing surface geometry) input files to define the design domain, solid, void and rigid regions, and voxelization
- Exclusion of passive elements from the optimization
- Application of boundary conditions using parametrization functions
- Multi-load cases and multi-constraints
- User-defined objective and constraint functions
- Local volume constraint for 3D printing infill and bone-like structures
- Continuation strategy for the penalization value
- Robust design via three-field density projection
- Test scripts for code verification

The extended topology optimization problem, excluding the modified robust formulation, is mathematically described by

$$
\begin{aligned}
&\min_{\rho}\ \phi = \textstyle\sum_{i=1}^{L} \mathbf{F}_i^T \mathbf{u}_i \\
&\text{s.t.}\ \ \mathbf{K}(\hat{\rho})\mathbf{u}_i = \mathbf{F}_i, && \text{(Equilibrium equations)} \\
&\quad \tfrac{V(\hat{\rho})}{V_0 \cdot f} - 1 \leq 0.0, && \text{(Total volume constraint)} \\
&\quad \frac{(\frac{1}{n}\sum_e \overline{\rho}_e^p)^{\frac{1}{p}}}{\alpha} - 1.0 \leq 0.0, && \text{(Local volume constraint)} \\
&\quad 0 \leq \rho_e \leq 1 && \text{(Box constraint)}
\end{aligned}
$$

where ϕ is the sum of the compliances for L load cases. The optimization problem only includes active design variables, whereas the FEM problem is solved for all elements in the mesh. The equilibrium equations are included in the optimization formulation for completeness, however, the problem is solved using the commonly used nested approach which excludes the state field from the set of optimization variables. Furthermore, a total volume constraint is imposed where $V(\hat{\rho})$ is the volume of the current design, V_0 is the total volume and f is the prescribed volume fraction.

The local volume constraint, considering active design variables, n, restricts the amount of material locally as defined by Wu et al. (2018). α is the local volume fraction and $\overline{\rho_e}$ is a measure of the average density in the neighbourhood of voxel e, i.e.

$$
\overline{\rho_e} = \frac{\sum_{i \in \mathbb{N}_e} \hat{\rho}_i}{\sum_{i \in \mathbb{N}_e} 1}
$$

where $\mathbb{N}_e$ is the set of voxels surrounding voxel e, R_e is the influence radius, $\mathbf{x}_i$ and $\mathbf{x}_e$ are center coordinates of the voxels,

$$\mathbb{N}_e = \{i \mid ||\mathbf{x}_i - \mathbf{x}_e||_2 \leq R_e\}$$

p is usually chosen to be $p = 16$ and n is the number of active elements. The box constraint requires the mathematical design variables ρ_e to be continuous ($0 \leq \rho_e \leq 1$).

The mathematical design variables ρ are filtered with one of the available filters, as mentioned above, to produce the filtered design variables $\tilde{\rho}$. Subsequently, the projection is performed to ensure a solid/void design. This projection step produces the physical design variables $\hat{\rho}$ (Wang et al. 2011) with,

$$\hat{\rho}_e = \frac{\tanh(\beta\eta) + \tanh(\beta(\tilde{\rho}_e - \eta))}{\tanh(\beta\eta) + \tanh(\beta(1 - \eta))}$$

where β is the projection parameter that controls the aggressiveness of the projection. η is the threshold parameter. It is common to start the TO process with $\beta = 1$ and increase this number after a certain number of iterations. This process is called *beta continuation*. The filtering and projection processes can be visualized as $\rho \rightarrow \tilde{\rho} \rightarrow \hat{\rho}$.

The modified robust formulation is available based on the three-field density projection approach by Wang et al. (2011) which considers three sets of design variables; dilated $\hat{\rho}_{dil}$, intermediate $\hat{\rho}_{int}$ and the eroded design $\hat{\rho}_{ero}$. The three sets are formulated using the projection with thresholds of $\eta - \Delta$, η and $\eta + \Delta$, with Δ a user-specified parameter. The advantage of using the robust formulation lies in its ability to control the length scale. This means that the minimum feature size is controllable. The modified robust formulation, excluding the local volume constraint, is mathematically described by

$$\begin{aligned}
&\min_{\rho} \ \phi = \textstyle\sum_{i=1}^{L} \mathbf{F}_i^T \mathbf{u}_i \\
&\text{s.t} \quad \mathbf{K}(\hat{\rho}_{ero})\mathbf{u}_i = \mathbf{F}_i, && \text{(Equilibrium equations)} \\
&\quad \tfrac{V(\hat{\rho}_{dil})}{V_0 \cdot f} - 1 \leq 0.0, && \text{(Total volume constraint)} \\
&\quad 0 \leq \rho_e \leq 1 && \text{(Box constraint)}
\end{aligned}$$

The objective function and FEM problem are evaluated using the eroded design $\hat{\rho}_{ero}$. The global volume constraint is evaluated using the dilated design $\hat{\rho}_{dil}$. For every 20 iterations the volume fraction used in the global volume constraint is updated with the volume fraction of the intermediate design $\hat{\rho}_{int}$. Lazarov et al. (2016) describe this as the worst-case formulation, because it uses the most compliant design (the eroded) to calculate the objective, and the most bulky design (the dilated) to calculate the global volume constraint. For further reading on the modified robust formulation, the reader is referred to Wang et al. (2011).

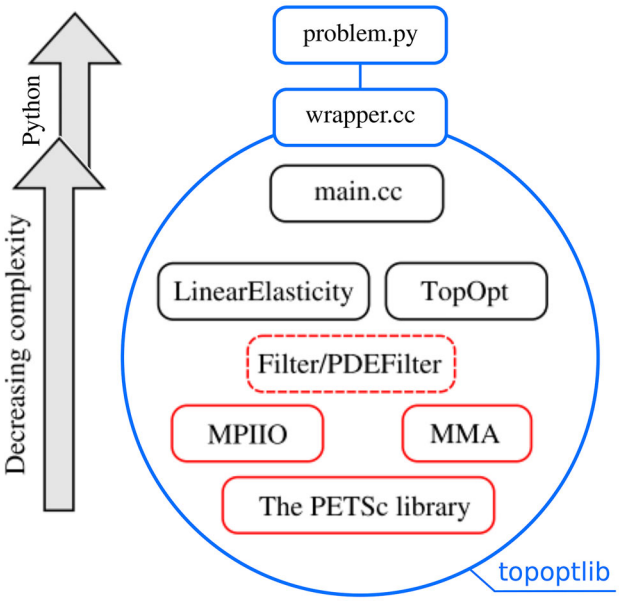

Fig. 1 The original code structure extended with the wrapper structure in blue. `topoptlib`. (Color figure online)

2.1 Code structure

The Python wrapper and functionality extensions are built on top of the existing code structure of the original code. Figure 1 shows the wrapper structure in blue added to the original code structure (Aage et al. 2015). The Python wrapper is a manually written C-Extension module. The wrapper is a C++ dynamic library named `topoptlib`. The `problem.py` file is the problem definition file in Python that imports the functionality of the framework. The `wrapper.cc` file contains the wrapping functions. Most of the classes are modified to facilitate the wrapper and extend the original code's functionality. The user of the wrapper should not need to modify the C++ code since the problem definition can be done with a Python script (`problem.py`).

2.2 Problem definition

A TO problem can be defined, either via the command line or in a script. The ease of use is still restrained because of the command line and scripting nature of the problem definition, rather than a graphical user interface. However, the authors believe that once this way of working is mastered, it is efficient and the user will appreciate the benefits.

The simplest method for running a problem is demonstrated here. The beam example, as defined in the original PETSc code by Aage et al. (2015), is pre-programmed as the default. The first step is to import the `topoptlib` module and initialize a data class. One can use the data class to store user-specific settings (explained later). Subsequently, the `solve`

function is called which will start the optimization loop with the settings stored in the data class.

```
import topoptlib
data = topoptlib.Data()
data.solve()
```

A problem can be setup with different settings by calling functions to load a particular functionality. As an example, mesh settings can be changed by using the `structuredGrid` function. The input of this function consists of 2 tuples, defining the domain boundaries and the mesh size. For the first tuple, the first 6 entries define the coordinates of the domain boundaries (x, y, z) and the additional entries can be used if needed. Furthermore, the material settings can be changed. The simulation is started by the following commands:

```
import topoptlib
data = topoptlib.Data()
data.structuredGrid((0.0, 2.0, 0.0, 1.0,
    0.0, 1.0), (129, 65, 65))
data.material(Emin, Emax, nu, Dens, penal)
data.solve()
```

Different filters, corresponding filter radius and MMA settings can be defined in the following manner:

```
# 0 is sensitivity filter
# 1 is density filter
# 2 is PDE filter
data.filter(1, rmin)
# (MaxIter, Tol)
data.mma(400, 0.01)
```

The user can define boundary conditions for multiple load cases. The number of load cases is defined with

```
data.loadcases(1)
```

Adding boundary conditions can be done by calling the boundary condition function. The first integer of this function indicates for which load case the boundary condition applies. The second integer is used to define a support (1) or forces (2). Then four lists are required. The first list defines the coordinate axis and the second list defines the coordinate on which to apply the boundary condition to. The third list indicates in which coordinate direction to apply the boundary condition. The fourth list defines the value. For a support, this value should be set to 0.0. The last integer of the function can be used to indicate a parametrization. With the parametrization function, a region can be specified, for which given boundary conditions apply. The parametrization is demonstrated in the `bracket.py` example. An example of applying a fully constrained plane (Fig. 2a) and a point load in the y direction (Fig. 2b) is

```
data.bc(0, 1, [2], [0.0] [0, 1, 2],
        [0.0, 0.0, 0.0], 0)
data.bc(0, 2, [0, 1, 2], [0.0, 0.0, 1.0],
        [1], [0.005], 0)
```

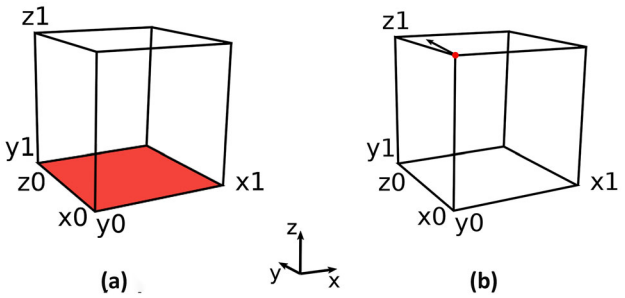

Fig. 2 Conceptual illustration of the application of boundary conditions. The domain is defined by $(x0, x1, y0, y1, z0, z1)$, with (0.0, 1.0, 0.0, 1.0, 0.0, 1.0). **a** A boundary condition applied on the $z = 0$ plane. **b** A force applied at the front corner node in y direction in blue

By default, the compliance is minimized with a global volume constrained, but the user can overwrite this by defining callback functions. User-specified objective and constraint functions will be called for every iteration. The sensitivity functions will be called for every iteration for every element. The `comp` denotes the total compliance value of the current design including the SIMP interpolation. The `sumXp` denotes the sum of the physical design variables $\hat{\rho}$. Here, `xp` refers to the physical design variable value and `uKu` is the strain energy of the current element. An example of the setup for minimizing volume with a target compliance is provided with the following mathematical description and code implementation:

$$
\begin{aligned}
&\min_{\rho} \ \phi = V(\hat{\rho}) \\
&\text{s.t.} \ \ \mathbf{K}(\hat{\rho})\mathbf{u} = \mathbf{F}, && \text{(Equilibrium equations)} \\
&\quad \frac{\mathbf{F}^T\mathbf{u}}{C_{ref}} - 1 \leq 0.0, && \text{(Target compliance)} \\
&\quad 0 \leq \rho_e \leq 1 && \text{(Box constraint)}
\end{aligned}
$$

with C_{ref} the target compliance value.

```
compliancetarget = 2.0
nEl = data.nElements
def objective(comp, sumXp, volfrac):
    return sumXp
def sensitivity(xp, uKu, penal):
    return 1.0
def constraint(comp, sumXp, volfrac):
    return comp / compliancetarget - 1.0
def constraintSensitivity(xp, uKu, penal):
    return (-1.0 * penal * np.power(xp, (
    penal - 1)) * (Emax - Emin) * uKu) /
    compliancetarget
```

The initial conditions can be changed with the following command:

```
# Homogenous initial condition
data.initialcondition(volumefraction)
```

The local volume constraint can be switched on. This will initialize a multi-constrained problem by adding the local

volume constraint as an additional constraint to the global volume constraint.

```
# Local volume constraint: (Rlocvol, alpha)
data.localVolume(0.16, 0.12)
```

The projection can be used as was described earlier, but is optional. Function inputs are the final and initial β and the threshold parameter η.

```
# Projection: (betaFinal, betaInitial, eta)
data.projection(64.0, 1.0, 0.5)
```

When control over the minimum length scale is required, the modified robust formulation can be used with the following command:

```
# Robust formulation:
# (betaFinal, betaInitial, eta)
data.robust(64.0, 1.0, 0.5)
```

2.3 Installation

The framework should be compiled once on a cluster or a desktop computer prior to first use. A problem file can call the functions of the framework without compiling thereafter. Setting up the framework on a Linux operating system is recommended. Testing was performed on the ETH Euler cluster and a desktop computer with Ubuntu version 20.04. Installation on a Windows machine should be possible; however, this has not been tested. PETSc version 3.14.1 (https://www.mcs.anl.gov/petsc/mirror/release-snapshots/), version control system Git (https://git-scm.com/) and Python 3.7 (https://www.python.org/) should be installed on the system. CMake version 3.10 (https://cmake.org) is used to compile the framework. To install PETSc (note that the file paths will differ), the following commands need to be executed:

```
cd petsc-3.14.1
./configure --with-cc=mpicc --with-cxx=
    mpicxx --with-fc=0 --download-
    f2cblaslapack=1 --with-debugging=0
make PETSC_DIR=/home/ts/Documents/petsc
    PETSC_ARCH=arch-linux-c-opt all
make PETSC_DIR=/home/ts/Documents/petsc
    PETSC_ARCH=arch-linux-c-opt check
```

The next step is to download the framework:

```
git clone https://github.com/thsmit/
    TopOpt_in_PETSc_wrapped_in_Python.git
```

The wrapper packs the framework into a C++ dynamic library to make it available for import into Python. PETSc needs to be available when the wrapper is compiled. This can be done by using the `export` command. To compile the wrapper and pack it in `topoptlib.so`, the following sequence of commands needs to be executed (file paths will differ):

```
export PETSC_ARCH=arch-linux-c-opt
export PETSC_DIR=/home/ts/Documents/petsc
cd TopOpt_in_PETSc_wrapped_in_Python
mkdir build
cd build
cmake .. -D PETSC_EXECUTABLE_RUNS=ON
make
```

2.4 Running and post-processing

Defining and running an optimization problem is most easily done with the help of a Python script. Examples can be found in the `examples` folder.

To run the cantilever beam example using one core:

```
cd TopOpt_in_PETSc_wrapped_in_Python
cp examples/beam.py .
python3 beam.py
```

Several test cases are provided to verify that the framework is working properly. To run the beam problem test using 4 cores, execute the following sequence of commands:

```
cd TopOpt_in_PETSc_wrapped_in_Python
cp topoptlib/test/test_beam.py .
mpirun -n 4 python3 test_beam.py
```

The results of the optimization process can be written to a vtr file and viewed in Paraview (https://www.paraview.org/) (Schroeder et al. 1996). The first ten designs are output by default. The user can choose with which frequency output should be generated after the first ten iterations. To generate vtr files from the designs, use (in the problem definition file)

```
data.vtr(1) # generate vtr files
```

Two bash scripts are added to the open-source repository: `run_topopt.sh` and `test_topopt.sh`. These scripts can be used to run examples and tests scripts. The user can edit them as needed.

2.5 Workflow

One of the main motivations for this work is to lower the barrier for using the unique capabilities of high-resolution TO in teaching and in real-world design applications. Figure 3 illustrates the workflow using the bracket example. This work flow includes STL file input, TO design process, smoothing of the design, 'ready-to-3D-print' output files and final design verification by an external FEM package. For details on the bracket example, see Sect. 3.2.

The design domain, void, rigid and solid regions can be defined via STL files (Fig. 3a). It is recommended that the STL input file is "water tight" and defect free. The following command voxelizes the design domain for the bracket example (paths will differ):

```
# STL read and voxelize: (encoding, back
    round, threshold, box around STL: (min
    corner)(max corner), full path to file)
data.stlread(
    -1.0,
    1.0,
    8,
    (-23.0, -1.0, -103.0),
    (169.0, 63.0, 1.0),
    "/cluster/home/thsmit/
    TopOpt_in_PETSc_wrapped_in_Python/stl/
    jetEngineDesignDomainFine.stl",
)
```

The design domain defines the region with active design variables. The function, `stlread`, needs several inputs. The first input, named encoding, determines the function of the cells within the region defined by the STL file: passive, active, rigid, solid or void. The encoding is defined as follows:

```
# Passive elements: 1.0
# Active elements: -1.0
# Solid elements: 2.0
# Rigid elements: 3.0
# Void elements: 4.0
# Do not overwrite: 0.0
```

The rigid regions are regions with stiffness significantly higher than the material stiffness so that loads can be applied and distributed. The passive, solid, void and rigid regions are not part of the design domain.

The resulting voxelization of the STL input file is shown in Fig. 3b. The problem definition is done in Python and interfaces with the underlying TO code for running the TO problem, see `bracket.py` for details. Results are written to vtr files for visualization using Paraview (Fig. 3c).

In practice, high-resolution TO produces already a nearly smooth surface. Therefore, it can be satisfactory to generate an STL file from the TO output directly (using Paraview filters: 3D and Extract surface). It is recommended to output the STL file in an ASCII format. This STL file can be used for 3D printing (Fig. 3d). However, a smoothing step can also be added (Paraview filter: Smooth), see Fig. 3e. The smooth design can be 3D printed (Fig. 3f). Smoothing of the final design will change the geometry of the object slightly. Therefore, a FEM analysis might be required in order to verify the final smooth design. The workflow proposed in this work avoids the transformation of the surface representation (STL file) of the design into a NURBS-based solid representation (STEP file). A FEM mesh with solid tetrahedral elements is generated from the smooth design using Tetgen (Si 2015), see Fig. 3g. Obtaining a solid tetrahedral mesh for complex designs might fail. Therefore, this method is not generally applicable. The following Tetgen command can be used:

```
./tetgen -p -k bracket_smooth.stl
```

Note that with an open-source application like `meshio` the Tetgen mesh can be converted to an input file for many different FEM solvers. In this case, we use Abaqus version 6.14 to analyse the final smooth design (Fig. 3h).

3 Examples

The wrapper and extended functionality is demonstrated by solving three TO problems. The examples are computed on a cluster (ETH Euler) using 32 cores, each with access to 4GB of memory. The local volume constraint and multi-loading extensions are demonstrated using the standard cantilever beam problem. The problem is solved for minimizing compliance and minimizing volume, respectively. The STL input and voxelization functionality is demonstrated by the bracket and torsion ball examples.

3.1 Cantilever beam

The cantilever beam problem is solved on a $2 \times 1 \times 1$ domain, using a $128 \times 64 \times 64$ mesh with about $5.2 \cdot 10^5$ design variables and about $1.6 \cdot 10^6$ state DOFs. We use a density filter with a radius of 0.08. For the total volume constraint, a volume fraction of 12% is used. The boundary conditions are illustrated in Fig. 4a. Figure 4b shows the standard, single-load, cantilever beam. All problem settings are available in `beam.py` and the problem can be run using the bash script:

```
./run_topopt.sh beam.py
```

Figure 4c shows the result for the beam when the volume is minimized with a target compliance. To change the objective and sensitivity functions for minimizing volume, see Sect. 2.2. A compliance target of 1.5 results in a volume fraction of 14.7%.

Figure 4d shows the result when the beam is optimized with a local volume constraint instead of the total volume constraint, using an averaging filter radius of 0.16. This defines which cells are under control of the local volume constraint. A local volume fraction of 12% is used, meaning that the volume within the local volume filter radius for every $\overline{\rho}_e$ should not be more than 12%. The resulting beam consists primarily of vertical members. This is because the principal strains are generally aligned with the x-y plane. The figure is made partly transparent to better illustrate this.

In Fig. 4e, a total volume constraint with 12% volume fraction is introduced back into the problem, together with the local volume constraint. This clearly restricts the total amount of material used, but still the influence of the local volume constraint is visible if we compare the design with Fig. 4b. The local volume constraint prevents the material from bulking into large struts, but forming a more evenly distributed porous network instead.

For the beam in Fig. 4f, a multi-load case is introduced. The total volume constraint is set to 24%. Interestingly, a

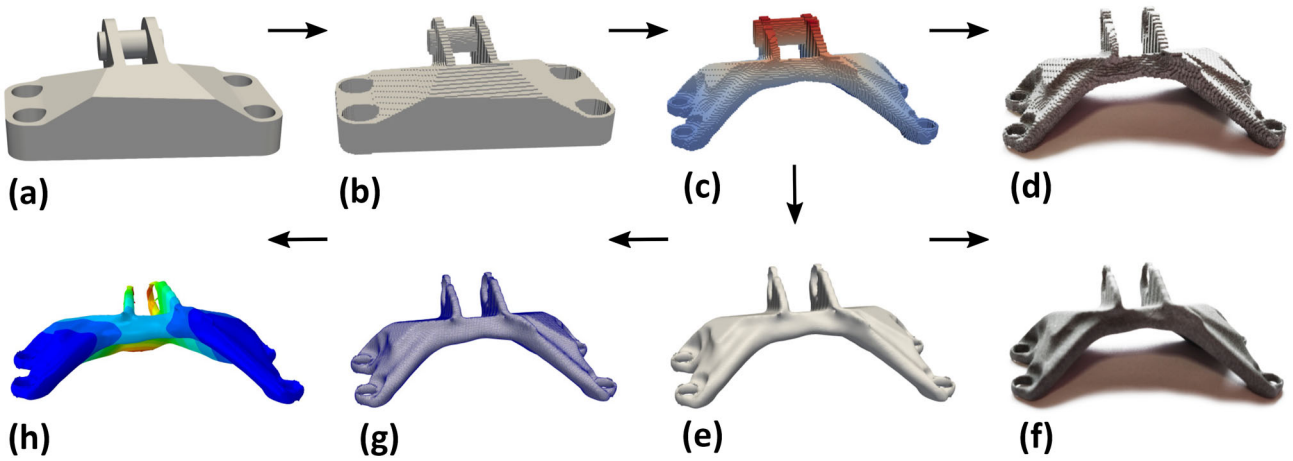

Fig. 3 Workflow: from idea to 3D print, **a** STL input file for design domain reading, **b** voxelization, **c** computational design by Topology optimization and vtr generation for visualization (displacement field is visualized in plot), **d** STL file generation and 3D printing of the non-smooth design, **e** surface smoothing, **f** STL file generation and 3D printing of the smooth design, **g** mesh generation of the smooth design, **h** FEM analysis of final smooth design for verification

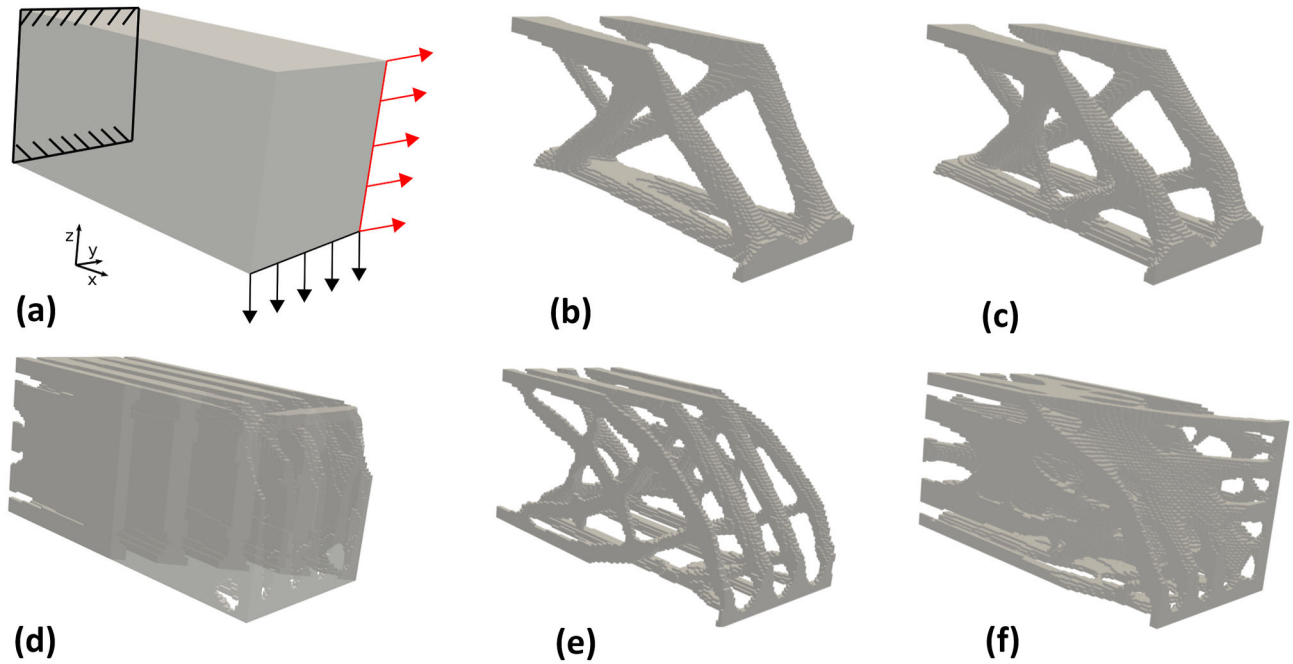

Fig. 4 Comparison of optimized cantilever beam problems, with the plots showing all elements with $\hat{\rho} > 0.5$ **a** problem definition for the cantilever beam, with the standard loading in black, **b** standard beam with a compliance of 1.962, **c** beam for minimizing volume, **d** beam with local volume constrained with a compliance of 0.999, but a total volume fraction of 34.5%, **e** beam with local and total volume constraint with a compliance of 3.875, **f** beam with local and total volume constraint and multiloads (red loading included)

porous structure is formed with nearly equally sized pores. Furthermore, the local volume constraint has a strong influence on the size of the struts and tuning the local volume constraint can be used to influence the strut sizes in the final design.

3.2 Bracket

The bracket example was previously solved as part of the GrabCAD design challenge in 2013 and later by Amir et al. (2017). The bracket design problem is in fact becoming the new standard test example for 3D TO codes, as was the

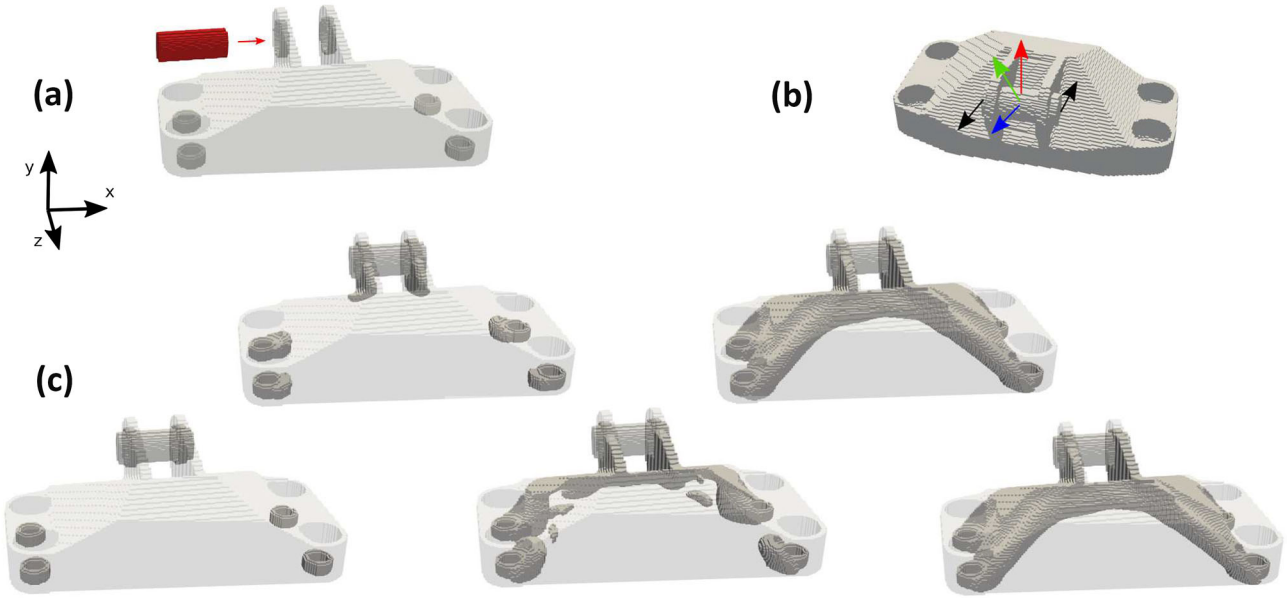

Fig. 5 Bracket example, with the plots showing all elements with $\hat{\rho} > 0.5$ **a** voxelization with the design domain in light grey, the solid elements in dark grey and the rigid elements in red, **b** definition of the multi-load cases, **c** showing the progression of the design through the TO process

case for the MBB beam in 2D. Three STL files are used as input to define the design domain (active elements), solid elements (relative density of 1) and rigid elements (with stiffness $E^* = 10^3 E$). The framework voxelizes the STL files and integrates the domains into a $192 \times 64 \times 104$ mesh with about $4.1 \cdot 10^5$ active designs DOFs (Fig. 5a). A density filter is applied to avoid checker boarding. A total volume constraint is prescribed with the volume fraction set to 30%. The rigid part is used to apply and distribute 4 load cases (Fig. 5b). Figure 5c shows the evolution of the 291 step design process. The problem settings are available in `bracket.py`. For running the problem use

```
./run_topopt.sh bracket.py
```

Two aspects of the final bracket design are particularly interesting and advocate for the increased applicability of TO in mechanical engineering design. First, a large weight reduction can be achieved. This is of interest when designing for industries like, aerospace, automotive and semiconductor equipment. Second, organic and round shapes emerge, avoiding sharp corners. This indirectly minimizes stress concentrations and contributes to improved fatigue resistance.

3.3 Torsion ball

The torsion ball example, described by Sigmund et al. (2016), is also solved for demonstrating the capabilities of the framework. The design domain is reduced to 1/8 part using symmetry boundary conditions (Fig. 6a). The loads are applied on the rigid part and anti-symmetry boundary conditions are used, restricting the displacements in the directions as indicated in Fig. 6b. A projection method is applied to converge to a solid/void design (Wang et al. 2011). The 1/8 domain is discretized by a $160 \times 192 \times 192$ mesh which results in about $5.8 \cdot 10^6$ active elements. All problem settings can be found in `sphere.py` and the problem can be run by executing

```
./run_topopt.sh sphere.py
```

The result of the optimization process is shown in Fig. 6c. Restoring the symmetries results in a fully closed sphere (Fig. 6d). Next step is to use the modified robust formulation with a three-field density projection as defined by (Wang et al. 2011). The same problem parameters, as described by (Sigmund et al. 2016) were used. This results in the Michell sphere as seen in Fig. 6e and f.

4 Performance comparison

The performances of the Python wrapper and the original code are compared, using the standard cantilever beam problem (`beam.py`). Both codes are run for three mesh sizes, $64 \times 32 \times 32$, $128 \times 64 \times 64$ and $256 \times 128 \times 128$. A density filter is used with a filter radius of 0.08, 0.04 and 0.02, respectively. The problems are run on two different platforms: a desktop computer (Ubuntu 20.04) and a cluster (ETH Euler). The installation process, described in Sect. 2.3, is used on both platforms. For the smallest mesh, 4 cores are used, and for the other meshes 32 cores. Due to memory requirements, the larger meshes could not be run on the desktop. The total CPU time and memory use are reported in

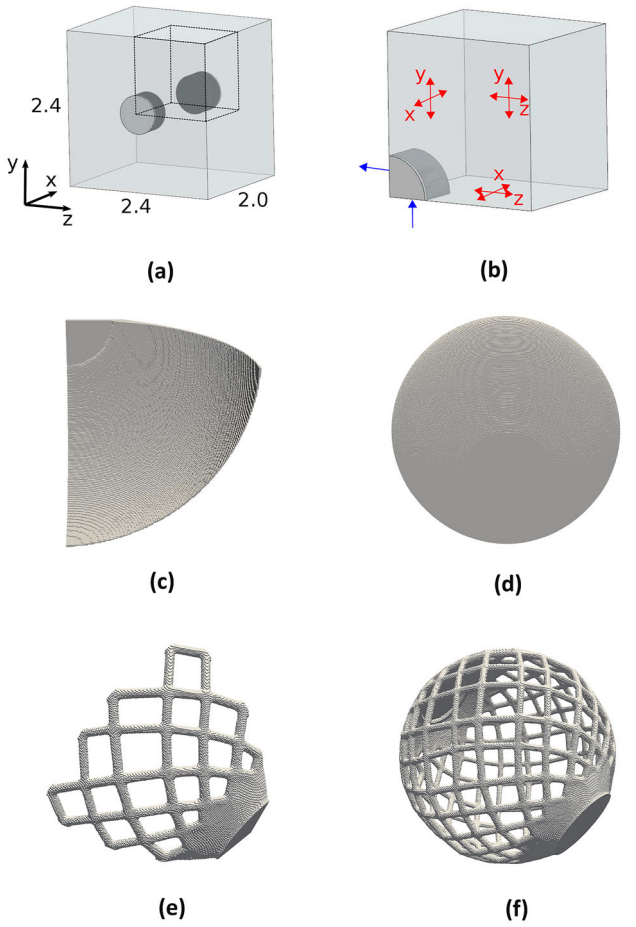

Fig. 6 Torsion ball example, with the designs c, d e and f showing all elements with $\hat{\rho} > 0.5$ **a** full design domain, with the 1/8 symmetry domain indicated, **b** 1/8 domain, with loads in blue, and anti-symmetry boundary conditions applied at the three symmetry planes. Light grey indicates the rigid part, with a radius of 0.3 and thickness of 0.025. Dark grey regions indicate the void or passive part of the domain with a radius of 0.3 and thickness 0.3, **c** resulting 1/8 of a sphere, **d** resulting full sphere after restoring the symmetries, **e** resulting 1/8 of the Michell sphere, **f** Michell sphere after restoring the symmetries

Table 1 Comparing CPU time

CPU time comparison				
Mesh	nC		Cluster (s)	Desktop (s)
64 × 32 × 32	4	`Wrap`	3036	5879
		`Orig`	2816	5864
128 × 64 × 64	32	`Wrap`	37,199	–
		`Orig`	34,827	–
256 × 128 × 128	32	`Wrap`	341,071	–
		`Orig`	340,713	–

Table 2 Comparing memory use

Memory use comparison				
Mesh	nC		Cluster (MB)	Desktop (MB)
64 × 32 × 32	4	`Wrap`	1122	989
		`Orig`	1212	865
128 × 64 × 64	32	`Wrap`	11,627	–
		`Orig`	10,437	–
256 × 128 × 128	32	`Wrap`	58,163	–
		`Orig`	56,491	–

Tables 1 and 2 with the wrapper code denoted as `Wrap` and the original code denoted as `Orig`

The user can test their own setup by running the `beam.py` script. This will display the total CPU time, the memory use and the final compliance value for comparison. It should be noted that different hardware setup will result in different performance numbers, but by comparing with Tables 1 and 2 the user can get an indication of the relative performance of their setup. For further reading on performance comparisons of topology optimization codes, the reader is referred to Aage and Lazarov (2013) on parallel computing and Ferrari and Sigmund (2020) on the performance of TO Matlab code.

The true compliance values between `Wrap` and `Orig` for the different meshes are compared and are found to be identical up to 3 decimals. The compliance values for the three meshes, with increasing size are 0.492, 1.395 and 4.914, respectively. Looking at the CPU time and memory use between the two codes it is clear that the ease of use of the wrapper comes with a decrease in performance. For the wrapper, the CPU time increases by 7.8% and the memory use increases by 14.3%, compared to the original code. However, for the problem run on the desktop, solving a similar problem on commercially available FEA software (Abaqus version 6.14) requires about 2.5 times more memory and about an order of magnitude more CPU time.

5 Conclusion

Making the 'Topology Optimization using PETSc' framework available in Python means that a larger user group gains access to a state-of-the-art large-scale (high-resolution) topology optimization code. Extending the functionality increases the framework's applicability to real-world 3D design applications which can be 3D printed easily and validated using the standardized workflow. It is the hope of the authors that the open-source code will accelerate research, teaching and application of large-scale (high-resolution) topology optimization across different fields. The user should be able to get familiar with the code by using the examples and tests. Furthermore, the tests can be run to check the functionality and performance of the code. The code is open-source and the authors encourage further use, extension and collaboration. This could involve, e.g. increasing its applica-

bility to real-world design applications, by implementation of anisotropic local volume filters and stress constraints. The anisotropic local volume filter implementation is straightforward and requires little programming in C++, where the stress constraints are more involved. The wrapper does not limit any changes to the C++ and the wrapper can be modified to accommodate the added functionality.

Acknowledgements Open Access funding provided by ETH Zurich. This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie Grant Agreement No. 812765. Niels Aage was supported by the Villum Investigator Project InnoTop funded by the Villum Foundation.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Replication of results The open-source code is available at https://github.com/thsmit/ under repository `TopOpt_in_PETSc_wrapped_in_Python`. The authors are welcoming contributions towards expanding the functionality of the framework.

References

Aage N, Lazarov BS (2013) Parallel framework for topology optimization using the method of moving asymptotes. Struct Multidiscip Optim 47(4):493–505. https://doi.org/10.1007/s00158-012-0869-2

Aage N, Andreassen E, Lazarov BS (2015) Topology optimization using PETSc: an easy-to-use, fully parallel, open source topology optimization framework. Struct Multidiscip Optim 51(3):565–572. https://doi.org/10.1007/s00158-014-1157-0

Amir O, Aage N, Lazarov BS (2014) On multigrid-CG for efficient topology optimization. Struct Multidiscip Optim 49(5):815–829. https://doi.org/10.1007/s00158-013-1015-5

Aage N, Andreassen E, Lazarov BS, Sigmund O (2017) Gigavoxel computational morphogenesis for structural design. Nature 550(7674):84–86. https://doi.org/10.1038/nature23911

Andreassen E, Clausen A, Schevenels M, Lazarov BS, Sigmund O (2011) Efficient topology optimization in MATLAB using 88 lines of code. Struct Multidiscip Optim 43(1):1–16. https://doi.org/10.1007/s00158-010-0594-7

Baandrup M, Sigmund O, Polk H, Aage N (2020) Closing the gap towards super-long suspension bridges using computational morphogenesis. Nat Commun 11(1):1–7. https://doi.org/10.1038/s41467-020-16599-6

Balay S, Gropp WD, McInnes LC, Smith BF (1997) Efficient management of parallelism in object oriented numerical software libraries. In: Bruaset AM, Langtangen HP, Arge E (eds) Modern Software Tools in Scientific Computing. Birkhäuser Press, Basel, pp 163–202

Balay S, Abhyankar S, Adams MF, Brown J, Brune P, Buschelman K, Dalcin L, Dener A, Eijkhout V, Gropp WD, Karpeyev D, Kaushik D, Knepley MG, May DA, McInnes LC, Mills RT, Munson T, Rupp K, Sanan P, Smith BF, Zampini S, Zhang H, Zhang H (2019) PETSc Web page. https://www.mcs.anl.gov/petsc

Balay S, Abhyankar S, Adams MF, Brown J, Brune P, Buschelman K, Dalcin L, Dener A, Eijkhout V, Gropp WD, Karpeyev D, Kaushik D, Knepley MG, May DA, McInnes LC, Mills RT, Munson T, Rupp K, Sanan P, Smith BF, Zampini S, Zhang H, Zhang H (2020) PETSc users manual. Tech. Rep. ANL-95/11—Revision 3.14, Argonne National Laboratory

Bourdin B (2001) Filters in topology optimization. Int J Numer Meth Eng 50(9):2143–2158. https://doi.org/10.1002/nme.116

Chin TW, Leader MK, Kennedy GJ (2019) A scalable framework for large-scale 3D multimaterial topology optimization with octree-based mesh adaptation. Adv Eng Softw 135:102682. https://doi.org/10.1016/j.advengsoft.2019.05.004

Chung H, Hwang JT, Gray JS, Kim HA (2019) Topology optimization in OpenMDAO. Struct Multidiscip Optim 59(4):1385–1400. https://doi.org/10.1007/s00158-019-02209-7

Ferrari F, Sigmund O (2020) A new generation 99 line Matlab code for compliance topology optimization and its extension to 3D. Struct Multidiscip Optim 62(4):2211–2228. https://doi.org/10.1007/s00158-020-02629-w

Guest JK (2009) Imposing maximum length scale in topology optimization. Struct Multidiscip Optim 37(5):463–473. https://doi.org/10.1007/s00158-008-0250-7

Hunter W (2017)Others ToPy - Topology optimization with Python.https://github.com/williamhunter/topy

Kennedy GJ, Martins JR (2014) A parallel finite-element framework for large-scale gradient-based design optimization of high-performance structures. Finite Elem Anal Des 87:56–73. https://doi.org/10.1016/j.finel.2014.04.011

Lazarov BS, Sigmund O (2010) Filters in topology optimization based on Helmholtz-type differential equations. Int J Numer Method Eng. https://doi.org/10.1002/nme.3072

Lazarov BS, Wang F, Sigmund O (2016) Length scale and manufacturability in density-based topology optimization. Arch Appl Mech 86(1–2):189–218. https://doi.org/10.1007/s00419-015-1106-4

Liu K, Tovar A (2014) An efficient 3D topology optimization code written in Matlab. Struct Multidiscip Optim 50(6):1175–1196. https://doi.org/10.1007/s00158-014-1107-x

Liu H, Hu Y, Zhu B, Matusik W, Sifakis E (2018) Narrow-band topology optimization on a sparsely populated grid. SIGGRAPH Asia 2018 Technical Papers. SIGGRAPH Asia 37(6):1–14. https://doi.org/10.1145/3272127.3275012

Schroeder W, Martin K, Lorensen B (1996) The visualization toolkit: an object oriented approach to 3D graphics, 3rd edition—Kitware Inc.pdf. J Aust Entomol Soc 34:335–342

Si H (2015) TetGen, a delaunay-based quality tetrahedral mesh generator. ACM Trans Math Softw. https://doi.org/10.1145/2629697

Sigmund O (1997) On the design of compliant mechanisms using topology optimization. Mech Struct Mach 25(4):493–524. https://doi.org/10.1080/08905459708945415

Sigmund O (2001) A 99 line topology optimization code written in matlab. Struct Multidiscip Optim 21(2):120–127. https://doi.org/10.1007/s001580050176

Sigmund O (2007) Morphology-based black and white filters for topology optimization. Struct Multidiscip Optim 33(4–5):401–424. https://doi.org/10.1007/s00158-006-0087-x

Sigmund O, Bendsøe MP (2004) Topology Optimization. Springer, Berlin, Heidelberg

Sigmund O, Aage N, Andreassen E (2016) On the (non-)optimality of Michell structures. Struct Multidiscip Optim 54(2):361–373. https://doi.org/10.1007/s00158-016-1420-7

Svanberg K (1987) The method of moving asymptotes—a new method for structural optimization. Int J Numer Meth Eng 24(2):359–373. https://doi.org/10.1002/nme.1620240207

TUD (2013) Topology optimization codes written in Python. DTU Mechanical Engineering and DTU Compute

Wang F, Lazarov BS, Sigmund O (2011) On projection methods, convergence and robust formulations in topology optimization. Struct Multidiscip Optim 43(6):767–784. https://doi.org/10.1007/s00158-010-0602-y

Wu J, Dick C, Westermann R (2016) A system for high-resolution topology optimization. IEEE Trans Visual Comput Graphics 22(3):1195–1208. https://doi.org/10.1109/TVCG.2015.2502588

Wu J, Aage N, Westermann R, Sigmund O (2018) Infill optimization for additive manufacturing-approaching bone-like porous structures. IEEE Trans Visual Comput Graph 24(2):1127–1140. https://doi.org/10.1109/TVCG.2017.2655523

Zuo ZH, Xie YM (2015) A simple and compact Python code for complex 3D topology optimization. Adv Eng Softw 85:1–11. https://doi.org/10.1016/j.advengsoft.2015.02.006