A new generation 99 line Matlab code for compliance Topology Optimization and its extension to 3D

Federico Ferrari · Ole Sigmund

Received: date / Accepted: date

Abstract Compact and efficient Matlab implementations of compliance Topology Optimization (TO) for 2D and 3D continua are given, consisting of 99 and 125 lines respectively. On discretizations ranging from $3 \cdot 10^4$ to $4.8 \cdot 10^5$ elements, the 2D version, named top99neo, shows speedups from 2.55 to 5.5 times compared to the well-known top88 code (Andreassen et al., 2011). The 3D version, named top3D125, is the most compact and efficient Matlab implementation for 3D TO to date, showing a speedup of 1.9 times compared to the code of Amir et al. (2014), on a discretization with $2.2 \cdot 10^5$ elements. For both codes, improvements are due to much more efficient procedures for the assembly and implementation of filters and shortcuts in the design update step. The use of an acceleration strategy, yielding major cuts in the overall computational time, is also discussed, stressing its easy integration within the basic codes.

Keywords Topology optimization · Matlab · Computational efficiency · Acceleration methods

1 Introduction

The celebrated top99 Matlab code developed by Sigmund (2001) has certainly promoted the spreading of Topology Optimization among engineers and researchers, and the speedups carried by its heir, top88 (Andreassen et al., 2011), substantially increased the scale of examples that can be solved on a laptop.

On these footprints, several other codes have followed, involving extension to 3D problems (Liu and Tovar, 2014; Amir et al., 2014), material design (Andreassen and Andreassen, 2014; Xia and Breitkopf, 2015), level-set parametrizations (Wang, 2007; Challis, 2010), use of advanced discretization techniques (Talischi et al., 2012; Suresh, 2010; Sanders et al., 2018), or integration of TO within some finite element frameworks.

With the evolution of TO and its application to more and more challenging problems, implementations in top88 may have become outdated. Also Matlab has improved in the last decade. Hence, we believe it is time to present a new “exemplary” code collecting shortcuts and speedups, allowing to tackle medium/large-scale TO problems efficiently on a laptop. Preconditioned iterative solvers, applied for example in (Amir and Sigmund, 2011; Amir et al., 2014) and (Ferrari et al., 2018; Ferrari and Sigmund, 2020) allow the solution of the state equation with nearly optimal efficiency (Saad, 1992). Thus, the computational bottleneck has been shifted on other operations, such as the matrix assembly or the repeated application of filters. Efficiency improvements for these operations were touched upon by Andreassen et al. (2011), however without giving a quantitative analysis about time and memory savings.

Here we provide compact Matlab codes for minimum compliance Topology Optimization of 2D and 3D continua which show a substantial speedup compared to the top88 code. We include several extensions by default, such as specification of passive domains, a volume-preserving density projection (Guest et al., 2004; Wang et al., 2011) and continuation strategies for the penalization and projection parameters in a very compact, yet sharp, implementation. Coincidentally, the new 2D TO implementation consist of 99 lines of code, and is thus named top99neo. We also show how to include an acceleration technique recently investigated for TO by (Li et al., 2020), with a few extra lines of...
code and potentially carrying major speedups. Changes needed for the extension to 3D problems are remarkably small, making the corresponding code (top3d125) the most compact and efficient Matlab implementation for 3D compliance TO to date.

Our primary goal is not to present innovative new research. Rather, we aim at sharing some shortcuts and speedups that we have noticed through time, to the benefit of the research community. Improvements introduced by the present codes will be much useful also on more advanced problems, such as buckling optimization, which will be dealt with in forthcoming work.

The paper is organized as follows. In Section 2 we recall the setting of TO for minimum compliance. Section 3 is devoted to describe the overall structure of the 2D code, focusing on differences with respect to top88. Subsection 3.1 Subsection 3.3 give insights about the main speedups and show performance improvements with respect to top88. The very few changes needed for the 3D code are listed in Section 4, where an example is presented and the efficiency is compared to the previous code from Amir et al (2014). Some final remarks are given in Section 5 Appendix A gives some details about the redesigns step that are useful for better understanding a method proposed in Subsection 3.2 and the Matlab codes are listed in Appendix B and Appendix C.

2 Problem formulation and solution scheme

We consider a 2D/3D discretization $\Omega_h$ consisting of $m$ equi-sized quadrilateral elements $\Omega_e$. Hereafter we denote by $n$ the global number of Degrees of Freedom (DOFs) in the discretization and by $d$ the number of (local) DOFs of each element.

Let $x = \{x_e\}_{e=1}^m \in [0,1]^m$ be partitioned between $x_A$ and $x_P$, the sets of active (design) variables and passive elements, respectively. The latter may be further split in the sets of passive solid $P_1 (x_e = 1)$ and void $P_0 (x_e = 0)$ elements, of cardinalities $m_{P_1}$ and $m_{P_0}$, respectively (see Figure 1(a)).

The set of physical variables $\hat{x} = H(\tilde{x})$ are defined by the relaxed Heaviside projection (Wang et al 2011)

$$H(\tilde{x}, \eta, \beta) = \frac{\tanh(\beta \eta) + \tanh(\beta (\tilde{x} - \eta))}{\tanh(\beta \eta) + \tanh(\beta (1 - \eta))}$$

with threshold $\eta$ and sharpness factor $\beta$, where $\hat{x} = H\tilde{x}$ is the filtered field, obtained by the linear operator

$$H(x_e, r_{min}) := \frac{\sum_{i \in N_e} h_{e,i} x_i}{\sum_{i \in N_e} h_{e,i}}$$

where $N_e = \{i \mid \text{dist}(\Omega_i, \Omega_e) \leq r_{min}\}$ and $h_{e,i} = \max(0, r_{min} - \text{dist}(\Omega_i, \Omega_e))$.

Given a load vector $f \in \mathbb{R}^n$ and the volume fraction $\epsilon \in (0,1)$ we consider the optimization problem

$$\min_{x_A \in [0,1]^m_A} c(\tilde{x})$$

s.t. $V(\tilde{x}) \leq \epsilon |\Omega_h|$ 

for the minimization of compliance $c(\tilde{x}) = u^T f$ with an upper bound on overall volume

$$V(\tilde{x}) = \sum_{e=1}^m |\Omega_e| \tilde{x}_e = \frac{1}{m} \left( m_{P_1} + \sum_{e \in A} \tilde{x}_e \right) \leq \epsilon$$

Problem (3) is solved with a nested iterative loop. At each iteration, the displacement $u$ is computed by solving the equilibrium problem

$$Ku = f$$ (5)
where the stiffness matrix $K = K(\hat{x})$ depends on the physical variables through a SIMP interpolation (Bendsoe and Sigmund, 1999) of the Young modulus

$$E(\hat{x}) = E_{\min} + \frac{d^2}{E_0 - E_{\min}}$$

with $E_0$ and $E_{\min}$ the moduli of solid and void ($E_{\min} \ll E_0$). The gradients of compliance and structural volume with respect to $\hat{x}$ read ($\chi_e = 1$ if $e \in \mathcal{A}$ and 0 otherwise and $1_m$ is the identity vector of dimension $m$)

$$\nabla_{\chi_e} \mathcal{C}(\hat{x}) = -u^T \nabla_{\chi_e} K u_{\chi_e}, \quad \nabla_{\chi_e} V(\hat{x}) = \frac{1}{m} 1_m \chi_e$$

and the sensitivities with respect to the design variables are recovered as

$$\nabla_{x_k} \mathcal{C}(x) = \nabla_{x_k} \mathcal{H} \circ (H^T \nabla_{\chi_e} \mathcal{C}(\hat{x}))$$

$$\nabla_{x_k} V(x) = \nabla_{x_k} \mathcal{H} \circ (H^T \nabla_{\chi_e} V(\hat{x}))$$

where $\circ$ represents the elementwise multiplication and

$$\nabla_{x_k} \mathcal{H} = \beta \frac{1 - \tanh(\beta(x - \eta))}{\tanh(\beta \eta) + \tanh(\beta(1 - \eta))}$$

The active design variables $e \in \mathcal{A}$ are then updated by the Optimality Criterion rule (Sigmund, 2001)

$$x_{k+1,e} = U(x_{k,e}) = \begin{cases} 
\delta_- & \text{if } \mathcal{F}_{k,e} < \delta_- \\
\delta_+ & \text{if } \mathcal{F}_{k,e} > \delta_+ \\
\mathcal{F}_{k,e} & \text{otherwise}
\end{cases}$$

where $\delta_-=\max(0,x_{k,e}-\mu)$, $\delta_+=\min(1,x_{k,e}+\mu)$, for the fixed move limit $\mu \in (0,1)$ and

$$\mathcal{F}_{k,e} = x_{k,e} \left( -\frac{\partial_u c_k}{\lambda_k \partial_v V_k} \right)^{1/2}$$

depends on the element sensitivities.

In (11) $\lambda_k$ is the approximation to the current Lagrange multiplier $\lambda_k^*$ associated with the volume constraint. This is obtained by imposing $V(\mathcal{X}_{k+1}(\lambda)) = f|\Omega_h| \approx 0$, e.g. by bisection on an interval $A_k^{(0)} > \lambda_k^*$.

### 3 Matlab implementation and speedups

The Matlab routine for 2D problems (see Appendix B) is called with the following arguments

```
top99neo(nelx, nely, volfrac, penal, rmin, ft, eta, beta, move, maxit);
```

where `nelx, nely` define the physical dimensions and the mesh resolution, `volfrac` is the allowed volume fraction on the overall domain (i.e. $\mathcal{A} \cup \mathcal{P}$), `penal` the penalization used in (6) and `rmin` the filter radius for (2). The parameter `ft` is used to select the filtering scheme: density filtering alone if `ft`=1, whereas `ft`=2 or `ft`=3 also allows the projection (1), with `eta` and `beta` as parameters. `move` is the move limit used in the OC update and `maxit` sets the maximum number of re-design steps.

The routine is organized in a set of operations which are performed only once and the loop for the TO iterative re-design. The initializing operations are grouped as follows

PRE.1) MATERIAL AND CONTINUATION PARAMETERS
PRE.2) DISCRETIZATION FEATURES
PRE.3) LOADS, SUPPORTS AND PASSIVE DOMAINS
PRE.4) DEFINE IMPLICIT FUNCTIONS
PRE.5) PREPARE FILTER
PRE.6) ALLOCATE AND INITIALIZE OTHER PARAMETERS

and below we give details only about parameters and instructions not found in the `top88` code.

To apply continuation on the generic parameter “par”, a data structure is defined

```
parCont = {istart, maxPar, isteps, deltaPar};
```

such that the continuation starts when `loop=istart` and the parameter is increased by `deltaPar` each `isteps`, up to the value `maxPar`. This is implemented in Line 6 and 7 for the penalization parameter $p$ and the projection factor $\beta$, respectively. The update is then performed, by the instruction (see Line 92)

```
par=par+(loop>=parCnt{1}).*(par<parCnt{2}).*mod(
  loop,parCnt{3})==0).*parCnt{4}
```

making use of compact logical operations. Continuation can be switched off e.g. by setting `maxPar<=par`, or `istart>=maxit`.

The blocks defining the discretization (PRE.2) contain some changes compared to `top88`. The number of elements (nEl), DOFs (nDof) and the set of node numbers (nodeNrs) are defined explicitly, to ease and shorten some following instructions. The setup of indices (K, jK, used for the sparse assembly, is performed in Lines 15–21 and follows the concept detailed in Subsection 3.1). The coefficients of the lower diagonal part of the elemental stiffness matrix are defined in vectorized form, such that $K_e = V(K_e^{(s)})$ (see Lines 22–26). $K_e$ is used for the assembly strategy described in Subsection 3.1). However, in Lines 27–29 we also recover the complete elemental matrix ($Ke0$), used to perform the double product $u^T K_e u$, when computing the compliance sensitivity (7). Although this could also be written in terms of the matrix $K_e^{(s)}$ only, this option would increase the number of matrix/vector multiplications.

In PRE.3) the user can specify the set of restrained (fixed) and loaded (1dof) DOFs and passive regions ($P_l \leftrightarrow \text{pass}S$ and $P_0 \leftrightarrow \text{pass}V$) for the given configuration. Supports and loads are defined as in the `top88`
code, whereas passive domains may be specified target-
ing a set of column and rows from the array \texttt{e1Nrs}.
Independently of the particular example, Lines 34–36
define the vector of applied loads, the set of free DOFs,
and the sets of active, $\mathcal{A} \leftrightarrow \mathcal{A} nt$ design variables.

In order to make the code more compact and read-
able, operations which are repeatedly performed within
the TO optimization loop are defined through inline
functions in \texttt{PRE.4} (Lines 38–43). The filter operator is
built in \texttt{PRE.5} making use of the built–in Matlab func-
tion \texttt{imfilter}, which represents a much more efficient
alternative to the explicit construction of the neighboring
array. A similar approach was already outlined by
Andreassen et al (2011), pointing to the Matlab func-
tions \texttt{imfilter} and \texttt{imfilter}ing the density filter and, if selected, also the projection.

Some final initializations and allocations are per-
formed in \texttt{PRE.6}). The design variables are initialized
with the modified volume fraction, accounting for the
passive domains (Line 52–53) and the constant volume
sensitivity ($\bar{\lambda}$) is computed in Line 51.

Within the redesign loop, the following five blocks of
operations are repeatedly performed

\textbf{RL.1) COMPUTE PHYSICAL DENSITY FIELD}
\textbf{RL.2) SETUP AND SOLVE EQUILIBRIUM EQUATIONS}
\textbf{RL.3) COMPUTE SENSITIVITIES}
\textbf{RL.4) UPDATE DESIGN VARIABLES AND APPLY CONTINUATION}
\textbf{RL.5) PRINT CURRENT RESULTS AND PLOT DESIGN}

In block RL.1), the physical field is obtained, apply-
ing the density filter and, if selected, also the projection.
If $ft=3$, the special value of the threshold $\texttt{eta}$ giving a
volume–preserving projection is computed, as discussed in
Subsection 3.2.

The stiffness interpolation and its derivative ($\delta K,
\delta s K$) are defined and the stiffness matrix is assembled
(see Lines 73–76). Ideally, one could also get rid of Lines
73–74 and directly define $\delta K$ in Line 75 and $\delta s K$ within
Line 79. However, we decide to keep these operations
apart, enhancing the readability of the code and to ease
the specification of different interpolation schemes.

\textbf{Equation 5} is solved on Line 77 by using the Matlab
function \texttt{decomposition}, which can work with only half
of the stiffness matrix (see Subsection 3.1). The sensi-
tivity of compliance is computed and the backfiltering
operations are performed in RL.3).

The update (10), with the nested application of the
bicorrection process for finding $\lambda_k$, is implemented in RL.4)
(Lines 86–91) and we remark that $\texttt{lm}$ represents $\sqrt{s}$.

Some information about the process are printed and
the current design is plotted in RL.5} (Lines 94–97).
On small discretizations repeated plotting operations
absorb a significant fraction of the CPU time (e.g. 15%
for $m = 4800$). Therefore, one might just plot the final
design, moving Lines 96–97 outside the redesign loop.

Tests in the following have been run on a laptop
equipped with an Intel(R) Core(TM)i7-5500U@2.40GHz
CPU, 15GB of RAM and Matlab 201b running in se-
rrial mode under Ubuntu 18.04 (but a similar perfor-
mance is expected in Windows setups). We will often
refer to the half MBB beam example (see Figure 2) for
numerical testing. Unless stated otherwise we choose
\begin{equation}
\Omega = 300 \times 100, \ f = 0.5 \text{ and } r_{\text{min}} = 8.75 \text{ (Sigmund}
\end{equation}
\begin{equation}
\text{2007). The load, having total magnitude } |q| = 1 \text{ is applied}
\end{equation}
to the first node. No passive domains are intro-
duced for this example, therefore $\texttt{pasS=1}$; $\texttt{pasV=1}$;
and we set $E_1 = 1$, $E_0 = 10^{-9}$ and $\nu = 0.3$ in all the
tests.

3.1 Speedup of the assembly operation

In \texttt{top88} the assembly of the global stiffness matrix is
performed by the built–in Matlab function \texttt{sparse}

\begin{verbatim}
K = sparse( iK, jK, sK );
K = ( K +K' ) / 2;
\end{verbatim}

where $\delta K \in \mathbb{R}^{m \times d \times 1}$ collects the coefficients of all the
elemental matrices in a column–wise vectorized format
(i.e. $\nabla(K_r)$) and $\mathcal{I} K$ are the sets of indices mapping
each $\delta K(i)$ to the global location $(\mathcal{I} K(i), \mathcal{J} K(i))$.

These two sets are set up through the operations

\begin{equation}
\mathcal{I} K = \nabla \left([C \otimes \mathbf{1}_d]T\right), \quad \mathcal{J} K = \nabla \left([C \otimes \mathbf{1}_d^T]T\right)
\end{equation}

where $C_{[m \times d]}$ is the connectivity matrix and $\otimes$ is
the Kronecker product (Horn and Johnson 2012). The size of
the array $\mathcal{I} \times [\mathcal{I} K, \mathcal{J} K] \in \mathbb{N}^{m \times d \times 2}$ grows very quickly
with the number of elements $m$, especially for 3D dis-
cretizations (see Table 1), and even though its elements are
integers, the \texttt{sparse} function requires them to be
Table 1: Number of entries in the array $\mathcal{I}$ and corresponding memory requirement for the 2D and 3D test discretizations. White background refers to the $F$ strategy with coefficients specified as double, cyan background to the $H$ strategy and light orange to the $\text{int32}$. The $H$ strategy cuts $|\mathcal{I}|$ and memory of $\approx 44\%$ in 2D and $\approx 48\%$ in 3D. Then, specifying the indexes as $\text{int32}$ further cuts memory of another $50\%$.

| $m$ | $120^2$ | $240^2$ | $480^2$ | $960^2$ | $1920^2$ |
|-----|----------|----------|----------|----------|----------|
| 2D $|\mathcal{I}|$ | 1,843,200 | 7,372,800 | 29,391,200 | 117,964,800 | 120/ | 858,200 | 4,147,200 | 16,588,800 | 66,355,200 | 265,420,800 |
| memory (MB) | 14.7 | 59 | 235 | 943 | - | 4.1 | 16.6 | 66.3 | 265.4 | 1061 |
| 3D $|\mathcal{I}|$ | 589,824 | 4,718,592 | 37,748,736 | 301,898,888 | 2,415,919,104 |
| memory (MB) | 8  | 32 | 64 | 128 | 32 | 420 | 1920 | 16 | 8 | 32 | 64 | 128 |

specified as $\text{double}$ precision numbers. The corresponding memory burden slows down the assembly process and restricts the size of problems workable on a laptop.

The efficiency of the assembly can be substantially improved by

1. Acknowledging the symmetry of both $K_e$ and $K$;
2. Using an assembly routine working with $iK$ and $jK$ specified as integers;
3. Factorization of the lower triangular matrix $K^{(s)}$ and we remark that the full operator can be recovered by $K = K^{(s)} + (K^{(s)})^T - \text{diag}[K^{(s)}]$ (15) which costs as much as the averaging operation $\frac{1}{2}(K + K^T)$, performed in top88 to get rid of roundoff errors. However, the Matlab built-in Cholesky solver and the corresponding decomposition routine can use just $K^{(s)}$, if called with the option ‘lower’.

Point 2 gives the most dramatic improvement, and can be accomplished by using routines developed by independent researchers. The $\text{sparsel2}$ function, from Suite Sparse (Davis 2019), was already pointed out by Andreassen et al (2011) as a better alternative to the built-in Matlab $\text{sparsel2}$; however, no quantitative comparisons were performed. According to the CHOLMOD reference manual (Davis 2009), $\text{sparsel2}$ works exactly as $\text{sparsel2}$, but allowing the indices $iK$ and $jK$ to be specified as integers (accomplished by defining this type for the connectivity matrix, see Lines 11 and 13).
elements, respectively. Figure 3 shows time scalings for the different strategies: “F” corresponds to the assembly in top88, “H” takes advantage of the matrix symmetry only and “H,fsparse” corresponds to the use of the fsparse routine (Engblom and Lukarski 2016) also. All the approaches exhibit a linear scaling of CPU time w.r.t the DOFs number. However, half the CPU time can be cut just by assembling $K^{(s)}$ (strategy H,fsparse). Therefore, we definitely recommend this to users who aim to solve medium–size ($10^3$ to $10^6$ DOFs) structural TO problems on a laptop. However, the most substantial savings follow from using fsparse (Engblom and Lukarski 2016) and by coupling these two strategies (H,fsparse) speedups of 10 for the 2D and 15 for 3D setting can be achieved. It is worth to highlight that a 3D stiffness matrix of the size of $\approx 9 \cdot 10^5$ can be assembled in less than a second and even one of size $6.2 \cdot 10^6$ can be assembled on a laptop in less than 10s. For this last case the sole storage of the arrays $iK, jK$ and $sK$ would cause a memory overflow, ruling out the “F” approach.

3.2 Speedup of the OC update

The cost of the re–design step $x_{k+1} = U(x_k)$ is proportional to the number of bisections ($n_{bs}$) required for computing the approximation $\lambda_k \approx \lambda_k$. The following estimate (Quarteroni et al, 2000)

$$n_{bs} \geq \frac{\log(|A^{(0)}|) - \log(\tau)}{\log(2)} - 1$$

is a lower bound to this number for a given accuracy $\tau > |\lambda_k - \tilde{\lambda}_k|$ and it is clear that $n_{bs}$ would decrease if $A^{(0)}$, the initial guess for the interval bracketing $\lambda_k$, could be shrunk. Moreover, the volume constraint should be imposed on the physical field ($\mathbf{x}$ or $\hat{\mathbf{x}}$) and, in the original top88 implementation, this requires a filter application at each bisection step, which may become expensive.

The efficiency of the re–design step can be improved by a two step strategy

1. Using volume–preserving filtering schemes;
2. Estimating the interval $A^{(0)}$ bracketing the current Lagrange multiplier $\lambda_k$.

Concerning point 1, the density filter is naturally volume–preserving (i.e. $V(x_k) = V(\tilde{x}_k)$) (Bourdin 2001, Bruns and Tortorelli 2001). Therefore, the volume constraint can be enforced on $V(x_k)$ as long as the density filter alone is considered ($ft=1$). The relaxed Heaviside projection (1), on the other hand, is not volume–preserving for any $\eta$; thus it would require one filter– and projection application at each bisection step. How-

---

1. https://github.com/stefanengblom/stenglib

---

Here we suggest the “fsparse” routine, developed by Engblom and Lukarski (2016). Besides working with integers $4K$ and $3K$, the function enhances the efficiency of the sparse assembly by a better sorting of the operations. From our experience on a single core process, fsparse gives a speedup of 170–250% compared to sparse2, and is also highly parallelizable (Engblom and Lukarski 2016). Defining the sets $iKr$ and $jKr$ as int32 type, we can drastically cut the memory requirements, still representing $n \approx 2.1 \cdot 10^9$ numbers, far beyond the size of problems one can tackle in Matlab.

In order to use fsparse, one needs to download the “stenglib” library [1] and follow the installation instructions in the README.md file. The packages of the library can be installed by running the “makeall.m” file. As fsparse is contained within the folder “Fast”, one may only select this folder when running makeall.m.

We test the efficiency of the assembly approaches on 2D and 3D uniform discretizations with $m^2$ and $m^3$

Fig. 3: Scaling of assembly time performed with the 3 strategies discussed in Subsection 3.1. Compared to the standard (F) assembly, the H strategy alone cuts near 50% of time and memory, and with the use of fsparse gives an overall efficiency improvement of 10–15 times.

---

[1] https://github.com/stefanengblom/stenglib

---
ever, \( \eta \) can also be made volume-preserving by computing, for each \( \tilde{x} \), the threshold \( \eta^*_k \) such that [Xu et al. 2010; Li and Khandelwal, 2015]

\[
\eta^*_k \rightarrow \min_{\eta \in [0,1]} |V(\tilde{x}_k(\eta)) - V(\tilde{x}_k)|
\]  

(17)

This can be done, e.g. by the Newton method, starting from the last computed \( \eta^*_{k-1} \) and provided the derivative of (1) with respect to \( \eta \)

\[
\frac{\partial V(\tilde{x}(\eta))}{\partial \eta} = -2\beta \sum_{i \in A} \left( e^{\beta(1-i)} - e^{\beta(1-i)} \right) \left( (\beta^2 - \epsilon \beta)x + \beta(x-\beta) \right)^2
\]  

(18)

Existence of \( \eta^* \in [0,1] \) for all \( \tilde{x} \in [0,1]^m \) follows from the fact that \( g(\eta) = V(\tilde{x}(\eta)) - V(\tilde{x}) \) is continuous on \([0,1]\) and \( g(\eta)g(1) \leq 0 \); uniqueness follows from the fact that \( \frac{\partial g}{\partial \eta} < 0 \) for all \( \eta \in (0,1) \).

Numerical tests on the MBB beam show that generally \( \eta^*_k \in [0.4,0.52] \), the larger variability occurring for low volume fractions (see Figure 4 (a)). We also observe that \( \eta^*_k \) takes values slightly above 0.5 when \( r_{\min} \) is increased or \( \beta \) is raised. Convergence to \( \eta^*_k \) is generally attained in 1–2 Newton iterations (see Figure 4 (a)).

The procedure for computing \( \eta^*_k \) from (17), with tolerance \( \epsilon = 10^{-6} \) and initial guess \( \eta_0 = \text{eta} \), provided by the user, is implemented in Lines 63–67, that are executed if the routine \text{top99neo} is called with the parameter \text{ft}=3. Otherwise, if \text{ft}=2, the input threshold \text{eta} is kept fixed. In case of the latter, the volume constraint should be consistently applied on \( V(\tilde{x}) \), otherwise some violation or over-shooting of the constraint will happen. In particular, if the volume constraint is imposed on \( x \) and \( \eta \) is kept fixed, one has \( V(\tilde{x}) > f|\Omega_h| \), if \( \eta < 0.5 \), and \( V(\tilde{x}) < f|\Omega_h| \), if \( \eta > 0.5 \).

Even tough we usually observed small differences, these may result in local optima or bad designs, especially for low volume fractions or high \( \beta \) values. Therefore, accounting for this more general situation Lines 87–91 should be replaced by the following:

```matlab
while (1/(2.*x(1)+1)+(1+1)>)le-4
   lmld=0.5*(1+1);
   x=x(max(min(min(min(x',lmld,xl),1),xl),0));
   if ft > 0
      xf=imfilter(reshape(x,nely,nexl),./Hs,b);
      [xf,pasb],z=deal(1,0);
      if ft=1, xf=pasr(x';eta,beta); end
   end
   if mean(xf(3))>volfrac, l(1)=lmld; else l(2)=lmld; end
end
```

However, there could be other situations when one cannot rely on volume preserving filters (e.g. when imposing length scale through robust design). Therefore, a more general strategy to reduce the cost of the OC update is to cut the number of bisection steps.

Fig. 4: Evolution of the parameter \( \eta^* \) realizing the equivalence \( V(\tilde{x}) = V(\tilde{x}) \), for different volume fractions \( f \) and filter radii \( r_{\min} (a) \) and evolution of the Lagrange multiplier estimate \( \lambda^* \) given by (19) compared to \( \lambda^* \) (b). For both plots, the cumulative number of Newton iterations \( n_{\text{Newton}} \) (viz. number of bisection steps) is shown along the right axis.

To this end, the selection of the initial bracketing interval \( A^{(0)} \) may build upon the upper bound estimate for \( \lambda^*_k \) (Hestenes 1969; Arora et al. 1991))

\[
\lambda_k^* = \left[ \frac{1}{mf} \sum_{e=1}^{m} (\frac{\partial c_e}{\partial V_k})^{1/2} \right]^2
\]  

(19)

More details on the derivation of (19) are given in Appendix A. The behavior of the estimate (19) is shown in Figure 4 (b) for the MBB example. The overall number of bisections \( n_{\text{bs}} \) in order to compute \( \lambda^*_k \) meeting the tolerance \( \tau = 10^{-8} \) when considering \( A^{(0)} = [0,\lambda]^* \) is cut by about 50%, compared with the one required by starting from \( A^{(0)} = [0,\lambda]^* \) as in \text{top88}. Moreover, if no projection is applied, (19) could be used together with (10) to perform an explicit Primal-Dual iteration.
to compute \((x_{k+1}, \lambda_k^*)\) and this would reduce the number of steps even more (see green curve in Figure 4b)).

However, in the basic versions of the codes, given in Appendix B and Appendix C, we consider the bisection process and (19) is used to bracket the search interval, as this procedure is more general.

### 3.3 Acceleration of the OC iteration

The update rule (10) resembles a Fixed-Point (FP) iteration \(x_{k+1} = U(x_k)\), generating a sequence \(\{x_k\}\) converging to a point such that \(r = U(x^*) - x^* = 0\).

Several methods are available to speedup the convergence of such a sequence (Brezinski and Chehab 1996; Ramire and Helfer 2015), somehow belonging to the family of quasi–Newton methods (Eyert, 1996). The acceleration proposed by Anderson (1965), for instance, the family of quasi–Newton methods (Eyert, 1996; Ramiere and Helfer, 2015), somehow belonging to the family of quasi–Newton methods (Eyert, 1996). The update rule (20) resembles a Fixed–Point (FP) iteration, \(x_{k+1} = \hat{x}_k + \zeta r_k^\#\), where \(\zeta \in [0, 1]\) is a damping coefficient and

\[
x_k^\# = x_k - \sum_{i=k-m_r}^{k-1} \gamma_i^{(k)} \Delta x_i = x_k - X_k \gamma_k
\]

(21)

\[
r_k^\# = r_k - \sum_{i=k-m_r}^{k-1} \gamma_i^{(k)} \Delta r_i = r_k - R_k \gamma_k
\]

The coefficients \(\gamma_i^{(k)}\) minimize the following

\[
\min_{\gamma} \|r_k^\#(\gamma)\|_2^2
\]

(22)

The rationale behind the method is to compute a rank–1 update of the inverse Jacobian matrix \(J_k^{-1}\) of the nonlinear system \(r_k = 0\). This has been shown to be equivalent to a multi–secant Broyden method (Eyert, 1996; Fang and Saad, 2009) starting from \(J_0^{-1} = -\zeta I\).

The update rule (20) is usually applied only once each \(q\) steps. Thus we can write more generally \(x_{k+1} = x_k + z_k\), where (Pratapa et al, 2016)

\[
z_k = \begin{cases} \alpha r_k & \text{if } k + 1 \notin \mathbb{N} \\ \zeta I - (X_k + \zeta F_k) \gamma_k & \text{if } k + 1 \in \mathbb{N} \end{cases}
\]

(23)

\((\alpha \in (0, 1))\) obtaining the so-called Periodic Anderson Extrapolation (PAE) (Pratapa et al, 2016; Li et al, 2020).

The implementation can be obtained, e.g. by adding the following few lines after the OC step (Line 91)

\[
\begin{align*}
& \text{fres} = z(\text{act}) - x(\text{act}); \\
& \text{if} \quad \text{loop} >= q0 \\
& \quad \text{sel} = \text{mod}(\text{loop} - q0, q) == 0; \\
& \quad \text{mix} = \text{sel} * \text{xi} + \text{alpha} * (1 - \text{sel}); \\
& \quad x(\text{act}) = \text{pae}(x(\text{act}), \text{fres}, \text{nr}, \text{loop} - q0, \text{mix}, q); \\
& \quad x(x > 1) = 1; \quad x(x < 0) = 0; \\
& \text{end}
\end{align*}
\]

where the part solving (22) and the update has been put in a separate routine for better efficiency.

In the above we use the “\(\alpha\)” for solving the least squares problem (22); however, strategies based on a QR (or SVD) decomposition may be preferred in terms of numerical stability. We refer to Fang and Saad (2009) for a deeper discussion on this point.

In order to assess the effect of different filtering schemes and the introduction of parameter continuation, Anderson acceleration is tested on the MBB example considering the following options

- **T1** Density filter alone, \(p = 3\);
- **T2** Density–and–projection filter, with \(\eta^*\) computed from (17) and \(\beta = 2\);
- **T3** As T2, but with continuation on both \(\beta\) and \(p\), defined by the parameters \(\text{betaCnt} = \{250, 16, 25, 2\}\) and \(\text{penalCnt} = \{50, 3, 25, 0, 25\}\);
- **T4** As T2, but for the discretization \(\Omega_h = 600 \times 200\)

For all the cases, the TO loop stops when \(\|r_k\|_2 / \sqrt{m} < 10^{-6}\), where the residual is defined with respect to the physical variables (i.e. \(r_k = x_k - x_{k-1}\) for T1 and \(r_k = x_k - x_{k-1}\) for T2–T4). The acceleration is applied each \(q\) steps, considering the last \(m_r = 4\) residuals, starting from iteration \(q_0 = 20\) for T1–T2 and from \(q_0 = 500\) for T3–T4, when both continuations have finished. We set \(\alpha = 0.9\) for the non–accelerated steps. The choice \(m_r = 4\) is based on the observation that convergence improvements increase very slowly for \(m_r > 3\) (Anderson, 1965; Eyert, 1996). However, a deeper discussion about the influence of all parameters on the convergence is outside the scope of the present work and we refer to Li et al (2020) or, in a more general context, to Walker and Ni (2011) for this.

Results are collected in Table 2 and Figure 6 showing the evolution of the norm of the residual, the flatness of the normalized compliance \(\Delta c_0 / c_0 = (c_k - c_{k-1}) / c_0\) and the non–discreteness measure \(m_{\text{ND}} = 100\)
Table 2: Comparison of convergence-related parameters for the standard (T) and accelerated (T-PAE) TO tests, for the MBB example

|     | it. | c     | \(\Delta c\) | \[1/\sqrt{n}\] | \(m_{ND}\) |
|-----|-----|-------|-------------|-------------|-----------|
| T1  | 2500| 252.7 | \(4.2 \times 10^{-8}\) | \(1.03 \times 10^{-5}\) | 0.025     |
| T1-PAE | 828 | 258.9 | \(4.2 \times 10^{-10}\) | \(9.95 \times 10^{-7}\) | 0.021     |
| T2  | 2500| 246.1 | \(5.1 \times 10^{-8}\) | \(3.21 \times 10^{-5}\) | 0.023     |
| T2-PAE | 352 | 253.9 | \(6.2 \times 10^{-9}\) | \(9.97 \times 10^{-7}\) | 0.014     |
| T3  | 2500| 199.6 | \(1.1 \times 10^{-4}\) | \(1.91 \times 10^{-3}\) | 0.014     |
| T3-PAE | 752 | 197.5 | \(3.7 \times 10^{-8}\) | \(8.72 \times 10^{-7}\) | 0.007     |
| T4  | 2500| 191.8 | \(2.0 \times 10^{-4}\) | \(3.21 \times 10^{-5}\) | 0.006     |
| T4-PAE | 818 | 192.1 | \(2.5 \times 10^{-7}\) | \(9.97 \times 10^{-7}\) | 0.001     |

Fig. 5: Optimized designs obtained without (left column) and with Anderson acceleration (right column) of the TO loop

\[4x^T(1-x)/m.\] We observe how Anderson acceleration substantially reduces the number of iterations needed to fulfill the stopping criterion, at the price of just a moderate increase in compliance (0.2–3%). Moreover, starting the acceleration just a few iterations later (e.g. it = 50 or it = 100 for T1), gives much lower compliance values \((c = 254.3\) and \(c = 252.9\), respectively) and for T3, T4 when the acceleration is started as the design has stabilized, compliance differences are negligible.

From Figure 5 it is easy to notice the trend of PAE of producing a design with some more bars. This may even give slightly stiffer structures, such as for case T3, where the non accelerated approach removes some bars after it = 2000, whereas stopping at the design of T3–PAE gives a stiffer structure.

A comment is about the convergence criterion used, which is different from the one in top88 (maximum absolute change of the design variables \(\|x_{t+1} - x_t\|_\infty\)). Here we consider it more appropriate to check the residual with respect to the physical design field, and the 2-norm seems to give a more global measure, less affected by local oscillations.

3.4 Performance comparison to top88

We compare the performance of top99neo to the previous top88 code. In the following we will refer to "top88" as the original code provided by [Andreassen et al. 2011] and to "top88U" as its updated version making use of the sparse2 function [Davis, 2009] for the assembly, with \(1^K\) and \(j^K\) specified as integers, and the filter implemented by using conv2.

The codes are tested by running 100 iterations for the MBB beam example (see Figure 2), for the discretizations 300 \(\times\) 100, 600 \(\times\) 200 and 1200 \(\times\) 400, a volume fraction \(f = 0.5\) and considering mesh independent filters of radii \(r_{\min} = 4, 8\) and 16, respectively. For top88 and top88U we only consider density filtering, whereas for the new top99neo we also consider the Heaviside projection, with the \(\eta^u\) computed as described in Subsection 3.2. It will be apparent that the cost of this last operation is negligible.

Timings are collected in Table 3 where \(t_H\) is the average cost per iteration, \(t_A\) and \(t_S\) are the overall time spent by the assembly and solver, respectively, and \(t_U\) is the overall time spent for updating the design variables. For top88 and top88U the latter consists of the OC updating and the filtering operations performed when applying the bisection on the volume constraint. For top99neo this term accounts for the cost of the OC updating, that for estimating the Lagrange multiplier \(\nu^e\) as discussed in Subsection 3.2 and the filter and projection (Lines 59–70). \(t_P\) collects all the preliminary operations, such as the set up of the discretization, filter etc, repeated only once, before the TO loop starts.

From \(t_H\) we clearly see that top99neo enhances the performance of the original top88 by 2.66, 3.85 and 5.5 times on the three discretizations, respectively. Furthermore, timings of top88 on the largest discretization (1200 \(\times\) 400), relate to a smaller filter size \((r_{\min} = 12)\), because of memory issues; thus, the speedup is even underestimated in this case. Comparing to top88U version, the improvements are less pronounced (i.e. 1.55, 1.57 and 1.78 times) but still substantial. The computational cost of the new assembly strategy is very low, even comparing to the top88U version, and its weight on the overall computational cost is basically constant. Also, from Table 3 it is clear that the design variables update weighs a lot on the overall CPU time, for both top88 and top88U. On the contrary, this becomes very
Fig. 6: Evolution of some parameters related to convergence for the standard and Anderson accelerated TO process. The first row shows the normalized norm of the residual defined on physical variables, the second row shows a measure of the flatness of the objective function and the last row shows the non-discretness measure.

cheap in the new top99neo thanks to the strategies discussed in Subsection 3.2.  In the top99neo, $t_U$ takes about 4–5% of the overall CPU time.

Computational savings would become even higher when adopting the larger filter size $r_{\text{min}} = 8.75$ for the mesh $300 \times 100$, and scaling to $r_{\text{min}} = 17.5$ and $r_{\text{min}} = 35$ on the two finer discretizations. For these cases, speedups with respect to top88 amount to 4.45 and 10.35 on the first two meshes, whereas for the larger one, the setup of the filter in top88 causes a memory overflow. Speedups with respect to top88 amount to 1.55, 2.55 and 3.6 times respectively.

3.5 Frame reinforcement problem

Let us go back to the example of Figure 1(a), adding the specification of passive domains and a different loading condition.

We may think of a practical application like a reinforcement problem for the solid frame, with thickness $t = L/50$ ($P_1$), subjected to two simultaneous loads. A vertical, uniformly distributed load with density $q = -2$ and a horizontal height–proportional load, with density $b = \pm y/L$. Some structural material has to be optimally placed within the active design domain $\mathcal{A}$, in order to minimize the compliance, while keeping the void space ($\mathcal{P}_0$), which may represent a service opening.

To describe this configuration we only need to replace Lines 31–33 with the following

```matlab
[lDofv, lDofh] = deal (2* nodeNrs (1,:) ,2* nodeNrs (: , end )-1); fixed = [1 ,2 , nDof ]; a1= elNrs (1: nely /50 ,:) ; a2= elNrs ( : ,[1: nelx /50 , end - nelx /50+1: end ]); a3= elNrs (2* nely /5: end ,2* nelx /5: end - nelx /5) ; [pasS , pasV ]= deal ( unique ([a1 (:) ;a2 (:) ]),a3 (:) );
```

where lDofv and lDofh target the DOFs subjected to vertical and horizontal forces, respectively. Then the load vector is assembled as

```matlab
F = fsparse (lDofv ’,1 ,[-2 , [nDof ,1 ]]) + fsparse (... lDofh ’,1 ,[-0:1/(nely+1):1 ]’,[nDof ,1]);
```

Figure 7 shows the two optimized design corresponding to the two orientations of the horizontal load $b$, after 100 re-design steps. The routine top99neo has been called with the following arguments: nely=nelx=900, volfrac=0.2, penal=3, rmin=8, ft=3, eta=0.5, beta=2.
and no continuation is applied. The cost per iteration is about 10.8s and, considering the fairly large discretization of $1.62 \cdot 10^6$ DOFs, is very reasonable.

### 4 Extension to 3D

The implementation described in Section 3 is remarkably easy to extended to 3D problems (see Appendix C).

Notable modifications are the definition of $K_e$ for the 8–node hexahedron (Lines 24–47) and the solution of the equilibrium equations (6), now performed by

$$L = \text{chol}(K(\text{free, free}), \text{'lower'})$$

$$U(\text{free}) = L' \setminus (L \setminus F(\text{free}))$$

which in this context has been observed to be faster than the decomposition routine. Then, apart from the plotting instructions, all the operations are the same as in the 2D code and only 12 lines need minor modifications, basically to account for the extra space dimension (see tags “#3D#” in Appendix C).

We test the 3D implementation on the cantilever example shown in Figure 8 (a), for the same data considered in Amir et al (2014). The discretization is set to $\Omega_h = 48 \times 24 \times 24$, the volume fraction is $f = 0.12$ and the filter radius $r_{\text{min}} = \sqrt{3}$. We also consider the volume–preserving Heaviside projection, (ft=3). Figure 8 (b,c) show the designs obtained after 100 re-design steps, for the two different filter boundary conditions. The design in (b), identical to the one in Amir et al (2014), corresponds to zero–Neumann boundary conditions (i.e. the option “symmetric” was used in imfilter). The design in (c) on the other hand, corresponds to zero–Dirichlet boundary conditions for the filter operator and is clearly a worse local minimum.

The overall CPU time spent over 100 iterations is 1741s and about 96% of this is due to the solution of the state equation. Only 1.2% of the CPU time is taken by matrix assemblies and 0.4% by filtering and the design update processes.

Upon replacing the direct solver in top3D125 with the same multigrid preconditioned CG solver of Amir et al (2014) we can compare the efficiency of the two codes. We refer to Table 4 for the CPU timings, considering the discretizations $\Omega_h = 48 \times 24 \times 24$ ($l = 3$ multigrid levels) and $\Omega_h = 96 \times 48 \times 48$ ($l = 4$ multigrid levels). top3D125 shows speedups of about 1.8 and 1.9, respectively and most of the time is cut on the matrix assembly. In the code of Amir et al (2014) this operation takes about 50% of the overall time (and notably has the same weight as the state equation solve) whereas in top3D125 this weight is cut to 7 – 10%. Also the time spent for the OC update is reduced, even though the code of Amir et al (2014) already implemented a strategy for avoiding filtering at each bisection step.

### 5 Concluding remarks

We have presented new Matlab implementations of compliance Topology Optimization for 2D and 3D domains. Compared to the previous top88 code (Andreasen et al 2011) and available 3D codes (e.g. by Liu and Tovar (2014) or Amir et al (2014)), the new codes show remarkable speedups.

Improvements are mainly due to the following

---

**Table 3:** Comparison of numerical performance between the old top88/top88U and new top99neo Matlab code. $t_\text{it}$ is the cost per iteration, $t_A$, $t_S$, $t_U$ are the overall times for assembly, equilibrium equation solve and design update, respectively. $t_P$ is the time spent for all the preliminary operations. Values within brackets represent the % weight of the corresponding operation on the overall CPU. On the larger mesh, top88 is run with $r_{\text{min}} = 12$, because of memory issues.

| $\Omega_h$ | 300 × 100, $r_{\text{min}} = 4$ | 600 × 200, $r_{\text{min}} = 8$ | 1200 × 400, $r_{\text{min}} = 16$ |
|------------|--------------------------------|--------------------------------|--------------------------------|
| $t_\text{it}$ | 0.615 0.358 0.231 | 4.57 1.87 1.19 | 31.3 10.1 5.69 |
| $t_A$ | 19.4(31.5) 5.4(15.0) 1.4 (6.1) | 83.1(18.2) 31.3(16.7) 5.6 (4.7) | 361.1(11.6) 151.5(15.2) 30.7 (5.4) |
| $t_S$ | 23.1(37.4) 22.9(59.3) 19.7(85.3) | 122.4(26.8) 109.3(58.4) 106.9(89.7) | 592.5(19.0) 513.2(50.9) 510.5(89.6) |
| $t_U$ | 13.3(21.6) 4.8(13.5) 1.2 (4.8) | 223.8(48.8) 38.0(20.3) 5.2 (4.4) | 1164.2(37.4) 310.4(31.4) 29.2 (5.1) |
| $t_P$ | 0.8(1.3) 0.06 (0.2) 0.1 (0.3) | 12.9 (2.8) 0.1(< 0.1) 0.2(< 0.1) | 92.3 (3.1) 0.5(< 0.1) 0.6(< 0.1) |
Fig. 8: Geometrical sketch of the 3D cantilever example (a) and optimized topology for $\Omega_h = 48 \times 24 \times 24$ and considering the two filter boundary conditions (b,c). The design in (d) corresponds to the finer mesh $\Omega_h = 96 \times 48 \times 48$ and has been obtained by replacing the direct solver with the multigrid–preconditioned CG (see Amir et al (2014) for details).

Table 4: Performance comparison between the new top3D125 code and the one from Amir et al (2014). $t_{it}$, $t_A$, $t_S$, $t_U$ and $t_P$ have the same meaning as in Table 3 and numbers between brackets denote the % weight of the operations on the overall CPU time.

| $\Omega_h$ | $48 \times 24 \times 24$, $r_{\min} = \sqrt{3}$ | $96 \times 48 \times 48$, $r_{\min} = 2\sqrt{3}$ |
|------------|-----------------------------------------------|-----------------------------------------------|
| top3dmcg  | t_{it} 3.19 (1.79) | t_{it} 3.19 (1.79) |
| top3d125  | t_{it} 3.19 (1.79) | t_{it} 3.19 (1.79) |
| top3dmcg  | t_A 160.6 (50.3) | t_A 136.9 (50.1) |
| top3d125  | t_A 160.6 (50.3) | t_A 136.9 (50.1) |
| top3dmcg  | t_S 148.1 (46.4) | t_S 125.0 (45.7) |
| top3d125  | t_S 148.1 (46.4) | t_S 125.0 (45.7) |
| top3dmcg  | t_U 1.97 (0.6) | t_U 21.2 (0.8) |
| top3d125  | t_U 1.97 (0.6) | t_U 21.2 (0.8) |
| top3dmcg  | t_P 0.74 (0.4) | t_P 0.74 (0.4) |
| top3d125  | t_P 0.74 (0.4) | t_P 0.74 (0.4) |

1. The matrix assembly is made much more efficient by defining mesh related quantities as integers (Matlab int32) and assembling just one half of the matrix;
2. The number of OC iterations is drastically cut by looking at the explicit expression of the Lagrange multiplier for the problem at hand;
3. Filter implementation and volume-preserving density projection allow to speed up the redesign step.

The new codes are computationally well balanced and as the problem size increases the majority of the time (85 to 90% for 2D and even 96% for 3D discretizations) is spent on the solution of the equilibrium system. This is precisely what we aimed at, as this step can be dealt with efficiently by preconditioned iterative solvers (Amir et al 2014, Ferrari et al 2018, Ferrari and Sigmund 2020). We also discussed Anderson acceleration, that has recently been applied to TO also by Li et al (2020), to accelerate the convergence of the overall optimization loop.

We point out that even if we specifically addressed volume constrained compliance minimization and density-based TO the methods above can be applied also to level-set and other TO approaches. Point 1 can be extended to all problems governed by symmetric matrices. Point 2 and 3 can also be extended to other problems, to some extent and Anderson acceleration is also usable in a more general setting (e.g. within MMA).

Therefore, we believe that this contribution should be helpful to all researchers and practitioners who aim at tackling TO problems on laptops, and set a solid framework for the efficient implementation of more advanced procedures.

Reproducibility of results Matlab codes are listed in the Appendix and available at www.topopt.dtu.dk. The stenglib package, containing the fsparse function, is available for download at https://github.com/stefanengblom/stenglib.

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

Acknowledgements The project is supported by the Villum Fonden through the Villum Investigator Project “InnoTop”. The authors are grateful to members of the TopOpt group for their useful testing of the code.

References

Amir O, Sigmund O (2011) On reducing computational effort in topology optimization: How far can we go? Structural and Multidisciplinary Optimization 44(1):25–29, DOI 10.1007/s00158-010-0586-7
Amir O, Aage N, Lazarov BS (2014) On multigrid–CG for efficient topology optimization. Structural and Multidisciplinary Optimization 49(5):815–829, DOI 10.1007/s00158-013-1015-5
Anderson DG (1965) Iterative procedures for nonlinear integral equations. Journal of the Association for Computing Machinery 12(4):547–560
Andreassen E, Andreasen CS (2014) How to determine composite material properties using numerical homogenization. Computational Material Science 83:488–495
Andreassen E, Clausen A, Shvechene M, Lazarov BS, Sigmund O (2011) Efficient topology optimization in matlab using 88 lines of code. Structural and Multidisciplinary Optimization 43(1):1–16, DOI 10.1007/s00158-010-0594-7
A new 99 line Matlab code for compliance Topology Optimization

A new 99 line Matlab code for compliance Topology Optimization

\[ \sum_{\delta} e \delta \min_{\partial=1} V_c(x_k) \approx x_{e+1} f | \Omega | \leq 0 \]

\[ \frac{\text{min}}{\mathbf{x} \in [\mathbf{a}, \mathbf{b}]} e(\mathbf{x}) \simeq c_0 + \sum_{m=1}^m (-s_{h,k} \partial_e c(\mathbf{x}_k))x_{e}^{-1} \]

\[ \text{s.t.} \sum_{e=1}^m \partial_e V(x_k)x_e - f[j] \leq 0 \]

\[ \mathbf{0} \]

\[ \mathbf{0} \]

\[ \mathbf{0} \]

\[ \mathbf{0} \]

\[ \mathbf{0} \]

\[ \mathbf{0} \]

\[ \mathbf{0} \]
We set up the Lagrangian associated with \( \sum_{e=1}^{m} \frac{\partial c}{\partial x_e} (x_e - f|\Omega_h|) \) and seek the pair \((x_{k+1}, \lambda^*_k)\) in \(\mathbb{R}^m \times \mathbb{R}_+\) solving the subproblem

$$
\max_{\lambda > 0} \left\{ \psi(\lambda) := \min_{x \in \mathcal{C}} L(x, \lambda) \right\}
$$

where \(\mathcal{C} = \{ x \in \mathbb{R}^m \mid \delta_- \leq x_e \leq \delta_+ , e = 1, \ldots, m \} \) and \(\psi(\lambda)\) is the dual function. \(\psi(\lambda)\) is solved by Primal-Dual (PD) iterations, as \(x\) and \(\lambda\) are interlaced. Replacing \(\xi = x_k\) and using subscripts \((j)\) to denote inner PD iterations we have

1. Fixed \(\lambda = \lambda(j)\), the inner minimization in \(25\) gives

$$
\xi^2 \partial_c c(\xi)x_e^2 + \lambda \partial c V(\xi) = 0 \implies x_e = \xi \left( -\frac{\partial_c c}{\lambda \partial c V(\xi)} \right)^{\frac{1}{2}}
$$

due to separability of the approximation. Let us denote the rightmost expression \(x_e = \mathcal{F}(j)(\lambda_e)\), and taking into account the box constraints in \(\mathcal{C}\) we have

$$
\mathcal{U}(x_e) = \begin{cases} 
  x_{(j+1),e} = \delta_+ & \text{if } e \in \mathcal{L} = \{ e \mid x_{(j+1),e} \leq \delta_- \} \\
  x_{(j+1),e} = \delta_- & \text{if } e \in \mathcal{U} = \{ e \mid x_{(j+1),e} \geq \delta_+ \} \\
  x_{(j+1),e} = \mathcal{F}(j),e & \text{if } e \in \mathcal{M} = \{ e \mid \delta_- < x_{(j+1),e} < \delta_+ \} 
\end{cases}
$$

where \(\mathcal{C} = \mathcal{L} + \mathcal{U} + \mathcal{M}\). The above is equivalent to \(10\).

2. We then evaluate the dual function for \(x_{(j+1)}\) given by \(26\), and the stationarity \((\partial_\lambda \psi = 0)\) gives

$$
\sum_{e=1}^{m} \frac{\partial c}{\partial x_e} (x_e - f|\Omega_h|) = \sum_{e=1}^{m} \frac{\partial c}{\partial x_e} (x_e - f|\Omega_h|) - f|\Omega_h| = 0
$$

where \(\chi_{|\cdot|}\) is the characteristic function of a set. In this simple case, the above can be solved for \(\lambda(j+1)\), the Lagrange multiplier enforcing the volume constraint for the updated density \(x_{(j+1)}\), and after some simplifications we obtain

$$
\lambda_{(j+1)} = \left( \frac{\sum_{e \in \mathcal{M}} x_{(j+1),e} (\partial_c c(\xi)/\partial c V(\xi)))^{1/2} \right)^2
$$

where \(|\cdot|\) denotes the number of elements in a set.

Equations \(25\) and \(27\) can be iteratively used to compute the new solution \((x_{k+1}, \lambda^*_k)\), as implemented in the code here below (again, note that \(\ln\) here represents \(\sqrt{\lambda}\))

```matlab
u=max(xT-move,1); l=max(xT-move,0);
ocP= @(s) xT.*sqrt(-dc(s)./dV0(s));
lm=mean(oCP(act))/volfrac;
lmOld = 0;
while abs(lm-lmOld)>1e-10
tmp=oCP(act)/lm;
[setu,setl]=deal(find(tmp>u).find(tmp<l));
setM=[abs(sign(1-tmp)+sign(u-tmp))];
den=volfrac*(sum(u(setu))+sum(1(setl)))/nel;
lmOld=lm;
lm=(sum(oCP(setM))/den)/nel;
end
x=oCP(act)/lm;
[setu, setl]=deal(find(x>u).find(x<l));
x(setl)=l(setl); x(setu)=u(setu);
```
function top99neo(nelx,nely,volfrac,penal,ft,eta,beta,move,maxit)

% ---------------------------------------- PRE. 1) MATERIAL AND CONTINUATION PARAMETERS
1 E0 = 1; % Young modulus of solid
2 E0min = 1e-9; % Young modulus of "void"
3 nu = 0.3; % Poisson ratio
4 penalCnt = {1,3,25,0.25}; % continuation scheme on penal
5 sI = 1, 2, 25; % continuation scheme on beta
6 % ---------------------------------------- PRE. 2) DISCRETIZATION FEATURES
7 nEl = nelx * nely; % number of elements
8 cMat = reshape(1:nEl,nely,nely); % element numbering
9 nodeNrs = int32(reshape([1:(1+nelx)]*(1+nely),1+nely,1+nexl)); % number of nodes (defined as int32)
10 cVec = reshape(2*nodeNrs(1:end-1,:)); % connectivity matrix
11 Umat = diag(cMat(1,:),1); % total number of DOFs
12
% -------------------------- RL. 1) COMPUTE PHYSICAL DENSITY FIELD (AND ETA IF PROJECT.)
13 [ dy,dx ] = meshgrid(-ceil(nmin)-1:ceil(nmin)-1); % conv. kernel
14 h = max(0, min - sqrt(dy.^2 + dx.^2)); % conv. kernel
15
%----------------------------------------- PRE. 3) LOADS, SUPPORTS AND PASSIVE DOMAINS
16 dprj = @(v,eta,beta) beta*(1-tanh(beta*(v-eta)).^2)./(tanh(beta*eta)+tanh(beta*(1-eta)));
17
% ---------------------------------------- PRE. 4) DEFINE IMPLICIT FUNCTIONS
18
% --------------------------------------- PRE. 5) DEFINE IMPLICIT FUNCTIONS
19
% ---------------------------------------- PRE. 6) ALLOCATE AND INITIALIZE OTHER PARAMETERS
20
B The 2D code for compliance minimization

A new 99 line Matlab code for compliance Topology Optimization
\( \text{prj} = \lambda(x, \eta, \beta) = (\tanh(\beta \eta) + \tanh(\beta(x(1) - \eta))) / \text{act} = \text{setdiff}(1:nEl, \text{union}(\text{pass}, \eta)) \); % set of active d.v.

\( \text{free} = \text{setdiff}(1:nDof, \text{fixed}) \); % set of free DOFs

\( F = \text{fsparse}([\text{lcDof}, 1], [nDof, 1], -\sin((0:nely) / nely * \pi)) \); % define load vector

\( \text{pasS, pasV} = \text{deal}([], []) \); % passive solid and void elements

\( \text{fixed} = 1:3 * (nely + 1, nely + 1) \); % nodes numbering #3D#

\( \text{elNrs} = \text{reshape}(1:nEl, nely, nelz, nelx) \); % element numbering #3D#

\( \text{nodeNrs} = \text{int32}(\text{reshape}(1:(1 + nelx) * (1 + nely), 1, 1 + nelx) * (1 + nely) * (1 + nelz) + [0, 1, 2, \ldots, 3*(nely + 1), 3*(nely + 1) + [0, 1, 2, -3, -2, -1, -3, -2, -1, 3*(nely + 1) + [0, 1, 2, -3, -2, -1]]) \); % connectivity matrix #3D#

\( \text{dof0} = \text{reshape}([\text{dof0, 1}, \text{dof0, 2}, \ldots, \text{dof0, 24}]) \); % elemental stiffness matrix #3D#
```matlab
61 deta = @(v, eta, beta) - beta * csch(beta) .* sech(beta * (v - eta)).^2 .* sinh(v * beta) .* sinh((1 - v) * beta); % projection eta-derivative
62 dprj = @(v, eta, beta) beta *(1 - tanh(beta *(v- eta)) .^2) ./ (tanh(beta* eta) + tanh(beta*(1- eta))); % proj. x-derivative
63 cnt = @(v, vCnt, l) v + (l >= vCnt{1}) .*(v < vCnt{2}) .*(mod(l, vCnt{3}) ==0) .* vCnt{4}; %
64 % -------------------------------------------------- PRE. 5) PREPARE FILTER
65 [dy, dz, dx] = meshgrid(-ceil(rmin) +1:ceil(rmin)-1, -ceil(rmin) +1:ceil(rmin)-1, -ceil(rmin) -1);
66 h = max(0, rmin - sqrt(dx.^2 + dy.^2 + dz.^2)); % conv. kernel #3D#
67 Hs = imfilter(ones(nely, nelz, nelx), h, 'symmetric'); % matrix of weights (filter)
68 dHs = Hs;
69 % ------------------------ PRE. 6) ALLOCATE AND INITIALIZE OTHER PARAMETERS
70 [x, dsK, dV] = deal(zeros(nEl, 1)); % initialize vectors
71 [xOld, ch, loop, U] = deal(1, 1, 0, zeros(nDof, 1)); % old it., it. counter, xchange, U
72 dV(act, 1) = 1/nEl/volfrac; % derivative of volume
73 x( act ) = (volfrac*(nEl-length(pasV))-length(pasS))/length(act); % volume fraction on active set
74 x(pasS) = 1; % set x = 1 on pasS set
75 % ================================================= START OPTIMIZATION LOOP
76 while ch > 1e-6 && loop < maxit
77 loop = loop + 1; % update iter. counter
78 xPhys = imfilter(reshape(x/nEl, nely, nelz, nelx) ./ Hs, h, 'symmetric');% filtered field #3D#
79 [xPhys(pasS), xPhys(pasV)] = deal(1, 0); % restore passive values
80 xPhys = xPhys(:); % reshape to column vector
81 if ft > 1 % compute optimal eta * with Newton
82 f = (mean(prj(xPhys, eta, beta)) - volfrac) * (ft ==3); % function (volume)
83 while abs(f) > 1e-6 % Newton process for finding opt. eta
84 eta = eta - f / mean(deta(xPhys, eta, beta));
85 f = mean(prj(xPhys, eta, beta)) - volfrac;
86 end
87 xPhys = prj(xPhys, eta, beta); % projected (physical) field
88 dHs = Hs ./ reshape(dprj(xPhys, eta, beta), nely, nelz, nelx); % sensitivity modification #3D#
89 end
90 x = prj(xPhys, eta, beta); % projected (physical) field
91 dVs = dHs ./ reshape(dprj(xPhys, eta, beta), nely, nelz, nelx); % sensitivity modification #3D#
92 end
93 % -------------------------- RL. 2) SETUP AND SOLVE EQUILIBRIUM EQUATIONS
94 sk = (Emin + xPhys.^penal * (E0 - Emin));
95 dsK(act) = -penal * (E0 - Emin) * xPhys(act).^(penal-1);
96 sK = reshape(Ke(:) * sK, [nDof, nDof]);
97 K = chol(K( free, free), 'lower'); % f/b substitution
98 U( free) = L' \ (L \ F( free)); % f/b substitution
99 % -------------------------- RL. 3) COMPUTE SENSITIVITIES
100 dc = dSx .* sum((U(cMat) * Ke0) .* xPhys(:, :)); % derivative of compliance
101 dc = imfilter(reshape(dc, nely, nelz, nelx), h, 'symmetric'); % filter objective sens. #3D#
102 dVs = dHs .* reshape(dprj(xPhys, eta, beta), nely, nelz, nelx); % filter compliance sens. #3D#
103 % ----------------- RL. 4) UPDATE DESIGN VARIABLES AND APPLY CONTINUATION
104 xT = x(act);
105 [xU, xL] = deal(xT + move, xT - move); % current upper and lower bound
106 ocP = xT(act) .* sqrt((0 * cMat) + Ke0) .* xPhys(act); % "current" upper and lower bound
107 while (l(2) - l(1)) / (l(2) + l(1)) > 1e-10 % constant part in resizing rule
108 l = [0, mean(ocP) / volfrac]; % initial estimate for LM
109 lmid = 0.5 * (l(1) + l(2)); % OC resizing rule
110 if mean(x) > volfrac, l(1) = lmid; else, l(2) = lmid; end
111 [penal, beta] = deal(cnt(penal, penalCnt, loop), cnt(beta, betaCnt, loop)); % apply continuation on parameters
112 % -------------------------- RL. 5) PRINT CURRENT RESULTS AND PLOT DESIGN
113 fprintf(’It.:%5i C :%6.5e V :%7.3f ch .:%0.2e penal :%7.2f beta :%7.1f eta :%7.2f lm :%0.2e \n’, ...
loop, F*’U, mean(xPhys(:, :)), ch, penal, beta, eta, lmid);
114 isovals = shiftdim(reshape(xPhys, nely, nelz, nelx), 2);
115 isovals = smooth3(isovals, ‘box’, 1);
116 patch(isosurface(isovals, .6), ’FaceColor’, ‘b’, ’EdgeColor’, ‘none’);
117 patch(isocaps(isovals, .6), ’FaceColor’, ’r’, ’EdgeColor’, ’none’);
118 drawnow; view([145, 25]); axis equal tight off; cla();
119 end
120 end
```