Università della Svizzera italiana

Faculty of Informatics

USI Technical Report Series in Informatics

High Performance Topology Optimization

Ezekiel Barnett[1], Sumeet Gyanchandani[1], Sameer Rawat[1], Dimosthenis Pasadakis[1], Olaf Schenk[1]

[1] Faculty of Informatics, Universitá della Svizzera italiana, Switzerland

Abstract

Topology Optimization (TO) is a mathematical method which optimizes the layout of a material given a set of loads and volume constraints such that global compliance is minimized. The solution algorithm is formulated as an iterative method based around a finite element discretization of the design domain and an optimization procedure to allocate material such that compliance is minimized. Topology Optimization is a computationally expensive procedure. Each iteration involves solving a high dimensional linear system arising from the FEM discretization and an optimization procedure. Most problems require hundreds if not thousands of iterations to converge.

We work with a TO algorithm in Python based on the canonical MBB truss structure in 2 dimensions using a procedure based on the 99 line code from Sigmund [1]. We also implement a FEniCS Linear Elasticity solver to simulate point load compliance and von Mises stresses. Lastly, we experiment with several linear solvers within the TO code and achieve significant improvements in the computation time.

Report Info

Published
June 2018

Number
USI-INF-TR-2018-3

Institution
Faculty of Informatics
Universitá della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

Many engineering problems can be formulated as constrained optimization problems, such as minimizing material use while maximizing global stiffness. Topology Optimization (TO) is one state of the art method for solving such problems algorithmically. TO optimizes the layout of a material in some domain for a given set of loads, material constraints and boundary conditions, while minimizing compliance. Typically, TO uses a Finite Element Method (FEM) to evaluate design performance, and some gradient-based algorithm to iteratively improve the distribution of the material across the domain.

We work with a SIMP TO algorithm in Python and test it on a canonical example of TO: the MBB truss structure in 2 dimensions, with a point load on one side.

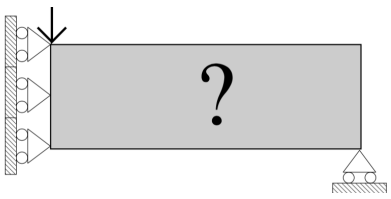


Figure 1: The visualization of the problem

The procedure is computationally intensive; each iteration (overall often consisting of several hundred) requires inverting a high dimensional stiffness matrix from the FEM discretization, and an optimization step. We implement several linear solvers to improve the speed of solution - with a particular focus on high dimensional problems. We also implement a mesh generator and FEM procedure to solve the linear elasticity on the optimal topology, and explore compliance for various materials.

In the following sections we will first explain the mathematical formulation of the TO algorithm including the linear elasticity problem, the finite element discretization, and the optimization methodologies. We follow the model of the SIMP optimization procedure detailed in [1]. In the third section, we will explore our implementation of both the TO algorithm as well as the FEniCS solver [2], and explain some of the computational bottlenecks, and our approach to overcoming them. In the fourth section, we will explore both performance and simulations on optimized topologies (FEniCS). We will then conclude and mention some potential next steps.

2 Mathematical Formulation and Finite Elements

At a very high level, Topology Optimization can be formulated as a solution to the standard compliance minimization problem with volume constraints:

$$\begin{aligned} &\min_{\boldsymbol{x}}: && c(\boldsymbol{x}) = \boldsymbol{u}^T \boldsymbol{K} \boldsymbol{u} = \sum_{e=1}^{N} (x_e)^p \boldsymbol{u}_e^T \boldsymbol{k}_e \boldsymbol{u}_e \\ &\text{subject to:} && \frac{V(\boldsymbol{x})}{V_0} = V_f \ , \ \boldsymbol{K}\boldsymbol{u} = \boldsymbol{f} \ , \mathbf{0} < \boldsymbol{x}_{\min} \leq \boldsymbol{x} \leq \mathbf{1} \end{aligned}$$

Where $c(x)$ is compliance, u is a vector of global displacements, F is a global force vector, K is the global stiffness matrix, $V(x)$ is the material volume, V_0 is the total design domain volume, V_f is the fraction of the material with volume, and N is the number of elements. x is a single element "design variable". Thus, each element "e" may be assigned a density x (between 0 and 1), such that the conditions are satisfied.

The stiffness matrix K arises from a discretization of the equations of linear elasticity:

$$-\nabla \cdot \sigma = \mathbf{f} \qquad \in \Omega \tag{1}$$

$$\sigma = \lambda(\nabla \cdot \mathbf{u})I + \mu(\nabla \mathbf{u} + (\nabla \mathbf{u})^T) \tag{2}$$

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega} \sigma(\mathbf{u}) : \epsilon(\mathbf{v}) d\,x \tag{3}$$

Where (1) deformations on the domain $\boldsymbol{\Omega}$ are written as relation between the gradient of the stress tensor σ and a point load $\mathbf{f}$. The stress tensor σ (2) is computed from $\mathbf{u}$, the displacement with Lamé's elasticity parameters μ and λ. The variational formulation (3) is used to construct our linear system, solve for displacements, and use these displacements to compute compliance $c(x)$.

TO solves the minimization problem by first discretizing the domain with a FEM mapping, and building a stiffness matrix. We discretize our domain with quadrilateral elements in two dimensions with two degrees of freedom. Below, see an example of a 4×4 grid, and its resulting stiffness matrices' sparsity pattern:

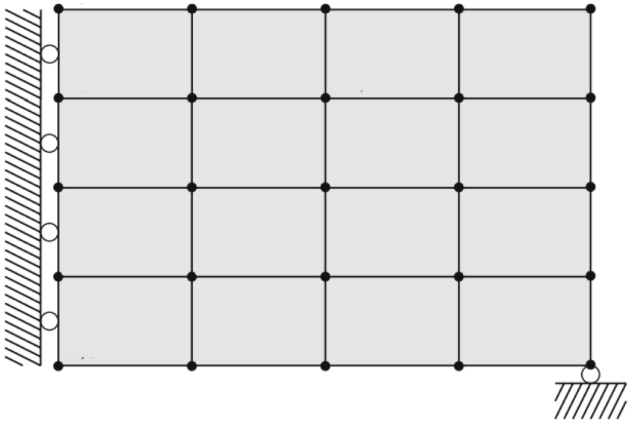


Figure 2: A visualization of a 4 × 4 Element Mesh

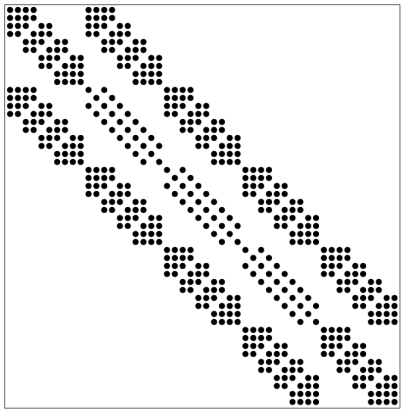


Figure 3: Sparsity pattern of the stiffness matrix

After constructing the domain, and distributing volume evenly across the space, the main "loop" of the Topology Optimization algorithm consists of 4 main stages: a FE procedure to determine compliance, then computation of sensitivities, a filtering (MIF) to ensure mesh-independence, and a design update through the element densities (OC). This loop is repeated until the change in the objective function is smaller than some tolerance (goes to zero).

See below for an overview of the TO algorithm using a modified Solid Isotropic Material with Penalization (SIMP) approach:

Algorithm 1 Topology Optimization

Input: nelx, nely, V_f, p, $r_{\min}$

Output: $c(\boldsymbol{x})$, $\boldsymbol{x}$

function TopOpt(nelx, nely, V_f, p, $r_{\min}$)
  Initialize all elements in $\boldsymbol{x} \leftarrow V_f$
  while not converged **do**
    $\boldsymbol{u} \leftarrow$ **FE(nelx, nely,** $\boldsymbol{x}, \boldsymbol{f}$**)**
    $c(\boldsymbol{x}) \leftarrow 0$
    for every element $\boldsymbol{e}$ in $\boldsymbol{x}$ **do**
      $c(\boldsymbol{x}) \leftarrow c(\boldsymbol{x}) + (x_e)^p \boldsymbol{u}_e^T \boldsymbol{k}_e \boldsymbol{u}_e$
      $\delta c(\boldsymbol{x}) \leftarrow -p(x_e)^{p-1} \boldsymbol{u}_e^T \boldsymbol{k}_e \boldsymbol{u}_e$
    end for
    $\delta c(\boldsymbol{x}) \leftarrow$ **MIF(nelx, nely,** $r_{\min}, \boldsymbol{x}, \delta c(\boldsymbol{x})$**)**
    $\boldsymbol{x} \leftarrow$ **OC(nelx, nely,** $\boldsymbol{f}, \boldsymbol{x}, \delta c(\boldsymbol{x})$**)**
  end while
  return $c(\boldsymbol{x})$, $\boldsymbol{x}$
end function

Algorithm 2 Finite Element Analysis

Input: $\boldsymbol{x}$ ▷ Vector of design variables

Output: $\boldsymbol{u}$ ▷ Global Displacement Vector

function FE(nelx, nely, $\boldsymbol{x}$, V_f)
  $\boldsymbol{K} \leftarrow 0$
  $\boldsymbol{f} \leftarrow 0$
  $\boldsymbol{u} \leftarrow 0$
  for every element $\boldsymbol{e}$ in $\boldsymbol{x}$ **do**
    $\boldsymbol{K} \leftarrow \boldsymbol{K} + x_e^p \boldsymbol{k}_e$
  end for
  $\boldsymbol{f}_{2,1} \leftarrow -1$ ▷ Force in the upper left corner
  return $\boldsymbol{u}$
end function

Algorithm 3 Mesh-Independency Filtering

Input: $r_{\min}$, $\boldsymbol{x}$, $\delta c(\boldsymbol{x})$

Output: $\widehat{\delta c(\boldsymbol{x})}$ ▷ Filtered Sensitivities

function MIF(nelx, nely, $r_{\min}$, $\boldsymbol{x}$, $\delta c(\boldsymbol{x})$)
  for every element $\boldsymbol{e}$ in $\boldsymbol{x}$ **do**
    for every element $\boldsymbol{f}$ within $r_{\min}$ radius of $\boldsymbol{e}$ **do**
      $\hat{H}_f \leftarrow r_{\min} - \text{dist}(e, f)$
      $\widehat{\delta c(\boldsymbol{x_e})} = \frac{1}{x_e \sum_{f=1}^{N} \hat{H}_f} \sum_{f=1}^{N} \hat{H}_f x_f \delta c(\boldsymbol{x_f})$
    end for
  end for
  return $\widehat{\delta c(\boldsymbol{x})}$
end function

Algorithm 4 Optimality Criteria Optimizer

Input: $r_{\min}$, $\boldsymbol{x}$, $\delta c(\boldsymbol{x})$

Output: x^{new} ▷ Updated design variables

function OC((nelx, nely, V_f, $\boldsymbol{x}$, $\delta c(\boldsymbol{x})$))
  $m \leftarrow 0.2$ ▷ Initialize move m to 0.2
  for every element $\boldsymbol{e}$ in $\boldsymbol{x}$ **do**
    if $x_e B_e^{\eta} \leq \max(x_{\min}, x_e - m)$ **then**
      $x_e^{\text{new}} = \max(x_{\min}, x_e - m)$
    end if
    if $\max(x_{\min}, x_e - m) < x_e B_e^{\eta} < \min(1, x_e + m)$ **then**
      $x_e^{\text{new}} = x_e B_e^{\eta}$
    end if
    if $\min(1, x_e + m) \leq x_e B_e^{\eta}$ **then**
      $x_e^{\text{new}} = \min(1, x_e + m)$
    end if
  end for
  return x^{new}
end function

Ultimately, our TO algorithm functions by iteratively redistributing volume across mesh elements toward areas with high "sensitivity" using a heuristic updating scheme (OC). The heuristic updating scheme (OC) updates variables in a local domain based on the sensitivities:

$$x_e^{\text{new}} = \begin{cases} \max(x_{\min}, x_e - m), & \text{if } x_e B_e^{\eta} \leq \max(x_{\min}, x_e - m), \\ x_e B_e^{\eta}, & \text{if } \max(x_{\min}, x_e - m) < x_e B_e^{\eta} < \min(1, x_e + m), \\ \min(1, x_e + m), & \text{if } \min(1, x_e + m) \leq x_e B_e^{\eta} \end{cases}$$

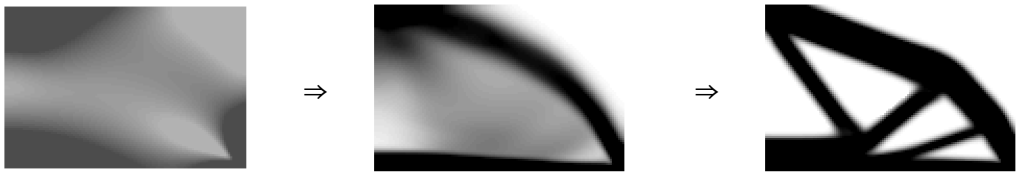
where m is a positive move limit, η is a numerical damping coefficient, B_e is found from the optimality condition and is computed after solving a Lagrange Multiplier problem via a bisection algorithm. Thus, the OC optimizer updates the volume of material allocated to a given element based on the sensitivities in some local domain. We select $m = 0.2$, and $\eta = 0.5$. We note it is a heuristic procedure.

3 Implementation and Bottlenecks

3.1 Python Implementation of the TO Algorithm

We worked with the Topology Optimization Algorithm based on the 99 line code from Sigmund [1]. We were advantaged by the numpy and scipy libraries, which allow the construction of the stiffness matrix and vectorizations to be done efficiently. The optimization algorithm computes an optimal topology in an identical fashion to the 99 line MATLAB code:

See below for the 2^{nd}, 5^{th}, and 100^{th} iteration of the algorithm on a domain of 200×50 elements.



As shown in the previous section, the program consists of six discrete segments. Of all 6, the clear computational bottleneck is the solving of the linear system $\boldsymbol{Ku} = \boldsymbol{f}$. As mesh size grows (and the resulting linear system grows exponentially), the solving of the linear system comes to dominate the computation time.

The first chart below shows the time taken by each part of the program, in comparison to the entire compute time. We show this for several mesh sizes. The right chart shows the scaling of the stiffness matrix K as the mesh size grows. Notice that a 320×320 mesh produces a stiffness matrix with over 40 billion elements.

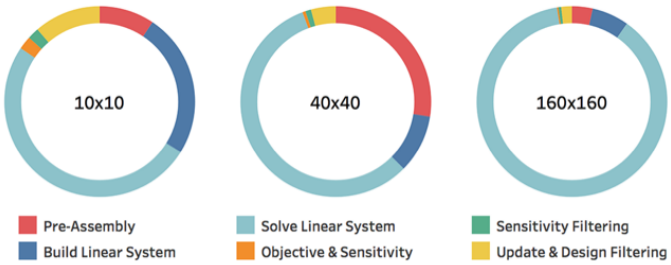


Figure 4: Computation time by components

$nelx \times nely$	**K**
10×10	242×242
20×20	882×882
40×40	3362×3362
80×80	13122×13122
160×160	51842×51842
320×320	206082×206082

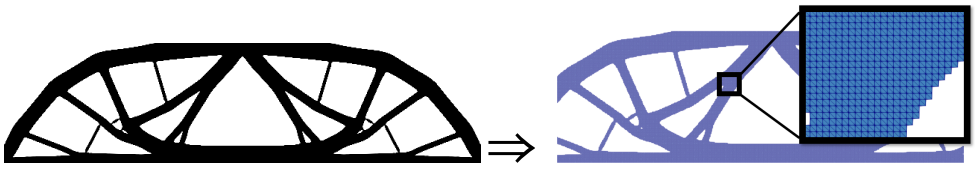
Table 1: Mesh size vs Stiffness Matrix size

Computing the compliance via inverting the stiffness matrix becomes the main computational bottleneck of the program at even moderate mesh sizes. To allow the algorithm to scale, an efficient linear solver is required to compute this section of the algorithm. However, at very large mesh sizes, both memory and and solver technique will be implicated, likely necessitating some domain decomposition.

3.2 FEniCS Linear Elasticity

In order to understand our results and explore the optimized geometries more deeply, we generate meshes from the optimal topology outputted by the algorithm (FEniCS requires triangular elements, we have quadrilaterals), and then used the FEniCS library to solve the linear elasticity problem on the produced meshes, with various material properties.

FEniCS takes as input the variational/weak formulation, and then constructs and solves the discrete system. We also compute von Mises stresses, which are typically included in elasticity analysis to examine the weak points in a structure.



Optimized Topology for MBB problem [left]. Mesh generation using quadrilateral elements [right].

Listing 1: Code Snippet of Linear Elasticity Solver in FEniCS

```
 #Variational Formulation
    u = TrialFunction(V)
    d = u.geometric_dimension()
    v = TestFunction(V)
    mu = E / (2. * (1 + nu))
    lmbda = E * nu / ((1. + nu) * (1. - 2. * nu))
    eps = 0.5 * (nabla_grad(v) + nabla_grad(v).T)
    sig = lmbda * nabla_div(u) * Identity(d) + 2 * mu *
        0.5 * (nabla_grad(u) + nabla_grad(u).T)
    K = inner(sig, eps) * dx

#Boundary conditions on the subdomains
    dss = ds(domain=mesh, subdomain_data=boundary_parts)
    f = dot(B, v) * dx + sum([dot(T, v) * dss(len(fixed_subdomains) + i)
        for i, T in enumerate(Ts)])

#Compute solution
    u = Function(V)
    solve(K == f, u, fixed_bcs)
    return u
```

4 Numerical Results

4.1 Solver Choice and Scaling

The naive solver for computing the $Ku = f$ system is SPSOLVE from the scipy library, which is relatively slow, and not capable of handling larger mesh sizes. Furthermore, this solver does not take advantage of many of the properties of the matrix corresponding to the linear system (symmetric, positive definite, hermetian). We implemented the following solvers and tested overall compute time to see if we could get some performance improvement via choice of solver.

We implement the following solvers and run the entire simulation.

- SCIPY-LGMRES
- SCIPY-SPSOLVE
- SCIPY-SUPERLU
- UMFPACK-LU [3]
- CVXOPT-CHOLMOD [3]

We performed experiments on a node of the ICS cluster, which has 2 × Intel E5-2630 v3, 16 (2 × 8) cores with 128GB DDR4 RAM. Results displayed below.

It is clear that the CHOLMOD solver from the CVXOPT library is the fastest solver; solving approximately two times faster than the next best solver - the UMFPACK implementation of the LU decomposition. CHOLMOD is a Cholesky Factorization routine which (theoretically) should solve the linear system about twice as fast as the LU decomposition and we find that it solves even faster than that. Furthermore, we observe that LGMRES is unable to solve the systems of meshes larger than 80×80 in a reasonable compute time and SPSolve and SUPERLU are unable to do calculations.

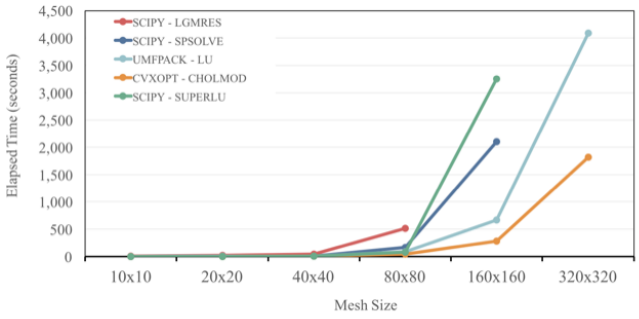
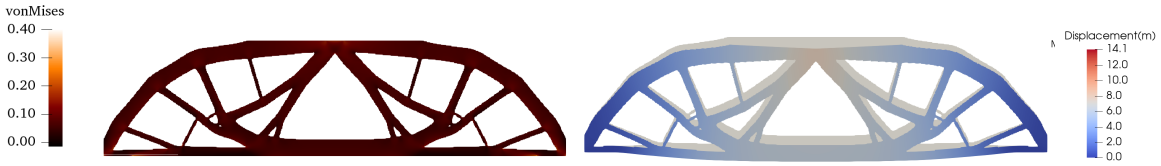


Figure 5: Performance of various Linear Solvers

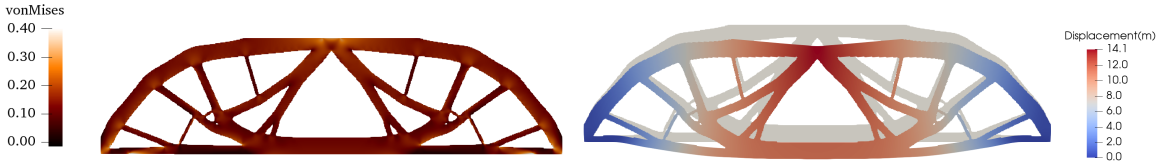
4.2 FEniCS

We performed experiments on the compliance and von Mises stresses of the optimal topology for a mesh size of 320×180 meters (each element is defined as 1×1 meter), for various materials. First we tested our optimal topology as though it was constructed from isotropic steel (the material properties are communicated through Young's Modulus and Poisson Ratio, which enter the linear elasticity problem). We then apply a point load directly atop the center of the truss structure, and simulate. We obtain the following:



Case	PointLoad(MN)	YoungsMod(GPa)	PoisRatio	MaxDisplace(m)	MinV.M.(GPa)	MaxV.M.(GPa)
SteelASTM-A36	5	200	.31	7.98	.00001	.1938

Next we conduct the same test for a "softer"(more compliant) material (lower Young's Modulus), Titanium. Thus we simulate with the same point load force atop the center of the beam. As we can see, the compliance of this Titanium structure is far higher than the steel, as visualized through the displacement.



Case	PointLoad(MN)	YoungsMod(GPa)	PoisRatio	MaxDisplace(m)	MinV.M.(GPa)	MaxV.M.(GPa)
TitaniumTi-6Al	5	113.3	0.37	14.45	.0001	.28672

5 Discussion and Conclusion

In this report we described our work on a Python implementation of the Topology Optimization Algorithm with the Solid Isotropic Microstructure with Penalization (SIMP) optimization procedure . We realized 2× speed improvements by implementing a sparse Cholesky factorization solver to invert the stiffness matrix, and implemented a FEniCS solver to simulate loads on the optimized topologies. Although we were able to solve the problem with a very large stiffness matrix, one could imagine using a far finer resolution mesh. This would only be computationally feasible using domain decomposition methods for solving the linear system.

References

[1] O. Sigmund, "A 99 line topology optimization code written in matlab," *Structural and Multidisciplinary Optimization*, 2001.

[2] M. S. Alnaes, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells, "The fenics project version 1.5," *Archive of Numerical Software*, vol. 3, 2015.

[3] M. S. Andersen, J. Dahl, and L. Vandenberghe, "Cvxopt: A python package for convex optimization, version 1.1.5." http://abel.ee.ucla.edu/cvxopt, 2012.