Homework 3 - Density-Based Topology Optimization

Released: May 18, 2022.
Submission deadline: May 31, 2022, 11:59pm.

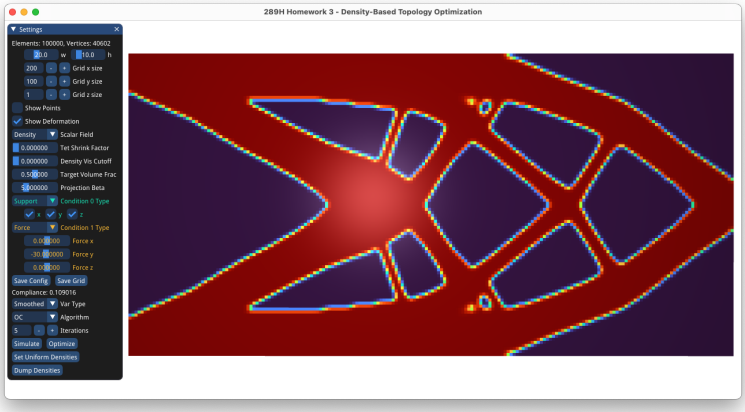


Figure 1: Expected result for input `cantilever.obj` with settings `cantilever_2d_hi.config` once you've completed the assignment. Here, β was increased to 5 after a few rounds of OC iterations.

In this assignment, you will implement a 3D topology optimization tool for volumetric linear elasticity based on your own linear tetrahedral finite element simulator. As a (substantial) optional bonus exercise, you can implement an approach that attemps to ensure the final design is "self-supporting" so that it can be directly 3D printed from bottom-to-top without additional support structure.

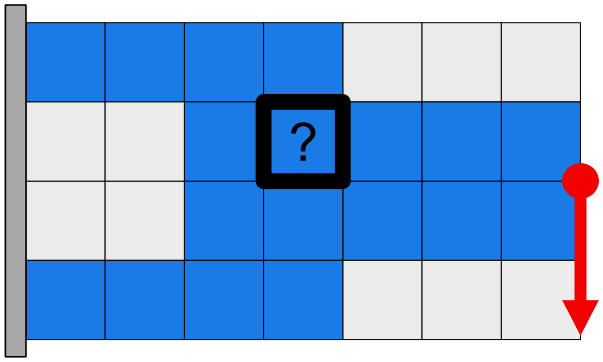
1 Getting Started

This assignment optionally depends on CHOLMOD for higher-performance Cholesky factorizations. I recommend installing it (with `sudo apt install libsuitesparse-dev` on Ubuntu or `sudo port install SuiteSparse` on macOS) before attempting the large examples. The `cmake` build script should print `Using CHOLMOD` if you have installed it correctly.

Download and unzip `hw3.zip` into `DFAB_HW` and make sure it builds/runs:

```
cd DFAB_HW/hw3
mkdir build && cd build && cmake .. -GNinja
ninja && ./hw3 ../data/cantilever.obj ../data/cantilever_2d_lo.config
```

2 Density-Based Topology Optimization



Density-based topology optimization divides the design domain into a grid of cubes (analogous to the ground structure used for truss optimization) and then decides whether or not to deposit material in each element. As stated, this would be an intractable combinatorial optimization problem, but the popular solution strategy is to *relax* the binary material occupancy variables into continuous *density* field ρ with values in $[0, 1]$. This enables the use of efficient smooth, gradient-based optimization of the density variables, just like you optimized the cross-sectional areas of beams in `hw2`.

The intermediate density values are not necessarily physically meaningful (although they can be interpreted as a micro-scale layout of solid and void material, just like a black-and-white printer can approximate grayscale images by dithering). Therefore, the optimization is typically formulated in a way that eventually drives the densities to either zero or one. For the nonphysical intermediate densities employed during the optimization, an interpolation rule is used to define the material properties (e.g., Young's moduli), the most popular of which is the SIMP (Solid Isotropic Material with Penalization) rule:

$$C(\rho) := (Y_{\min} + (1 - Y_{\min})\rho^p)C_0, \tag{1}$$

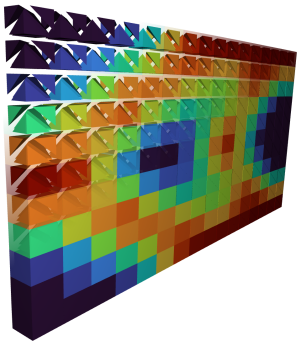
where C_0 is the elasticity tensor of the full solid material, and $C(\rho)$ is the weaker material associated with density ρ. It turns out that setting power $p > 1$ encourages binary designs by making intermediate densities "inefficient," and we will stick with the popular setting $p = 3$ (available as `TopologyOptimizer::p`) for this assignment. Constant $Y_{\min}$, available as `TopologyOptimizer::Y_min`, prevents material stiffnesses from vanishing (which would result in a singular full stiffness matrix K).

Using this elasticity tensor, we can write the *linearized* elasticity energy stored in the object under displacement $\mathbf{u}$ as:

$$E_{\text{elastic}}(\mathbf{u}) = \frac{1}{2}\int_\Omega \varepsilon(\mathbf{u}) : C(\rho) : \varepsilon(\mathbf{u})\, \mathrm{d}\mathbf{x}, \tag{2}$$

where $\varepsilon(\mathbf{u}) := \frac{1}{2}\left(\nabla\mathbf{u} + (\nabla\mathbf{u})^\top\right)$ is the small strain tensor.

2.1 Finite Element Discretization



When working with regular grids of density variables, it is most natural to use quadrilateral (2D) or hexahedral (3D) finite elements for simulation. However, those are more complicated than the linear elements introduced in lecture since their basis functions are no longer piecewise linear. For this reason, our code framework simply tetrahedralizes each hex element and performs all simulation using linear tets. In order to still work with density variables defined on a regular grid, we apply a change of variables using the `DensityFilter` framework described in Section 2.5.

Denoting the flattened vector of per-vertex displacement variables as $\tilde{\mathbf{u}} = [u_{0,x}, u_{0,y}, u_{0,z}, u_{1,x}, \dots]^\top$, the corresponding displacement field is given by:

$$\mathbf{u}(\mathbf{x}) = \sum_i u_i \vec{\phi}_i(\mathbf{x}),$$

where the sum is over individual scalar entries of $\tilde{\mathbf{u}}$, and $\vec{\phi}_i$ is the *vector-valued basis function* corresponding to the i^{th} scalar displacement variable. These vector-valued basis functions are related to the scalar-valued basis functions by the definition:

$$\vec{\phi}_{3v+d} := \mathbf{e}_d \phi_v,$$

where $\mathbf{e}_d$ is the d^{th} column of the 3×3 identity matrix, and ϕ_v is the v^{th} vertex's scalar-valued basis function. The elastic energy defined by (2) can now be computed by the quadratic form:

$$E_{\text{elastic}}(\tilde{\mathbf{u}}) = \frac{1}{2}\tilde{\mathbf{u}}^\top K(\tilde{\rho})\tilde{\mathbf{u}},$$

where $K(\tilde{\rho})$ is the *stiffness matrix* for current *per-tetrahedron* density variable vector $\tilde{\rho}$ given by:

$$K_{ij}(\tilde{\rho}) := \int_\Omega \varepsilon(\vec{\phi}_i) : C(\rho) : \varepsilon(\vec{\phi}_j)\, \mathrm{d}\mathbf{x}.$$

The equilibrium problem is then simply the linear system:

$$K(\tilde{\rho})\tilde{\mathbf{u}} = \tilde{\mathbf{f}},$$

where $\tilde{\mathbf{f}}$ is the right-hand-side *load* vector accounting for the applied forces. (It is the vector whose dot product with $\tilde{\mathbf{u}}$ computes the work done by the applied surface tractions throughout the displacement $\tilde{\mathbf{u}}$.) Note that rows and columns of this system will need to be modified to enforce the Dirichlet (clamp) conditions $\tilde{\mathbf{u}}_c = 0$ on fixed displacement components c.

2.2 Minimum Compliance Topology Optimization

We wish to solve the minimum compliance topology optimization problem, which expressed in reduced form as an optimization over density variables alone is:

$$\begin{aligned} &\min_{\hat{\rho}} \tilde{\mathbf{f}} \cdot \tilde{\mathbf{u}}(\hat{\rho}) \quad \text{with } \tilde{\mathbf{u}}(\hat{\rho}) := K(\hat{\rho})^{-1}\mathbf{f} \\ &\text{s.t. } \texttt{Vol}(\hat{\rho}) \leq \texttt{maxVolumeFrac} * \texttt{Vol}_0. \end{aligned} \tag{3}$$

Here, $\texttt{Vol}_0$ and `maxVolumeFrac` are the design domain volume (`TrussOptimizer::domainVolume()`) and maximum volume fraction (`TrussOptimizer::maxVolumeFrac`), respectively, and $\hat{\rho}$ is a set of variables controlling the density field. Note that $\hat{\rho}$ can differ from $\tilde{\rho}$ depending on the problem configuration as discussed in Section 2.5.

2.3 Per-Element Stiffness Matrix Assembly

It is simplest to construct and reason about $K(\tilde{\rho})$ as a sum of per-element (per-tetrahedron) stiffness matrices K_e that have been "globalized" into a large sparse matrix:

$$\begin{aligned} K_{ij}(\tilde{\rho}) = \sum_e \underbrace{{S_e}^\top K_e(\tilde{\rho}_e) S_e}_{\texttt{globalize}(K_e,e)}, \qquad [K_e(\tilde{\rho}_e)]_{ab} &:= \varepsilon(\overrightarrow{\phi}^e_a) : C(\tilde{\rho}_e) : \varepsilon(\overrightarrow{\phi}^e_b)\texttt{Vol}(e) \\ &= (Y_{\min} + (1 - Y_{\min})\tilde{\rho}_e^p)[K_e(1)]_{ab}, \end{aligned} \tag{4}$$

where the globalization operation is represented explicitly here by the sparse $12 \times 3|V|$ *selection matrix* S_e that has a single 1 in each row so that $\mathbf{u}_e := S_e\tilde{\mathbf{u}}$ pulls out the 12 displacement components corresponding to the corners of tet element e. In your code, however, you should implement the `globalize` operation just like you always have (by generating an output triplet with the correct global row/column indices for each entry of the 12×12 matrix K_e); the formula involving S_e will be useful in a moment for our derivative calculation. In the expression for K_e, $\varepsilon(\overrightarrow{\phi}^e_a)$ is the strain of the basis function for corner a of element e evaluated at (restricted to) element e; because basis functions ϕ are linear, this strain is constant and the integral over tetrahedron e used to compute K_e is ultimately just a multiplication of the constant integrand by the element volume `Vol(e)`. Formulas for the gradients of the scalar-valued basis functions were derived in lecture.

2.4 Gradient Computation

To solve (3), we need derivatives of the design's compliance and volume with respect to the density variables. We provide here the formulas for differentiating with respect to the (constant) density of each tetrahedron $\tilde{\rho}_e$ and then later discuss in Section 2.5 how derivatives with respect to the actual optimization variables $\hat{\rho}$ can be computed using the chain rule.

The derivative of volume is particularly simple and can be obtained immediately from the formula $\texttt{Vol}(\tilde{\rho}) = \texttt{m\_vols.dot}(\tilde{\rho})$, where `m_vols` is the vector of individual tetrahedron volumes. The

derivative of compliance is more complicated, but can be obtained using the same approach we used in the truss optimization setting, differentiating both sides of the equation $K(\tilde{\rho})\tilde{\mathbf{u}}(\tilde{\rho}) = \tilde{\mathbf{f}}$ to find $\frac{\partial \tilde{\mathbf{u}}}{\partial \tilde{\rho}_e}$:

$$\frac{\partial K}{\partial \tilde{\rho}_e}\tilde{\mathbf{u}} + K\frac{\partial \tilde{\mathbf{u}}}{\partial \tilde{\rho}_e} = 0 \quad \Longrightarrow \quad \frac{\partial \tilde{\mathbf{u}}}{\partial \tilde{\rho}_e} = -K^{-1}\frac{\partial K}{\partial \tilde{\rho}_e}\tilde{\mathbf{u}}.$$

Plugging this into the compliance objective derivative:

$$\frac{\partial \tilde{\mathbf{f}} \cdot \tilde{\mathbf{u}}}{\partial \tilde{\rho}_e} = -\tilde{\mathbf{f}} \cdot K^{-1}\frac{\partial K}{\partial \tilde{\rho}_e}\tilde{\mathbf{u}} = -\tilde{\mathbf{u}} \cdot \frac{\partial K}{\partial \tilde{\rho}_e}\tilde{\mathbf{u}}.$$

Finally, we use the expression in (4) to obtain an explicit formula:

$$\frac{\partial \tilde{\mathbf{f}} \cdot \tilde{\mathbf{u}}}{\partial \tilde{\rho}_e} = -\tilde{\mathbf{u}}^\top \frac{\partial \sum_l S_l^\top K_l(\tilde{\rho}) S_l}{\partial \tilde{\rho}_e}\tilde{\mathbf{u}} = -\tilde{\mathbf{u}}^\top S_e^\top \frac{\partial K_e(\tilde{\rho}_e)}{\partial \tilde{\rho}_e} S_e \tilde{\mathbf{u}} = \boxed{-p(1-Y_{\min})\tilde{\rho}_e^{p-1}\mathbf{u}_e^\top K_e(1)\mathbf{u}_e,} \quad (5)$$

where we used (1) to differentiate $C(\rho_e)$ and extracted the element-local displacement vector $\mathbf{u}_e = S_e\tilde{\mathbf{u}}$.

2.5 Density Filters

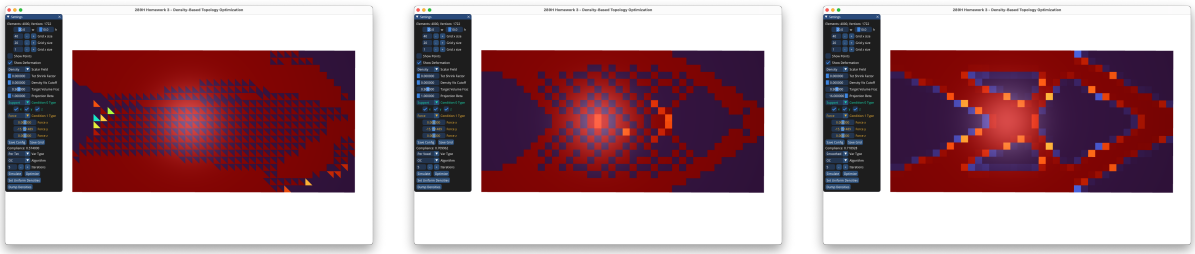
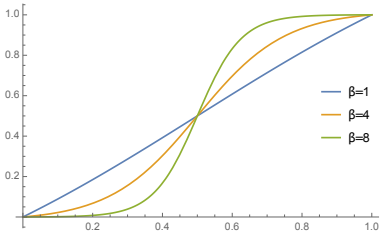


Figure 2: Topology optimization run with three different density representations: per-tet, per-voxel, and per-voxel with smoothing.

Working directly with the per-tet density variables $\tilde{\rho}_e$ obtains poor results as seen in the leftmost screen capture in Figure 2. This is because a high-frequency alternation between solid and void at the resolution of the finite element mesh produces an artificially stiff behavior that the optimizer will exploit. In the case of a per-voxel density field representation (middle), the resulting alternating pattern is the notorious *checkerboard* phenomenon.

The popular strategy for avoiding these issues is to use a "filtering" approach where the final densities used for simulation are obtained by blurring the input or "blueprint" design densities with a smoothing filter (*e.g.,* taking a weighted average over the neighboring design variables). Additional filters can be introduced to address other drawbacks of density-based topology optimization.

Specifically, in this assignment we use a *projection* filter (implemented by `ProjectionFilter` in `DensityField.hh`) that helps enforce a binary design by tuning the steepness parameter β of a sigmoid function that acts to push values away from $\frac{1}{2}$, and optionally an additive manufacturing filter (`VoxelSelfSupportingFilter`, discussed in Section 5.2) that promotes self-supporting designs that can be directly 3D printed bottom-to-top without support material.



Once the desired filters f_i are configured (via the `Var Type` popup box in the GUI), the per-tet densities $\tilde{\rho}$ will be a nonlinear function of the design variables $\hat{\rho}$:

$$\tilde{\rho} = f_n \circ \cdots \circ f_1 \hat{\rho} = f_n(\cdots f_2(f_1(\hat{\rho}) \cdots).$$

Given $\hat{\rho}$, each filter is applied in sequence using the `DensityFilter::apply` method. Then, given the gradient $\frac{\partial J}{\partial \tilde{\rho}}$ of a quantity J with respect to per-tet density fields, the derivative with respect to the actual optimization variables $\hat{\rho}$ can be obtained by repeatedly applying the chain rule:

$$\frac{\partial J}{\partial \hat{\rho}} = \frac{\partial J}{\partial \tilde{\rho}} \mathbf{f}_n' * \cdots * f_1' \quad \Longrightarrow \quad \nabla_{\hat{\rho}} J = \mathbf{f}_1'^{\top} * \cdots * \mathbf{f}_n'^{\top} \nabla_{\tilde{\rho}} J.$$

The application of each transposed Jacobian $\mathbf{f}_i'^{\top}$ (the "backprop" through the i^{th} density filter) is implemented by each filter subclass' override of the `DensityFilter::backprop` method. Conceptually, the `backprop` method takes a gradient with respect to the filter's output variables and returns a gradient with respect to the filter's input variables.

3 Tasks

3.1 Compute the Basis Function Strains (15pts).

Implement `TopologyOptimizer::getStrainVecPhis` in `TopologyOptimizer.cc` to construct the (constant) strain of each of the 12 local basis functions in element `e` as a `SymmetricMatrix`. The numbering of the basis functions should match our flattening convention: the vector-valued basis function $\mathbf{e}_d \phi_c$ (interpolating component d of corner vertex c's displacement) is assigned index $3c + d \in \{0...11\}$.

The `SymmetricMatrix` class implements a compact "flattened" storage format (similar to Voigt notation) and supports double contraction with another symmetric matrix with its `doubleContraction` method. It is also designed to work with the also-provided `ElasticityTensor` class (used to declare the base material's elasticity tensor, `C`) whose `doubleContraction` method implements the double contraction necessary to map strain to stress. Note that you can build your strain tensors as ordinary `Eigen::Matrix3d` matrices and then assign them directly to a `SymmetricMatrix`.

3.2 Per-Element Stiffness Matrix Evaluation (15pts).

Implement `TopologyOptimizer::perElementStiffnessMatrix` to construct the 12×12 per-element stiffness matrix $K_e(1.0)$ for the *full-density* material. In other words, you should, *not* apply the SIMP interpolation rule here; instead, this scale factor will be applied when assembling K_e into the global stiffness matrix.

Some suggestions for better performance:

1) Take advantage of symmetry to compute only the upper triangle and copy the result into the lower triangle. This copy can be done with the line:

```
Ke.triangularView<Eigen::Lower>() = Ke.transpose();
```

2) Try to compute the stress quantity $C : \varepsilon(\overrightarrow{\phi}{}_a^e)$ only once per basis function.

3.3 Stiffness Matrix Assembly (15pts)

Assemble the *SIMP-interpolated* per-element stiffness matrices into the global sparse stiffness matrix K. Suggestion for better performance: take advantage of symmetry to omit the strict upper triangle. By default, both CHOLMOD and Eigen's built-in Cholesky factorization algorithms access only the lower triangle of the system matrix, so the strict upper triangle can be empty.

3.4 Equilibrium Solve (10pts)

Modify the stiffness matrix and right-hand side to implement the clamp boundary conditions. Note that in this assignment, clamp conditions can be specified on a per-component basis (clamping, e.g., only the y displacement component) as opposed to the per-node conditions specified in `hw2`. The relevant user input is available in `m_isSupportVar`.

3.5 Gradients of Compliance and Volume (15pts)

Implement the gradient of compliance and volume with respect to each tet's density value in `TopologyOptimizer::gradCompliance` and `TopologyOptimizer::gradVolume`. For the former, you should use the formulas in (5).

Once you have finished implementing your gradient formulas, you should be able to use the `tests` binary to validate them with finite differences.

3.6 Optimality-Criterion-Based Optimization (15pts)

Implement the optimality criterion method for solving the density-based topology optimization problem. Recall from `hw2` that the OC method is a heuristic fixed-point iteration based on the

condition satisfied by the optimal design:

$$\frac{\partial}{\partial \hat{\rho}}(\tilde{\mathbf{f}} \cdot \tilde{\mathbf{u}}) = -\lambda \frac{\partial \texttt{Vol}(\hat{\rho})}{\partial \hat{\rho}}.$$

It works by, given a Lagrange multiplier estimate λ and a "move limit" `m`, computing the updated design variables:

$$\hat{\rho}_i^{\text{new}}(\lambda) = \texttt{min}\left(\texttt{min}\left(\texttt{max}\left(\texttt{max}\left(\hat{\rho}_i\left(-\frac{\partial \tilde{\mathbf{f}} \cdot \tilde{\mathbf{u}}}{\partial \hat{\rho}_i} \Big/ \lambda \frac{\partial \texttt{Vol}(\hat{\rho})}{\partial \hat{\rho}_i}\right)^{\frac{1}{2}}, 0\right), \hat{\rho}_i - \texttt{m}\right), 1\right), \hat{\rho}_i + \texttt{m}\right). \quad (6)$$

Note that this operation just multiplies densities by 1 if the design is already optimal, leaving optimal designs unchanged (hence it truly is a fixed-point iteration). If a density variable is too low, the strain energy in the region influenced by this density variable (and thus the compliance derivative's magnitude) will be too large, and this formula will increase density. Likewise it will reduce density in under-strained regions.

With the updated densities, we can compute a corresponding design volume $\texttt{vol}(\lambda) = \texttt{Vol}(\hat{\rho}^{\text{new}}(\lambda))$. We obtain our Lagrange multiplier estimate by solving for the λ that ensures the volume constraint is exactly satisfied ($\texttt{vol}(\lambda) = \texttt{maxVolumeFrac * Vol}_0$). As in `hw2`, we cannot solve this equation in closed form due to the `min` and `max` clamping operations in (6). However, we note that the constraint violation $c(\lambda) := \texttt{maxVolumeFrac * Vol}_0 - \texttt{vol}(\lambda)$ is a monotonically increasing function of λ (increasing λ decreases densities, so `Vol` is decreasing). We can therefore find a λ satisfying $c(\lambda) \approx 0$ using a binary search.

Your tasks are to, for each of the `numSteps` iterations:

1) Solve the equilibrium problem for the current density variables $\hat{\rho}$.
2) Perform a binary search to find λ^* satisfying $|c(\lambda^*)| \leq \texttt{ctol}$:
   a) "Bracket" the solution by iteratively halving λ_{min} until $c(\lambda_{\text{min}}) < 0$ and doubling λ_{max} until $c(\lambda_{\text{max}}) > 0$.
   b) While $|c(0.5(\lambda_{\text{min}} + \lambda_{\text{max}}))| > \texttt{ctol}$
      i) If $(c(0.5(\lambda_{\text{min}} + \lambda_{\text{max}})) < 0)$, set $\lambda_{\text{min}} = 0.5(\lambda_{\text{min}} + \lambda_{\text{max}})$
      ii) If $(c(0.5(\lambda_{\text{min}} + \lambda_{\text{max}})) > 0)$, set $\lambda_{\text{max}} = 0.5(\lambda_{\text{min}} + \lambda_{\text{max}})$
3) Apply design improvement $\hat{\rho} = \hat{\rho}^{\text{new}}(0.5(\lambda_{\text{min}} + \lambda_{\text{max}}))$.

3.7 Smoothing Filter (5pts)

Complete the implementation in the constructor `VoxelSmoothingFilter::VoxelSmoothingFilter` in `DensityField.cc` to build the sparse matrix A such that $\rho_{\text{smoothed}} = A\rho_{\text{grid}}$ where $[\rho_{\text{smoothed}}]_i$ is the unweighted average of ρ_{grid} over the 3×3 cube neighborhood around grid cell i. Code has already been provided to access the indices of these neighbors.

3.8 Stress Analysis (5pts)

Implement `TopologyOptimizer::cauchyStress` in `TopologyOptimizer.cc` to compute the (linearized) Cauchy stress tensor $C : \varepsilon(\mathbf{u})$ for element `e`. Due to linearity of the displacement-to-stress-operator, this can be done by computing a linear combination of the element's (constant) vector-valued basis function stresses.

After you have completed this step, you should be able to visualize the *maximum principal stress* (the maximum-magnitude eigenvalue of the Cauchy stress tensor) by selecting `Stress` from the `Scalar Field` selection pop-up in the GUI.

4 Theory (5 pts)

Compare the compliance under a perturbation of the design load before and after running the topology optimization. For example, if you are optimizing the structure to withstand a force pointing along the $-y$ direction, try simulating with a small additional force component in the x or z directions. Do this both for your optimal design under the original load and for the uniform-area design of the same volume. Briefly report your findings in a `theory.pdf` file. How do they differ from the analogous experiment you performed in `hw2`?

5 Bonus Opportunities

5.1 Experiment with Other Optimizers (5 pts)

Install `nlopt` and experiment with some of the gradient-based optimizers it provides, both constrained and unconstrained. The following unconstrained formulation of (3) is provided by `SoftVolumeConstrainedProblem` in `TopologyOptimizer.hh`:

$$\min_{\hat{\rho}} \tilde{\mathbf{f}} \cdot \tilde{\mathbf{u}}(\hat{\rho}) + \frac{w}{2}(\texttt{Vol}(\hat{\rho}) - \texttt{maxVolumeFrac} * \texttt{Vol}_0)^2,$$

where w is a user-tunable penalty weight controlling the trade-off between meeting the volume budget and minimizing compliance. This formulation unfortunately behaves poorly with the included `LBFGS++` library, but should work reasonably with a better optimizer. I would recommend in particular experimenting with `nlopt`'s MMA optimizer (using the original constrained formulation (3)) and its implementation of the low-memory BFGS algorithm.

5.2 Additive Fabrication Filter (20pts).

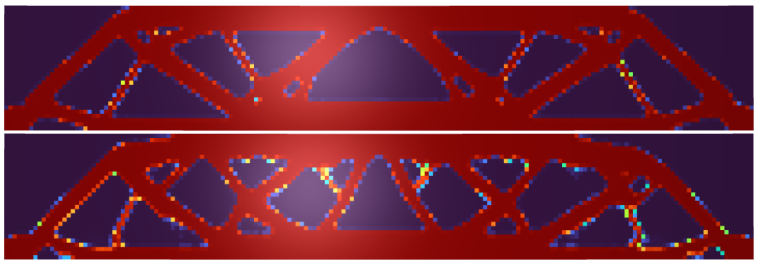


Figure 3: Example (mbb.obj, mbb_2d_hi.config) without (above) and with (below) the additive fabrication filter. The additive fabrication filter promotes steeper, more vertical features and avoids extended overhangs. Note that the filter makes it significantly more difficult to converge to a near-binary design (even the imperfect result shown here required a carefully scheduled increase of the β parameter across multiple runs of the OC algorithm). A superior optimization algorithm Section 5.1 would likely ease this process.

When 3D printing, especially on FDM or SLA machines, it is important all material deposited in a given layer to be robustly attached to material in the layer below. Designs satisfying this property are referred to as self-supporting. If a design is not self-supporting it must be augmented with a support structure, which adds additional cost, fabrication complexity, and postprocessing effort (to remove the support). Recently several approaches have been proposed to encourage self-supporting designs in density-based topology optimization. For this assignment, you will implement the particularly simple filter-based approach proposed in [Langelaar 2016].

The idea is illustrated in the figure below: we should only allow material to be deposited in a given voxel (cube cell shown in green) if material is present in the plus-shaped "support stencil" in the layer below.

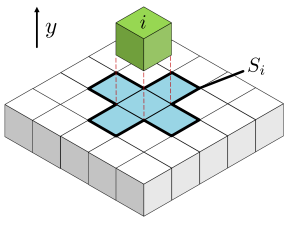


Figure 4: Stencil used in the self-supporting additive manufacturing density filter [Langelaar 2016].

The "relaxed" version of this rule that works also for intermediate densities is to limit the density of the green voxel not to exceed the maximum *filtered* density appearing in the support stencil. This operation could be expressed with max and min operators, but those are non-smooth and would pose

problems for optimization. Instead, we use smooth approximations to the min and max operators that are already provided for you in the code by methods of the `VoxelSelfSupportingFilter` class. The method `smin(a, b)` computes the smooth minimum of two numbers `a` and `b`, and the method `smax(x, I)` computes the smooth maximum of the values `x[i]` for `i` in the index set `I`.

Your tasks are to:

1) Implement `VoxelSelfSupportingFilter::getSupportingNeighbors` in `DensityField.cc` to construct the index set `supportingNeighbors` containing indices of the up-to-five voxels in the plus-shaped stencil in the layer below. Note that we assume the printing direction is along the vertical y axis (the second grid axis, indexed by `j` in the code), so "below" means in the $-y$ direction. Hint: look at how the `VoxelSmoothingFilter` constructor accesses neighboring voxels.

2) Implement `VoxelSmoothingFilter::apply` to, for each layer except the bottom, replace voxel densities ρ_i with the smooth-minimum of *input density* ρ_i and the smooth maximum of the *output (filtered) densities* within the support stencil below.

3) Implement `VoxelSmoothingFilter::backprop` to convert a gradient of some quantity `J` with respect to the filtered densities, `dJ_dout`, into a gradient with respect to the input densities, `dJ_din`. Notice that the "top layer" of `dJ_din` is straight-forward to compute directly with the chain rule (using derivatives of `smin` and `smax` that are provided for you in `dsmin_dx0`, `dsmin_dx1`, and `dsmax_dxk`). This is because changing the input density for a top-layer voxel influences the output density of only that same voxel. Unfortunately, changing an intermediate-layer voxel's input density will influence the output densities for all voxels in an expanding pyramid-shaped region above. Nevertheless, the desired gradient can be computed efficiently using the "backprop" technique you should be familiar with if you've implemented a neural network before. The trick here is to process the grid in the reverse order from `apply` (i.e., from top-to-bottom). When processing a given voxel `v` you should accumulate chain rule terms to entries `dJ_dout[s]` for voxels `s` in `v`'s support stencil that account for the change in `J` due to the change in `v`'s output density caused by changing `s`'s output density. Your implementation should guarantee that by the time `v` is processed, `dJ_dout[v]` holds the full sensitivity of `J` to `v`'s output density, accounting for indirect effects due to the neighbors it supports above. You cannot do this literally since `dJ_dout` is read-only, but you can either maintain a copy "`dJ_dout_copy`" or, to minimize memory usage, temporarily store these values for unprocessed voxels in `dJ_din`. Note that your chain rule formulas will need to use both `m_input_x` and `m_filtered_x`, the current input and output density arrays saved by the last call to `apply`.

You should use the `tests` binary to validate your gradient implementation.

6 What to Turn In

Please submit a `.zip` or `.tgz` archive containing all the source files you edited (probably this is just `TopologyOptimizer.cc` and `DensityField.cc`) along with your `theory.pdf`.

References

Langelaar, M. 2016. Topology optimization of 3D self-supporting structures for additive manufacturing. *Additive Manufacturing 12*, 60–70.