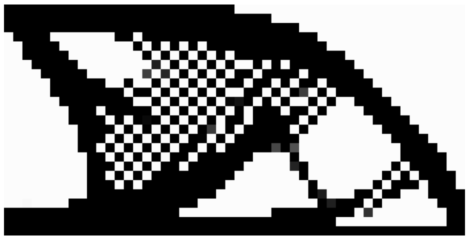
6.8420: Computational Design and Fabrication

Charlotte Folinus
Massachusetts Institute of Technology
Spring 2024

Homework 4: Topology Optimization

Section 2: Topology Optimization

Problem 2.1.1: Implement Optimality Criteria in the `optimality_criteria` function in `part1/topopt.py`. Provide the screenshot of your results after this step.


(a) Image of optimized structure

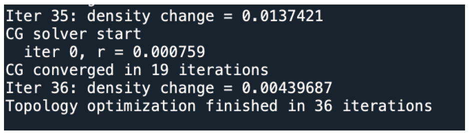
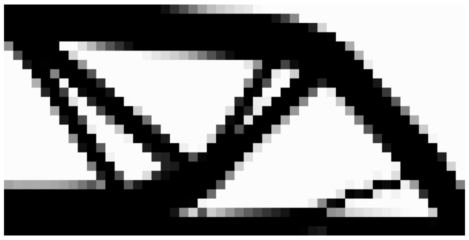

```
Iter 35: density change = 0.0137421
CG solver start
  iter 0, r = 0.000759
CG converged in 19 iterations
Iter 36: density change = 0.00439687
Topology optimization finished in 36 iterations
```
(b) Console/solver output

Figure 1: Results after implementing optimality criteria, shown for the MBB scenario

Problem 2.1.2: Implement Sensitivity Filtering in the `sensitivity_filtering` function in `part1/topopt.py`. Provide the screenshot of your results after this step.


(a) Image of optimized structure

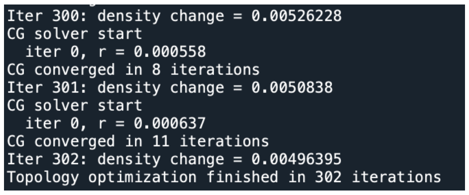
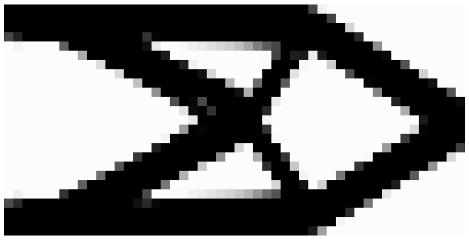
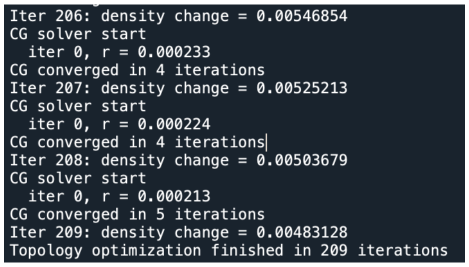

```
Iter 300: density change = 0.00526228
CG solver start
  iter 0, r = 0.000558
CG converged in 8 iterations
Iter 301: density change = 0.0050838
CG solver start
  iter 0, r = 0.000637
CG converged in 11 iterations
Iter 302: density change = 0.00496395
Topology optimization finished in 302 iterations
```
(b) Console/solver output

Figure 2: Results after implementing sensitivity filtering, shown for the MBB scenario

Problem 2.1.3: Implement the boundary conditions for cantilever beam design and bridge design in the `initialize_boundary_condition` function in `part1/topopt.py`. Provide the screenshot of the results of these two models.



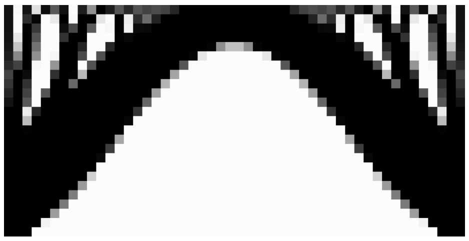
(a) Image of optimized structure



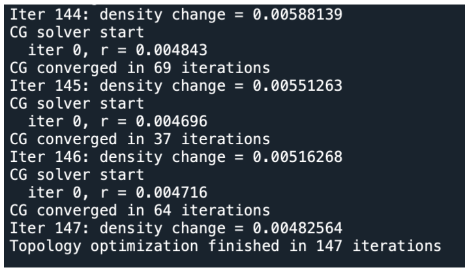
```
Iter 206: density change = 0.00546854
CG solver start
  iter 0, r = 0.000233
CG converged in 4 iterations
Iter 207: density change = 0.00525213
CG solver start
  iter 0, r = 0.000224
CG converged in 4 iterations
Iter 208: density change = 0.00503679
CG solver start
  iter 0, r = 0.000213
CG converged in 5 iterations
Iter 209: density change = 0.00483128
Topology optimization finished in 209 iterations
```

(b) Console/solver output

Figure 3: Results after implementing boundary conditions, shown for the cantilever beam with the default settings



(a) Image of optimized structure



```
Iter 144: density change = 0.00588139
CG solver start
  iter 0, r = 0.004843
CG converged in 69 iterations
Iter 145: density change = 0.00551263
CG solver start
  iter 0, r = 0.004696
CG converged in 37 iterations
Iter 146: density change = 0.00516268
CG solver start
  iter 0, r = 0.004716
CG converged in 64 iterations
Iter 147: density change = 0.00482564
Topology optimization finished in 147 iterations
```

(b) Console/solver output

Figure 4: Results after implementing boundary conditions, shown for the cantilever beam with the default settings

Section 3: Design and Performance Space Exploration

Problem 3.2.1: Complete an O(n log n) algorithm for finding the Pareto front of 2D points in the `pareto_front` function in `compfab-hw4-24s-skeleton/part2/pareto.py`.
This section did not ask any specific questions.

Problem 3.2.2: Integrate your solutions in Assignment 2 and Assignment 4 to complete the mapping from design space to performance space for bridge models and compute the Pareto front for them. You need to copy and paste relevant coding sections into `fem.py`, `material.py`, and `voxelizer.py` in the Part 2 folder, then fill the code in `main.py`. Please plot your performance space and the Pareto front, and attach the image in your report.

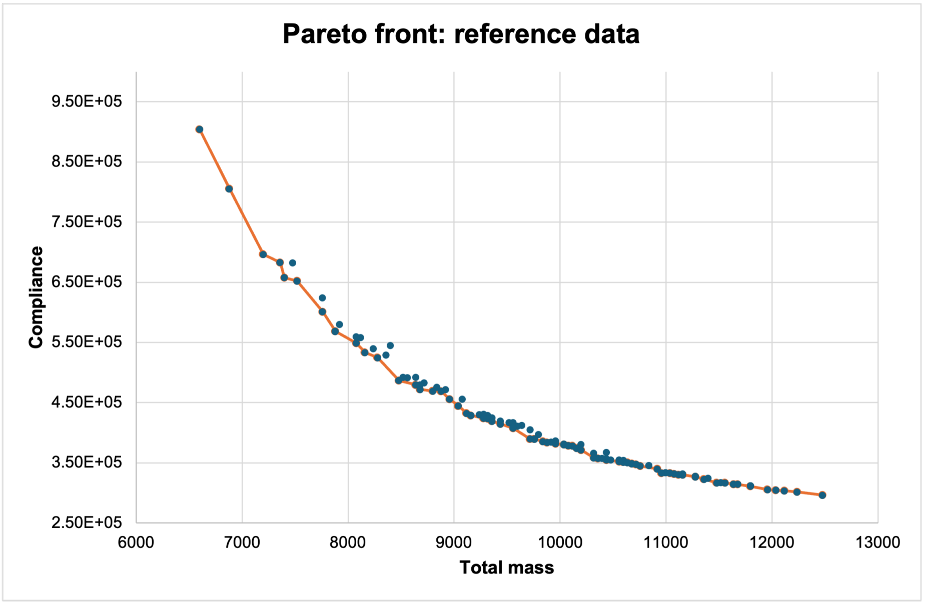


Figure 5: Pareto front of the arch bridges (orange line). All bridges shown as blue dots

Section 4: Writeup

What did you implement in order to complete the assignment?

- **Section 2:** This section asked us to implement a few specific components:
  - Optimality criteria: This is an algorithm used for topology optimization (I've previously mostly implemented/worked with MMA, so it was cool to be exposed to something else!). Implementing the algorithm involved determining how much to change the element densities, which is in part based on the lagrangian multiplier λ and the change limit M. The pseudocode provided to us was pretty straightforward, but the trickiest part was understanding how to implement the min/max formulation using the two calls to `np.clip()` (i.e., going from Eq. 1 in the handout to the two lines of code). Before clipping to the valid density range (0.01-1.0), we first need to clip the "new" densities to the range defined by the change limit. This range is defined relative to the original densities, not relative to the "new" values: `d_new = np.clip(d_new, d - change_limit, d + change_limit)`.
  - Sensitivity filtering: This section asked us to implement a distance-based filter to get rid of checkerboarding-based artifacts. This filter works by "blurring" across voxels/pixels, which works as a 2D convolution. For this function, I needed to compute the convolutional kernel (w) and understand what "image" to apply the convolution to; the input image for the numerator term is the densities multiplied by the sensitivities, but the input image for the denominator term is just the densities.
- **Section 3:** This section asked us to implement a few specific components:
  - Pareto front: I started by sorting all of the performance data by one of the objective values (I chose the first objective, total mass) and adding the first point to my list of indices corresponding to pareto-optimal designs. I then looped through the remaining designs in order of the first objective value. If a design's y-value (its second objective value) was better (lower) than that of the most-recently-added design on the pareto front, it was nondominated, which meant that I added it to my Pareto front and updated the record of the most-recently-added pareto-optimal design.
  - Integrating solutions from other assignments: We hadn't previously implemented `voxelizer.py` in other assignments, and I implemented the `run_accelerated()` portion.. This function uses ray casting to shoot a bunch of vertically-oriented (so, parallel) rays from a lower surface through our part (via `parallel_ray_may_mesh_intersection`) and determine the indices where each ray intersects the part geometry. Most rays will either not intersect the part (a blank intersection) or intersect at two indices (corresponding to the bottom and top of the part at that XY location). A ray could also exactly intersect an individual node, which would produce only

one intersection. After determining the intersection locations, I looped over each XY location; if a ray intersected this XY location, I “filled” the voxels between the upper and lower intersection location.

What did you like? I liked that this assignment provided reference images/data for comparing our results at each stage of the implementation – this really helped with troubleshooting and making sure that one function was correct before proceeding to the next one.

What did you not like? I noticed that the data in Figure 6 of the assignment packet did not correspond to the reference csv data that was shared with us. My data seemed to match the reference csv data, but it was a little confusing when it did not match the data from the assignment handout (particularly because the csv data was not initially shared with us). The images in section 2 also seem to have been generated with the default settings, but this wasn’t explicitly stated. This wasn’t a pain point for me, but I know other students who were really stuck on why they couldn’t replicate the provided images!

Any other issues Nope, all good!