Self-Supporting Filter in HW3

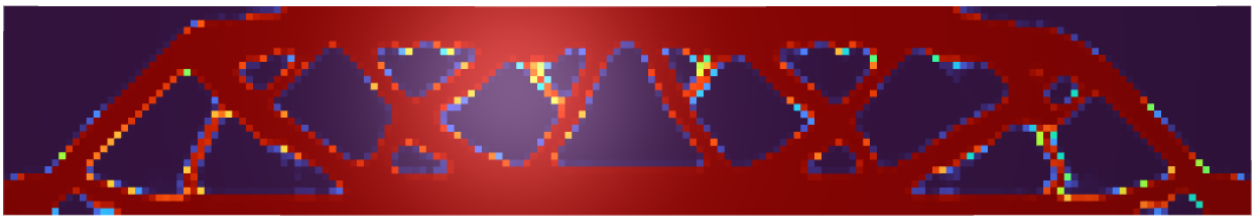

Figure 1: Result of the self-supporting additive manufacturing filter on the MBB beam example.

This document provides additional implementation details for the additive manufacturing filter proposed in [Langelaar 2016]. The central idea of this filter is to ensure all material deposited in a layer is securely attached to the layer below by forbidding the density in a cell i to exceed the maximum density appearing within its "support stencil" S_i.

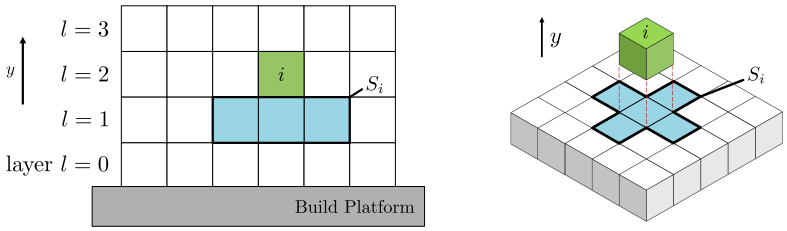

Figure 2: Plus-shaped support stencil S_i for cell i.

Like any filter used in density-based topology optimization, this AM filter is a change of variables from input "blueprint" density variables $\mathbf{x}$ to filtered output densities $\tilde{\mathbf{x}}$. To plug this filter into the "filter pipeline" implemented by the `DensityField` class, it must provide the `apply` method (mapping $\mathbf{x}$ to $\tilde{\mathbf{x}}$) and the `backprop` method (mapping gradients with respect to $\tilde{\mathbf{x}}$ to gradients with respect to $\mathbf{x}$).

Using the smoothed (differentiable) minimum and maximum functions `smin(a, b)` and `smax(vec)`[1] provided for you, we can express the output densities for all layers $l > 0$ above the bottom with the following pseudocode:

$$\tilde{x}_i = \texttt{smin}(x_i, \texttt{smax}([\tilde{x}_j \text{ for } j \in S_i])). \tag{1}$$

Bottom-layer densities are not modified as they are supported by the build platform. We emphasize that $\tilde{x}_i$ depends not only on input density x_i but also the *filtered* density values within its stencil.

[1] This notation differs slightly from the `smax` function actually provided to you, which accepts two arguments: a vector of all density variables `x` and an index vector `I` specifying a subset of `x`; you will actually call $\texttt{smax}(\tilde{\mathbf{x}}, S_i)$.

1 Implementing `apply`

Applying the recurrence relation (1) in order to map $\mathbf{x}$ to $\tilde{\mathbf{x}}$ is easy and efficient as long as you sweep upward starting from $l = 1$. This way, when processing cell i in layer l, the filtered densities needed for the cells in S_i from layer $l-1$ are already available in `m_filtered_x`.

2 Implementing `backprop`

The purpose of `backprop` is to compute the gradient of some function $J(\mathbf{x}) = \tilde{J}(\tilde{\mathbf{x}}(\mathbf{x}))$ given $\frac{\partial \tilde{J}}{\partial \tilde{\mathbf{x}}}$, the derivative with respect to the filtered variables. This might appear to be an easy application of the chain rule, $\frac{\partial J}{\partial \mathbf{x}} = \frac{\partial \tilde{J}}{\partial \tilde{\mathbf{x}}} \frac{\partial \tilde{\mathbf{x}}}{\partial \mathbf{x}}$, however the recursive definition of $\tilde{\mathbf{x}}$ in (1) makes the Jacobian $\frac{\partial \tilde{x}_i}{\partial x_k}$ a little tricky to reason about. With some thought, we observe $\frac{\partial \tilde{x}_i}{\partial x_k}$ is *lower triangular* (assuming cell indices are sorted from bottom to top) since a given filtered variable depends only on the corresponding input variable and the variables in an expanding pyramid-shaped domain *below*. However, it is highly dense since output densities at the top layer are influenced by a large fraction of the input variables. This means it will be inefficient to construct and multiply by $\frac{\partial \tilde{x}_i}{\partial x_k}$ directly.

We can nevertheless compute $\frac{\partial \tilde{J}}{\partial \tilde{\mathbf{x}}} \frac{\partial \tilde{\mathbf{x}}}{\partial \mathbf{x}}$ efficiently by exploiting the recursive structure of (1). We begin by disentangling x_k from the many $\tilde{x}_i$ it influences *indirectly* through $\tilde{x}_k$ by organizing the computation of gradient component $\frac{\partial J}{\partial x_k}$ as follows:

$$\frac{\partial J}{\partial x_k} = \sum_{i \geq k} \frac{\partial \tilde{J}}{\partial \tilde{x}_i} \frac{\partial \tilde{x}_i}{\partial x_k} = \left(\sum_{i \geq k} \frac{\partial \tilde{J}}{\partial \tilde{x}_i} \frac{\partial \tilde{x}_i}{\partial \tilde{x}_k} \right) \frac{\partial \tilde{x}_k}{\partial x_k} = \boxed{\frac{\mathrm{d}\tilde{J}}{\mathrm{d}\tilde{x}_k} \frac{\partial \tilde{x}_k}{\partial x_k}}, \tag{2}$$

where we have introduced the "total derivative"

$$\frac{\mathrm{d}\tilde{J}}{\mathrm{d}\tilde{x}_k} := \sum_{i \geq k} \frac{\partial \tilde{J}}{\partial \tilde{x}_i} \frac{\partial \tilde{x}_i}{\partial \tilde{x}_k}$$

that measures the full change in J due to changing $\tilde{x}_k$, including not only the direct effect $\frac{\partial \tilde{J}}{\partial \tilde{x}_k}$, but the indirect effect through changing the density variables $\tilde{x}_s$ it supports in the layers above. The key observation is that we can formulate a recurrence equation for efficiently computing this total derivative by decomposing $\frac{\partial \tilde{x}_i}{\partial \tilde{x}_k}$, the effect of $\tilde{x}_k$ on $\tilde{x}_i$, in terms of $\frac{\partial \tilde{x}_s}{\partial \tilde{x}_k}$, the effects of $\tilde{x}_k$ on densities of the voxels s that it *directly* supports:

$$\frac{\mathrm{d}\tilde{J}}{\mathrm{d}\tilde{x}_k} = \frac{\partial \tilde{J}}{\partial \tilde{x}_k} + \sum_{s \in S_k^\top} \sum_{i \geq s} \frac{\partial \tilde{J}}{\partial \tilde{x}_i} \left(\frac{\partial \tilde{x}_i}{\partial \tilde{x}_s} \frac{\partial \tilde{x}_s}{\partial \tilde{x}_k} \right) = \frac{\partial \tilde{J}}{\partial \tilde{x}_k} + \sum_{s \in S_k^\top} \left(\underbrace{\sum_{i \geq s} \frac{\partial \tilde{J}}{\partial \tilde{x}_i} \frac{\partial \tilde{x}_i}{\partial \tilde{x}_s}}_{\frac{\mathrm{d}\tilde{J}}{\mathrm{d}\tilde{x}_s}} \right) \frac{\partial \tilde{x}_s}{\partial \tilde{x}_k}$$

$$= \boxed{\frac{\partial \tilde{J}}{\partial \tilde{x}_k} + \sum_{s \in S_k^\top} \frac{\mathrm{d}\tilde{J}}{\mathrm{d}\tilde{x}_s} \frac{\partial \tilde{x}_s}{\partial \tilde{x}_k},} \tag{3}$$

where $S_k^\top := \{s | k \in S_s\}$ is the set of indices of voxels that voxel k directly supports. The transpose notation is natural because if we encode the support stencils S_i for all voxels as rows of a large sparse binary matrix S (with a 1 in each column $j \in S_i$ and 0 elsewhere), then index set $S_k^\top$ is really encoded as the k^{th} row of $S^\top$.

This matrix S can be interpreted as the adjacency matrix of the (directed) graph describing the data dependencies in our computation. Transposing S has the effect of reversing the direction of data flow so that, while density information propagates from the bottom layer up to the top, derivative information propagates backwards from the top layer down to the bottom (hence the "backprop" terminology): all entries $s \in S_k^\top$ satisfy $s > k$. Accordingly, our `backprop` method will sweep downward from the top layer $l =$ `numLayers` $- 1$ to $l = 0$. By processing cells in this order, the necessary entries $\frac{\mathrm{d}\tilde{J}}{\mathrm{d}\tilde{x}_s}$ will already be available when computing $\frac{\mathrm{d}\tilde{J}}{\mathrm{d}\tilde{x}_k}$.

We make one last observation to simplify the implementation and allow us to avoid literally constructing transposed stencils (and just reuse `VoxelSelfSupportingFilter::getSupportingNeighbors`). We declare an array `dJ_dxfilt` to hold the total derivatives $\frac{\mathrm{d}\tilde{J}}{\mathrm{d}\tilde{x}_k}$ and initialize it to `dJ_dout` (*i.e.*, $\frac{\partial \tilde{J}}{\partial \tilde{x}_k}$). Instead of having our processing of voxel k loop over voxels $s \in S_k^\top$ to compute the sum in (3), we can make it accumulate contributions $\frac{\mathrm{d}\tilde{J}}{\mathrm{d}\tilde{x}_k} \frac{\partial \tilde{x}_k}{\partial \tilde{x}_j}$ to `dJ_dxfilt[`j`]` for each $j \in S_k$. This way, by the time voxel k is processed, `dJ_dxfilt[`k`]` already holds the complete total derivative. The added benefit of this approach is that we only differentiate a single $\tilde{x}_k$ at a time (rather than the multiple $\tilde{x}_s$ appearing in (3)), which results in simpler code and helps us avoid recomputation of certain chain rule terms. We finally compute entries of the result array `dJ_din[`k`]` by multiplying `dJ_dxfilt[`k`]` by $\frac{\partial \tilde{x}_k}{\partial x_k}$ as indicated in (2).

You can either allocate a separate array `dJ_dxfilt` as described, or use the result array `dJ_din` to temporarily hold `dJ_dxfilt[`k`]` in `dJ_din[`k`]` until the multiplication by $\frac{\partial \tilde{x}_k}{\partial x_k}$ has been performed.

References

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