APMA2560 FINAL PROJECT: TOPOLOGY OPTIMIZATION

MATTHEW MEEKER

1. Introduction

Briefly, topology optimization offers a class of techniques and algorithms for discovering the optimal distribution of material within a given domain, subject to some set of physics and other constraints, which tend to form PDE-constrained optimization problems. In general, these problems take the form

$$\begin{aligned} \min_{\rho} \quad & F = F(u(\rho), \rho) = \int_{\Omega} f(u(\rho), \rho) \mathrm{d}V \\ \text{s.t.} \quad & \int_{\Omega} \rho \mathrm{d}V - V_0 \leq 0, \\ & G_i(u(\rho), \rho) \leq 0, \end{aligned} \tag{1}$$

where Ω is the design domain, $\rho \in L^2(\Omega)$ is a function describing the material distribution, f forms the objective function, θ is the *mass fraction* (that is, the fraction of Ω which may be occupied by material; this may be made physical by considering material cost constraints, etc.), and G_j describes a related set of constraints.

Contents

2. Classical Methods

To this end, Equation 1 is not quite complete: we will require also that ρ take on binary values 0 or 1 within Ω, with the natural interpretation being that this indicates whether or not that point should be empty or solid material. Typically, one will use a finite element method to assess the state and develop some notion of a gradient to inform the design steps.

However, we run immediately into a few issues:

(1) *Existence/ Uniqueness*: In general, topology optimization problems may not have solutions, much less unique solutions.

(2) *Mesh dependence*: Often, introducing more "holes" to the design will decrease the objective. If the discretized Ω has a very large number of elements, we make way for the introduction of more and more holes, meaning that an extremely fine mesh could yield a substantially different solution than that which might be reached on a coarser mesh.

(3) *Discrete ρ*: Such a discrete optimization problem is extremely hard to solve and would be certainly very costly for any reasonably fine mesh. Algorithms depending on continuity are, in general, better understood, computationally efficient, etc, so we can instead require $0 \leq \rho \leq 1$ throughout the domain.

(4) *Gray areas with continuous ρ*: Using continuous ρ makes for a tractable problem, however it is itself not without loss.

    In particular, we lose the clear physical interpretation allowing for any value between 0 and 1 at each location. Moreover, we need to penalize cases where ρ takes on values at least not very close to 0 or 1, otherwise we may have significant portions of "gray area," which are un-physical and uninterpretable.

This full topology optimization problem with continuous design variables can be written

$$\begin{aligned} \min_{\rho} \quad & F = F(u(\rho), \rho) = \int_{\Omega} f(u(\rho), \rho) \mathrm{d}V \\ \text{s.t.} \quad & \int_{\Omega} \rho \mathrm{d}V - V_0 \leq 0, \\ & G_i(u(\rho), \rho) \leq 0, \\ & 0 \leq \rho \leq 1. \end{aligned} \tag{2}$$

Several approaches for continuous problems exist; the largest of these are the density, topological derivative, and level set approaches. The density approach will be adopted to run several simulations herein, so we will be most interested in this. Unless stated otherwise, we are interested only in the continuous problems here.

2.1. **Density Approach.** By and large the most prevalent method is the density approach. Here, one chooses a way to introduce a notion of material property that, given the presence or absence of material at that location, will impact the physics introduced; the geometry of the structure is then built by describing where the material should be placed. The objective in Equation 2 can often be written as some $r(\rho) \cdot f_0(u)$, where $r(\rho)$ is the choice density interpolation function and f_0 is a function of the field for solid material.[BS03]

Turning to addressing gray areas, we can explicitly penalize it (incentivize $0-1$ solutions) with an additional term

$$\alpha \int_\Omega \rho(1-\rho). \tag{3}$$

While initially taken to be artificial and later given physical meaning,[SM13] the Simplified Isotropic Material with Penalization (SIMP) approach is the foremost density approach for single-variable material problems[1] and is the particular methodology adopted for the later simulations; it serves as an implicit penalization against the explicit penalization strategy above.

The idea in general is to provide a continuous interpolation between solid and void while penalizing the intermediate density values (thereby addressing the issue raised in item (3) above) with the power-law, ie,

$$E(\rho_i) = \rho_i^p \cdot E_0,$$

where ρ_i denotes the value at a location i, p is a chosen exponent (usually chosen $p = 3$), and E_0 is the Young's modulus for solid material. For $p = 1$, it is worth noting that there is a unique solution for the compliance objective due to the problem's convexity; for $p > 1$, we will begin to favor $0 - 1$ solutions.

To address issue (2) from above, restriction methods have been introduced and provide a way to ensure mesh-independency and well-posedness. Heuristically, we should never allow for the "infinitesimal holes" to be a part of the solution; in the discretization, this is akin to allowing structural features of the solution to be captured with maybe a single low-order element. It is with this that we can recover our expectations of mesh convergence and better serve physical intution.

2.1.1. *Sensitivity Filtering/ One-field SIMP.* The first method is the sensitivity filter, which allows the introduction of a notion of length-scale. Specifically, the filter modifies element sensitivty values as weighted averages within a (mesh-independent) radius r_{min}. Because of this, there will consistently be a "gray edge" separating areas of solid material and of void, though this remains much more physically intuitive than without.

Other one-field SIMP methods come in the form of explicit constraints/ penalties on the gradient of ρ or its perimeter, which can be expressed in q-norms $||\cdot||_q$ integrated over

[1] as opposed to having several materials.

the domain Ω. It's easy to see a natural extension of the explicit penalization Equation 3 of the form

$$\alpha \int_{\Omega} \overline{\rho}(1-\overline{\rho}), \tag{4}$$

where $\overline{\rho}$ is some localized density average over an $r_{\min}$. However, it is difficult to select parameter values a priori.

2.1.2. *Density Filtering/ Two-Field SIMP.* The main alternative to sensitivity filtration is density filtration, where we work with e.g. again density averages $\overline{\rho}$ over a length parameter r which can be related to $r_{\min}$ from above. This can be written as a Helmholtz type diffusion

$$-r^2\Delta\overline{\rho} + \overline{\rho} = \rho, \tag{5}$$

though again we will have the issue of grey transition areas. The term 'two-field' comes from this filter working with both the design variable field ρ and the physical density field $\overline{\rho}$. If well penalized, the solution will end up being very nearly discrete.

2.2. **Other Major Algorithms.** While not a primary focus, it is worth briefly describing the topological derivative and level set approaches.

The method of topological derivatives in topology and shape optimization's central idea is to assess the influence of introducing infinitesimal holes at points in the domain Ω and use that to inform the design update steps. However, the mathematics required to derive this method is rather complicated; more importantly, the holes introduced in a finite element discretization will always be of finite volume, rather than the infinitesimal holes given on paper. The rammifications thereof are not well understood.

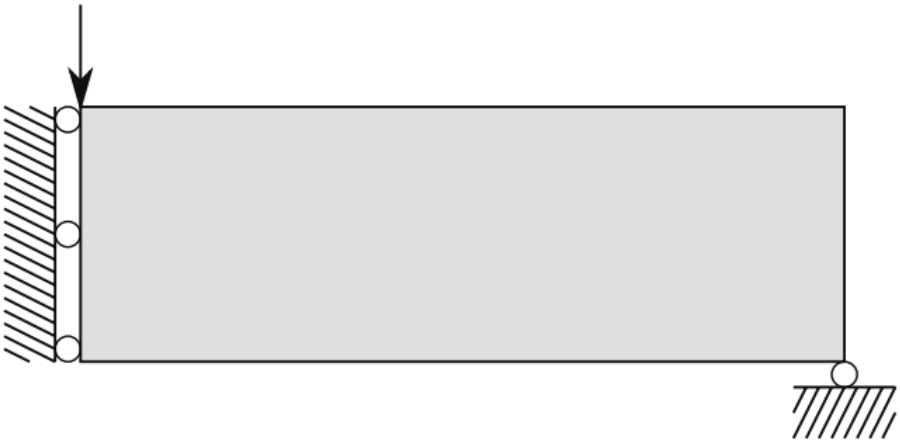
The level set approach's central idea is to construct the geometry via the definition of a solid-void interface, rather than explicitly indicating where the material should be placed as is done with the density approach. Specifically, the design boundary will be specified by the zero level counter of the level set function $\phi(x)$ with the structure itself defined by the domain where the level set function is positive.

$$\rho = \begin{cases} 0 : \forall x \in \Omega : \phi < 0 \\ 1 : \forall x \in \Omega : \phi \geq 0. \end{cases} \tag{6}$$

2.3. **Discrete Approaches: ESO.** Since Equation 1 was posed for discrete ρ, it is also worth mentioning the evolutionary methods that have seen success on smaller problems, though these are also not a primary focus.

The most notable family is the evolutionary approach, which is based off a simple hard-kill strategy where one removes the lowest energy element from the structure (ESO). This can be extended to a bidirectional strategy when elements are added only if they are considered rewarding in the analogous way (BESO). More recently came the genetic evolutionary structural optimization strategy (GESO), which combined the genetic algorithm with ESO.

FIGURE 1. Ω for `top88.jl` with the applied boundary conditions and loads.



3. STRUCTURAL COMPLIANCE WITH `top88`

As a first example problem, we will consider structural compliance and the method given in [And+10]. A Julia implementation of the code may be found in the appendices.

The MBB beam is a classic problem from mechanical engineering for topology optimization. Within Ω, subject to natural constraints on cost and area, we would like to build a maximally stiff structure. Intuition would serve a solution with stacked triangles, as will be observed.

3.1. Problem Specification.

The domain Ω is discretized with (square) quadrilateral elements and the density approach is adopted, so, to each element e in the domain, we will associate a material density x_e that is used to determine its Young's modulus E_e by

$$E_e(x_e) = E_{\text{min}} + x_e^p(E_0 - E_{\text{min}}). \tag{7}$$

As mentioned, p is typically taken to be 3 and E_{min} is a relatively very small value that is assigned to all elements for primarily numerical reasons – in particular, we avoid running into a singualr stiffness matrix. [And+10] titles this a "modified SIMP approach." Following the format of Equation 2, the problem may be written

$$\begin{aligned} \min_{\mathbf{x}} \quad & c(\mathbf{x}) := U^T K U = \sum E_e(x_e) u_e^T k_0 u_e \\ \text{s.t.} \quad & V(x)/V_0 = \theta, \\ & KU = F, \\ & 0 \le \rho \le 1. \end{aligned} \tag{8}$$

Here, c defines the compliance, U and F denote resp. the global displacement and force vectors, K is the global stiffness matrix, u_e is the element displacement vector, k_0 is the element stiffness matrix (for a unit Young's modulus), and $\mathbf{x}$ denotes the element densities/ material distribution. Lastly, $V(x)$ denotes the volume of x, V_0 of the entire domain Ω, and θ is a prescribed volume fraction (ie, the fraction of the design domain the structure is allowed to take up).

3.1.1. *Optimality Criteria Method.* The optimization problem is solved with a standard optimality criteria method.

$$x_e^{t+1} = \begin{cases} \max(0, x_e - m) & \text{if } x_e B_e^\eta \leq \max(0, x_e - m) \\ \min(1, x_e + m) & \text{if } x_e B_e^\eta \geq \min(1, x_e + m) \\ x_e B_e^\eta & \text{otherwise.} \end{cases} \tag{9}$$

m is a positive move limit, $\eta (= 1/2)$ is a numerical damping coefficient, and B_e is obtained from the optimality condition as

$$B_e = \frac{-\frac{\partial c}{\partial x_e}}{\lambda \frac{\partial V}{\partial x_e}}, \tag{10}$$

where the Lagrangian multiplier λ must be chosen such that the volume constraint is satisfied, which can be found with a bisection algorithm.

The partials of the objective function c and material volume V with respect to x_e are

$$\frac{\partial c}{\partial x_e} = -p x_e^{p-1} (E_0 - E_{\text{min}}) u_e^T k_0 u, \tag{11}$$

and

$$\frac{\partial V}{\partial x_e} = 1, \tag{12}$$

which comes from the assumption that each element has unit volume (so, observe that a decrease in density at one element will lead to an equal increase in density at another element such that the volume of the structure overall has not shifted).

3.1.2. *Filtering.* The code implements both a density and a sensitivity filter, and the filtered densities are output.

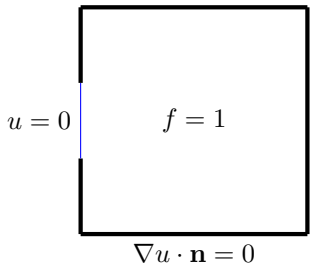
3.2. **Result.** Confirming roughly our intuition, we see a number of triangles forming. The following was run on a rectangle of 150x50 elements with a volume fraction of 0.5, $r_{\text{min}}/V(e) = 2.0$ (ie, radius divided by element volume), and using a density filter.

FIGURE 2. `top88` solution on 150x50 rectangle.

4. Thermal Compliance with toph

Using an identical algorithm, we will now solve a thermal compliance problem that will be revisited with EFEM in subsubsection 5.4.1; the algorithm, implementation, and derivation are worked out in greater detail in Appendix A.

Figure 3. Ω for toph.



That is, Ω is a square with a small component of the left side cut out allowing heat to flow through; the interior of the domain is evenly heated. We again discretize with square elements.

4.1. **Results.** First with a 40x40 square discretization, mass fraction $\theta = 0.4$, SIMP penalization exponent of 3.0, and $r_{\min}$ of 1.2, and secondly with an 80x80 square.

Keeping the constant even heating in mind, the solution reached is fairly natural: the goal is essentially to a build a heatsink that draws heat from Ω out to the left. So, the tree structure formed first serves this purpose as a conductor and secondly maximizes the surface area (which can be seen as the mesh is refined).

FIGURE 4. 40x40 `toph` simulation.

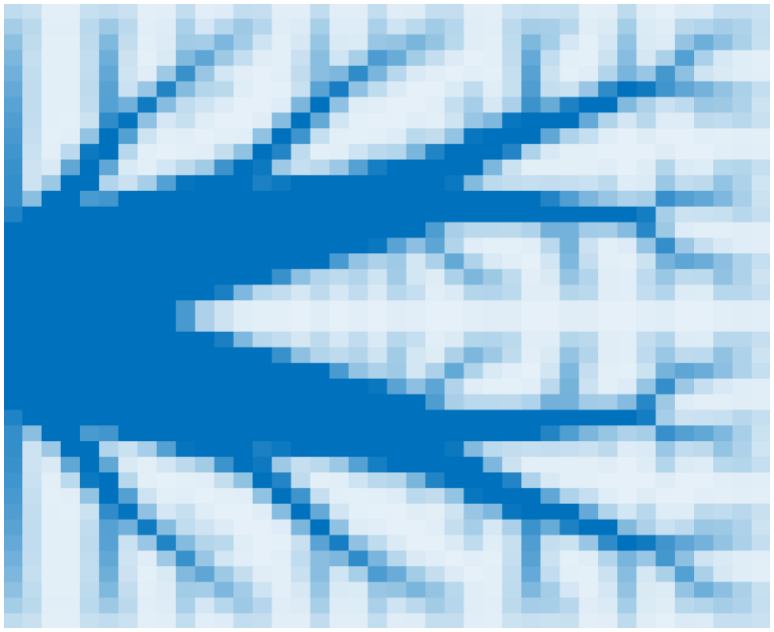
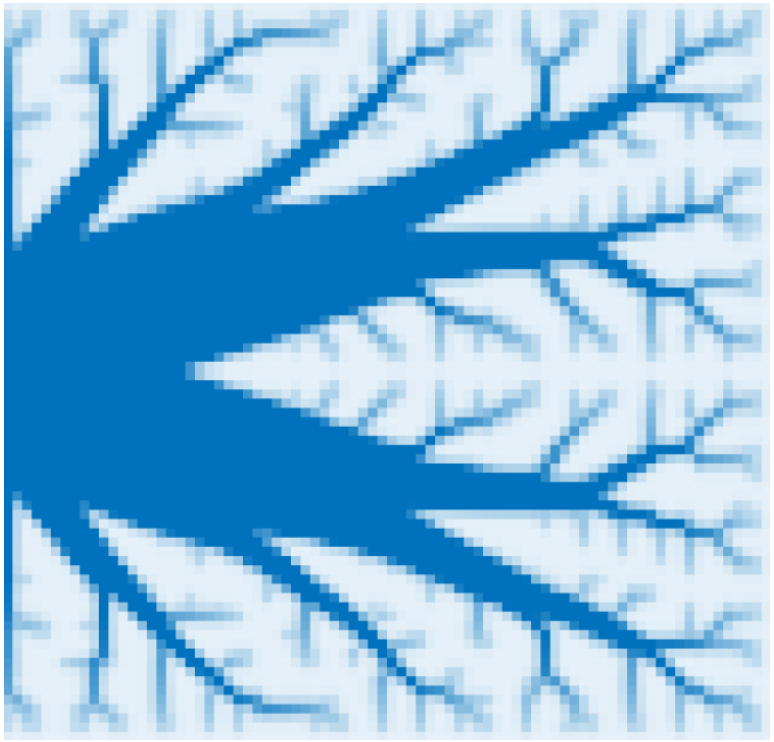


FIGURE 5. 80x80 `toph` simulation.



5. Thermal Compliance with the Entropic Finite Element Method (EFEM)

We would like to solve the thermal compliance problem

$$\min_{\rho} \quad \int_\Omega f \cdot u \, \mathrm{d}x$$
$$\text{s.t.}$$
$$\begin{cases} -\nabla \cdot (r(\tilde{\rho})\nabla u) = f & \text{in } \Omega \text{ and BCs}, \\ -\epsilon^2 \Delta \tilde{\rho} + \tilde{\rho} = \rho & \text{in } \Omega \text{ and Neumann BCs}, \\ 0 \le \rho \le 1 & \text{in } \Omega, \\ \int_\Omega \rho \, \mathrm{d}x = \theta \cdot \mathrm{vol}(\Omega), \end{cases} \tag{13}$$

over $u \in [H^1(\Omega)]^2$ and $\rho \in L^2(\Omega)$. Here, $r(\tilde{\rho}) = \rho_0 + \tilde{\rho}^3(1-\rho_0)$ is the SIMP law, ϵ is the design length scale, and $0 < \theta < 1$ is the volume fraction. Recall the second constraint as the density filter for length parameter ϵ.

5.1. Method.

Breaking from the methodology in `toph.jl`, we will now use Keith and Surowiec's Entropic Mirror Descent algorithm (2023) tailored to the constraint $\rho \in [0,1]$. Below, we will let σ denote the sigmoid function, σ^{-1} the inverse sigmoid function, clip(y) $:=$ min(max_val, max(min_val, y)), and projit(ψ) is a (compatible) projection operator yet to be defined.

Algorithm 1 Entropic Mirror Descent for PDE-constrained Topology Optimization

Require: $\mathcal{L}(u, \rho, \tilde{\rho}, w, \tilde{w})$ (the Lagrangian), $\alpha > 0$ (the step size)
Ensure: $\rho = \arg\min \int_\Omega f \cdot u \, \mathrm{d}x$
  $\psi \leftarrow \sigma^{-1}(\theta)$ ▷ So that $\int \sigma(\psi) = \theta \cdot \mathrm{vol}(\Omega)$
  while not converged **do**
    $\tilde{\rho} \leftarrow$ solution to $\partial_{\tilde{w}} \mathcal{L} = 0$, ▷ Filter solve
    $u \leftarrow$ state solution; solution to $\partial_w \mathcal{L} = 0$, ▷ Primal problem
    $\tilde{w} \leftarrow$ filtered gradient; solution to $\partial_{\tilde{\rho}} \mathcal{L} = 0$,
    $G \leftarrow M^{-1}\tilde{w}$; ie, is the solution to $(G, v) = (\tilde{w}, v)$ for all $v \in L^2$,
    $\psi \leftarrow$ clip(projit($\psi - \alpha G$)). ▷ Mirror descent update
  end while

for the following discretization choices $u \in V \subset [H^1]^d$, $\psi \in L^2$ (with $\rho = \sigma(\psi)$), $\tilde{\rho} \in H^1$, $w \in V$, $\tilde{w} \in H^1$, where

Algorithm 2 projit(ψ) Nonlinear Projection

Require: $\psi \in L^2(\Omega)$, $0 < \theta < 1$
Ensure: $\int_\Omega \sigma(\psi + c) \, \mathrm{d}x = \theta \cdot \mathrm{vol}(\Omega)$
    $c \leftarrow$ Newton-Raphson result for $f(c) := \int_\Omega \sigma(\psi + c) \, \mathrm{d}x - \theta \mathrm{vol}(\Omega)$
    $\psi \leftarrow \psi + c$ ▷ Performed in-place

5.2. Deriving the Lagrangian.

Following [GW21], we will be able to construct the required gradient using the Lagrangian; for this, we first require the weak form of the forward problem. Note that neither of the conditions on ρ itself ([3] and [4] in the constraints to Equation 13) will be covered here, as they are incorporated into and satisfied automatically as a result of the projection algorithm.

5.2.1. *Weak Form for Poisson.* For the PDE, we first multiply by a test function w and integrate over Ω, yielding

$$\int_\Omega -\nabla \cdot (r(\tilde{\rho})\nabla u) \cdot w = \int_\Omega f \cdot w. \tag{14}$$

Now, integrating by parts[2], we expand the left hand side to reach

$$-\int_{\partial\Omega} w(\nabla u) \cdot \mathbf{n} + \int_\Omega (r(\tilde{\rho})\nabla u) \cdot (\nabla w) = \int_\Omega f \cdot v. \tag{15}$$

5.2.2. *Weak Form for Helmholtz type Diffusion.* We assume homogeneous Dirichlet boundary conditions, so it is safe to omit the boundary term as in Equation 15.[3] Since all integrals are over the same domain, we will immediately write the inner products; multiplying by a test function $\tilde{w}$ and integrating, we begin with

$$(-\epsilon^2\Delta\tilde{\rho}, \tilde{w})_\Omega + (\tilde{\rho}, \tilde{w})_\Omega = (\rho, \tilde{w})_\Omega. \tag{16}$$

Integrating by parts similarly, we reach

$$(\epsilon^2\nabla\tilde{\rho}, \nabla\tilde{w})_\Omega + (\tilde{\rho}, \tilde{w})_\Omega = (\rho, \tilde{w})_\Omega \tag{17}$$

and are done.

5.2.3. *Constructing the Lagrangian from the Weak Forms.* Finally, noting that the objective can be written as (f, u), we reach

$$\begin{aligned}\mathcal{L}(u, \rho, \tilde{\rho}, w, \tilde{w}) = (f, u)_\Omega - (r(\tilde{\rho})\nabla u, \nabla w)_\Omega + (\nabla u \cdot \mathbf{n}, w)_{\partial\Omega} \\ + (f, w)_\Omega - (\epsilon^2\nabla\tilde{\rho}, \nabla\tilde{w})_\Omega - (\tilde{\rho}, \tilde{w})_\Omega + (\rho, \tilde{w})_\Omega\end{aligned} \tag{18}$$

5.3. **The Gradient.** Following [GW21] and to satisfy Keith and Surowiec's algorithm (2023), we work towards the construction of the functional derivative G, for which we require the Gateaux derivatives in each direction but ρ; these derivatives will recover several variational problems of interest.

First, we will recall the definition.

Definition 1. Gateaux Derivative Let X and Y be Banach and $U \subseteq X$ be open. For $F : X \to Y$, the Gateaux differential of F at $u \in U$ in the direction $\psi \in X$ is defined as

$$\frac{d}{dt}F(u + t\psi)\bigg|_{t=0}. \tag{19}$$

F is Gateaux differentiable at u if the limit exists for all $\psi \in X$. Supposing F is differentiable at u, the Gateaux derivative at u will be denoted $\partial_u F$.

We will assume basic linearity properties, which are fairly clear from the definition and essentially inherited from the linearity of the limit. The following formulas will be useful. The first is rather intuitive.

Lemma 5.1 (Gateaux derivative is zero against a "constant functional")**.** *Assuming that v and w are not functions of u, it holds that*

$$\partial_u(v, w) = 0. \tag{20}$$

[2]Problem set 4; this is a result of the divergence theorem and the product rule for gradients.

[3]For a bit more detail, $u|_{\partial\Omega} = 0 \implies \nabla u \cdot \mathbf{n}|_{\partial\Omega} = 0$; so, the integral on $\partial\Omega$ that appears is certainly 0.

Proof. Observe that

$$\partial_u(v,w) = \frac{d}{dt}(v,w)\bigg|_{t=0} = 0|_{t=0} = 0.$$

□

Lemma 5.2 (Simple inner product formula)**.** *Assuming that* $u, v \in L^2$,[4] *It holds that*

$$\partial_u(v,u) = (v,\psi). \tag{21}$$

Moreover, by symmetry of the inner product, we have equivalently that

$$\partial_u(u,v) = (\psi,v). \tag{22}$$

Proof. We will demonstrate the first and leave the second to symmetry.

$$\begin{aligned}\partial_u(v,u) &= \frac{d}{dt}(v,u+t\psi)\bigg|_{t=0} \\ &= \frac{d}{dt}\int_\Omega v\cdot(u+t\psi),\end{aligned}$$

but this is

$$\lim_{t\to 0}\frac{\int_\Omega v\cdot(u+t\psi)-\int_\Omega v\cdot u}{t} = \lim_{t\to 0}\frac{t\int_\Omega v\cdot\psi}{t} = \int_\Omega v\cdot\psi,$$

as desired. □

Lemma 5.3 (Poisson weak form LHS formula)**.** *Under the same conditions, it holds that*

$$\partial_u(\nabla v,\nabla u) = (\nabla v,\nabla\psi) \tag{23}$$

Moreover,

$$\partial_u(\nabla u,\nabla v) = (\nabla u,\nabla\psi) \tag{24}$$

holds also by symmetry.

Proof. We will demonstrate the first and leave the second to symmetry.

$$\begin{aligned}\partial_u(\nabla v,\nabla u) &= \frac{d}{dt}(\nabla v,\nabla(u+t\psi))\bigg|_{t=0} \\ &= \frac{d}{dt}(\nabla v,t\nabla\psi)\bigg|_{t=0} \\ &= (\nabla v,\nabla\psi),\end{aligned}$$

where the second equality holds by 5.1 and the third by bilinearity and evaluation of the derivative and its restriction (notice t disappears in the derivative). □

[4]Noticing our discretization choices from above, this will be sufficient for application.

5.3.1. *Filter Equation.* Taking the derivative $\partial_{\tilde{w}}\mathcal{L}$, we will obtain the filter equation from the two-field SIMP approach. Skipping the application of 5.1, we have

$$\begin{aligned}\partial_{\tilde{w}}\mathcal{L} &= \partial_{\tilde{w}}\left[-(\epsilon^2\nabla\tilde{\rho},\nabla\tilde{w}) - (\tilde{\rho},\tilde{w}) + (\rho,\tilde{w})\right]\\ &= -(\epsilon^2\nabla\tilde{\rho},\nabla v) - (\tilde{\rho},v) + (\rho,v),\end{aligned}$$

by 5.3 and 5.2. Setting this equal to zero and moving terms around, solving the filter equation is to find $\tilde{\rho}$ such that

$$(\epsilon^2\nabla\tilde{\rho},\nabla v) + (\tilde{\rho},v) = (\rho,v) \qquad \forall v \in H^1.$$

5.3.2. *Primal Problem.* Taking the derivative $\partial_w\mathcal{L}$, we will obtain the primal problem (state evaluation). Again skipping the application of 5.1, we have

$$\begin{aligned}\partial_w\mathcal{L} &= \partial_w\left[-(r(\tilde{\rho})\nabla u,\nabla w) + (\nabla u\cdot\mathbf{n},w) + (f,w)\right]\\ &= -(r(\tilde{\rho})\nabla u,\nabla v) + (\nabla u\cdot\mathbf{n},v) + (f,v).\end{aligned}$$

We would now like to find u such that

$$(r(\tilde{\rho})\nabla u,\nabla v) - (\nabla u\cdot\mathbf{n},v) = (f,v) \qquad \forall v \in V.$$

5.3.3. *Dual Problem.* Taking the derivative $\partial_u\mathcal{L}$ yields the dual problem.

$$\begin{aligned}\partial_u\mathcal{L} &= \partial_u\left[(f,u) - (r(\tilde{\rho})\nabla u,\nabla w) + (\nabla u\cdot\mathbf{n},w)\right]\\ &= -(r(\tilde{\rho})\nabla v,\nabla w) + \partial_u(\nabla u\cdot\mathbf{n},w)_{\partial\Omega} + (f,u),\end{aligned}$$

for all of which we have formulas excepting the boundary term. For this derivative,

$$\begin{aligned}\partial_u(\nabla u\cdot\mathbf{n},w) &= \frac{d}{dt}(\nabla(u+tv)\cdot\mathbf{n},w)\Big|_{t=0}\\ &= \frac{d}{dt}t(\nabla v\cdot\mathbf{n},w)\Big|_{t=0}\\ &= (\nabla v\cdot\mathbf{n},w)\end{aligned}$$

by the linearity of ∇ and properties of the dot product. So,

$$\partial_u\mathcal{L} = -(r(\tilde{\rho})\nabla v,\nabla w) + (\nabla v\cdot\mathbf{n},w) + (f,v).$$

Again setting this equal to zero, we would now like to find w such that

$$(r(\tilde{\rho})\nabla v,\nabla w) - (\nabla v\cdot\mathbf{n},w) = (f,v) \qquad \forall v \in V.$$

By appealing again to symmetry, we see that this is identical to the primal problem.

5.3.4. *Filtered Gradient.* We require finally $\partial_{\tilde{\rho}}\mathcal{L}$.

$$\begin{aligned}\partial_{\tilde{\rho}}\mathcal{L} &= \partial_{\tilde{\rho}}[-(r(\tilde{\rho})\nabla u,\nabla w) - (\epsilon^2\nabla\tilde{\rho},\nabla\tilde{w}) - (\tilde{\rho},\tilde{w})]\\ &= -(\epsilon^2\nabla v,\nabla\tilde{w}) - (v,\tilde{w}) - \partial_{\tilde{\rho}}(r(\tilde{\rho})\nabla u,\nabla w)\\ &= -(\epsilon^2\nabla\tilde{w},\nabla v) - (\tilde{w},v) - \partial_{\tilde{\rho}}(r(\tilde{\rho})\nabla u,\nabla w).\end{aligned}$$

For the derivative of the third term, for convenience, the SIMP law $r(\tilde{\rho})$ is provided again for convenience. ρ_0 denotes the minimum value ρ may take on and is used for numerical reasons; typically, this will be set a relatively very small value, like `1e-6`.

$$r(\tilde{\rho}) := \rho_0 + \tilde{\rho}^3(1-\rho_0)$$

So,

$$r'(\tilde{\rho}) = 3\cdot\tilde{\rho}^2(1-\rho_0).$$

Now, with the differential interpass being justified under the functions' regularity and the dominated convergence theorem,

$$\begin{aligned}\partial_{\tilde{\rho}}(r(\tilde{\rho})\nabla u, \nabla w) &= \frac{d}{dt}(r(\tilde{\rho}+tv)\nabla u, \nabla w)\Big|_{t=0} \\ &= (r'(\tilde{\rho}+tv)v\nabla u, \nabla w)\Big|_{t=0} \\ &= (r'(\tilde{\rho})\nabla w \nabla u, v) \\ &= (r'(\tilde{\rho})|\nabla u|^2, v),\end{aligned}$$

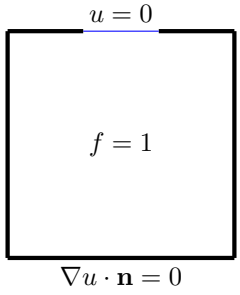
by the equivalence of the primal and dual problems (ergo their solutions; it is from this that we deduce the final equivalence). Setting the derivative equal to zero, we would like to find $\tilde{\rho}$ satisfying

$$(\epsilon^2 \nabla \tilde{w}, \nabla v) + (\tilde{w}, v) = (-r'(\tilde{\rho})|\nabla u|^2, v), \qquad \forall v \in H^1.$$

5.4. **Results.** We will now simulate a few problems over varying domains subject to varying boundary conditions. What remains is to describe the design domains Ω and apply boundary conditions as appropriate.

5.4.1. *Unit Square with Single Sink.* We will first implement the problem solved in `toph.jl`, that is, solve Equation 13 over the unit square under uniform, constant heating. Heat is allowed to escape through the top third (marked blue; corresponding to 0 Dirichlet boundary conditions), and elsewhere there is no flow (corresponding to 0 Neumann boundary conditions). The following simulation was run with step-size $\alpha = 0.1$ and mass fraction $\theta = 0.4$. Briefly on the physicality of the solution, first, note that the problem can

FIGURE 6. Ω for the first heat conduction problem.



be interpreted as the construction of a heat sink within Ω for which we specify heating loads and where the heat is to dissipate out to, in this case the load being uniform and constant over Ω (ie, the heating that the heat sink is subjected to is uniform throughout the domain) with the heat being able to dissipate out one end.

We see, generally, the sort of surface area maximization that intuition would serve with the tree-like branching, just as we observe the presence of triangles in solutions to the `top88.jl` structural compliance problems.

5.4.2. *Unit Square with Two Sinks.* Here, we allow for heat to escape also through a second port; due to the homogeneous boundary conditions, we end up with an identical problem excepting the boundary condition specification (ie, there are still no boundary terms). This amounts to a minor change in the boundary element labeling in the implementation (ie, that the lower third must now also be identified with the same attribute as the top third).

FIGURE 7. Ω with two sinks.

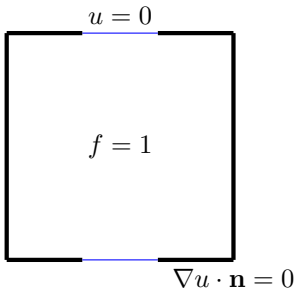
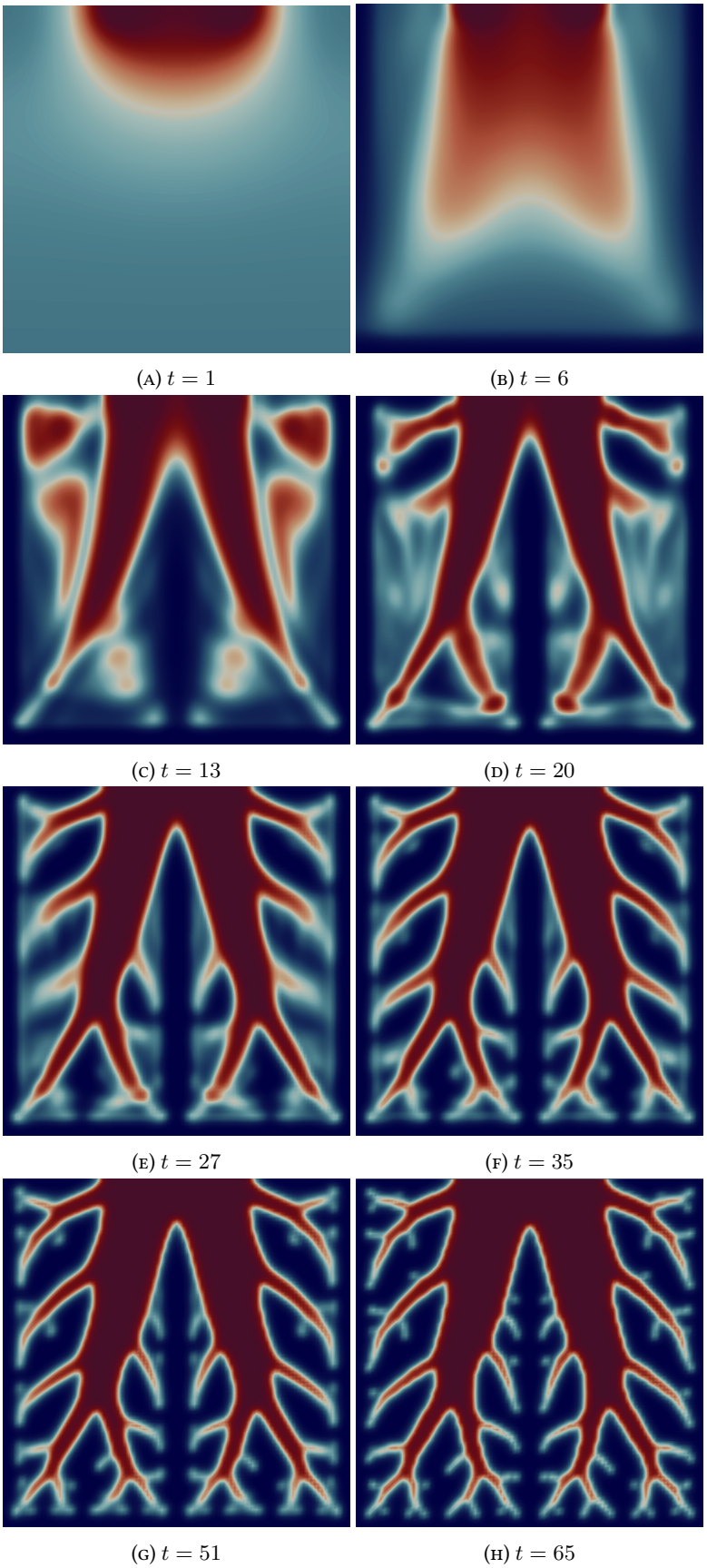


FIGURE 8. Unit Square with Single Sink: Filtered Distribution



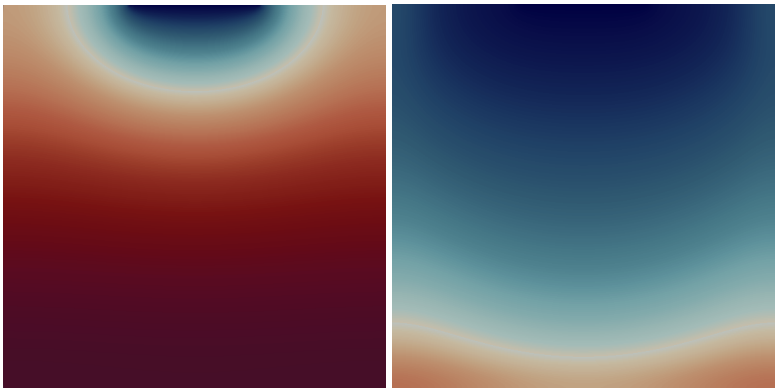
(A) $t = 1$ (B) $t = 6$

(C) $t = 13$ (D) $t = 20$

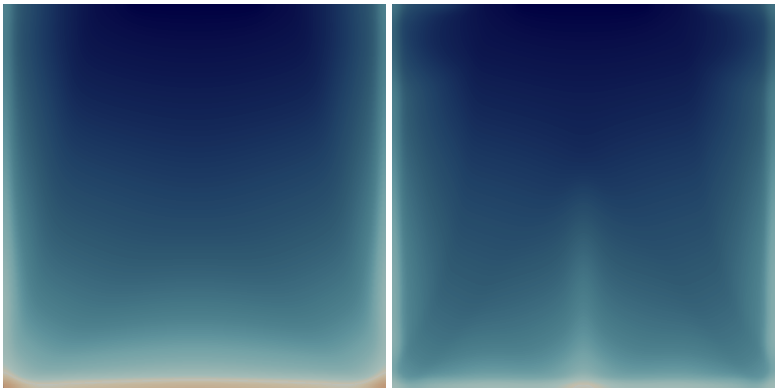
(E) $t = 27$ (F) $t = 35$

(G) $t = 51$ (H) $t = 65$

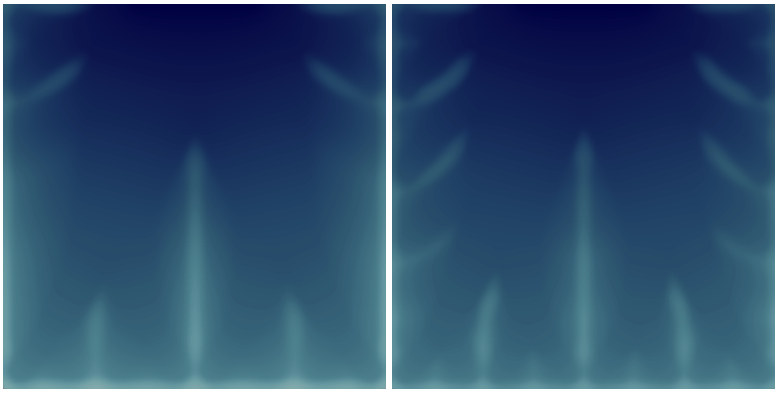
FIGURE 9. Unit Square with Single Sink: State Solution



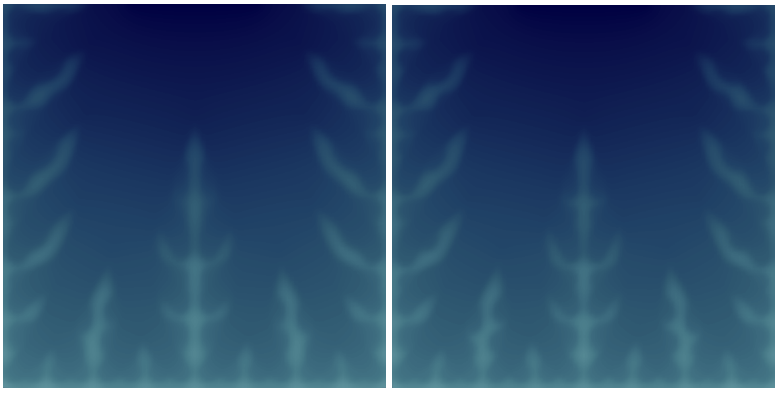
(A) $t = 1$ (B) $t = 6$



(C) $t = 13$ (D) $t = 20$

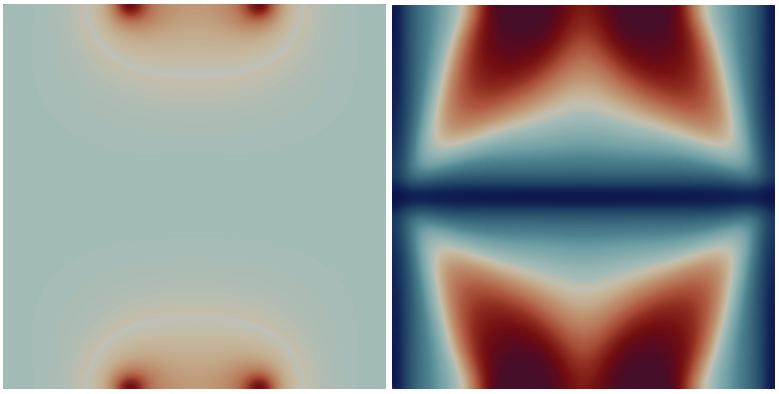


(E) $t = 27$ (F) $t = 35$

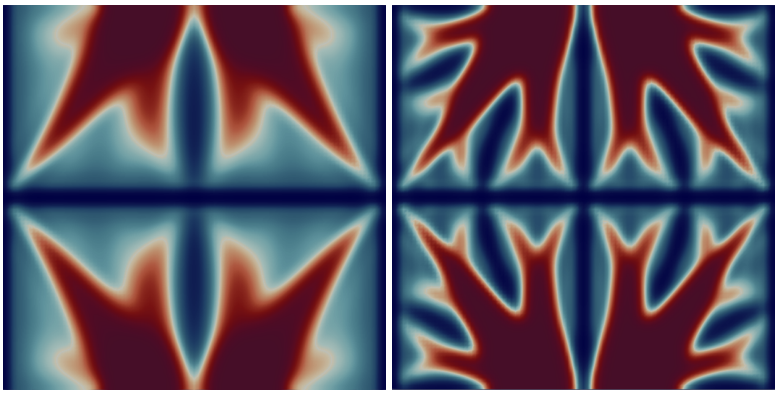


(G) $t = 51$ (H) $t = 65$

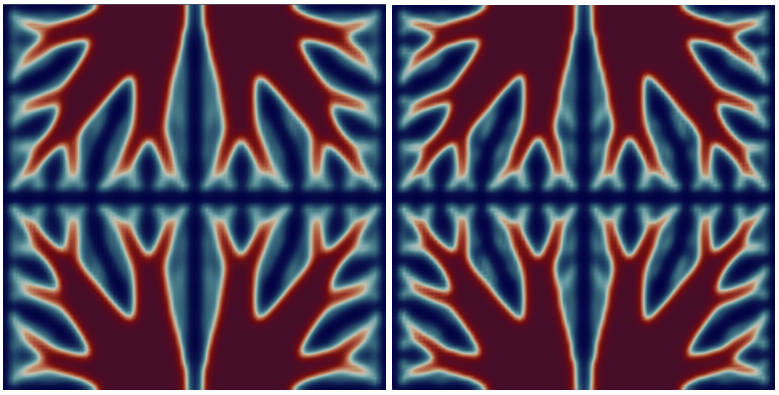
FIGURE 10. Unit Square with Two Sinks: Filtered Distribution



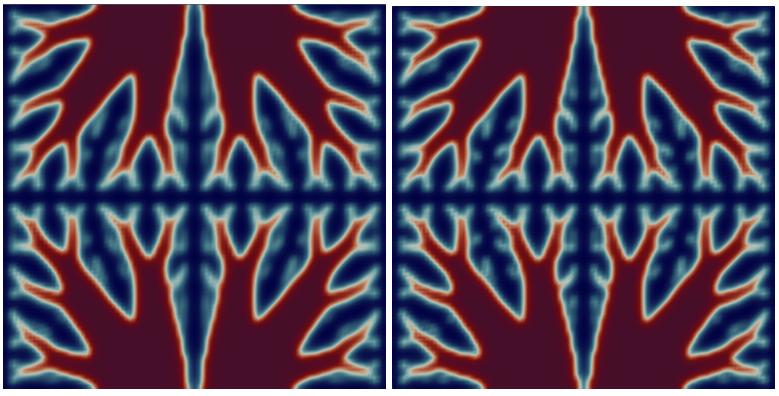
(A) $t = 2$ (B) $t = 16$



(C) $t = 32$ (D) $t = 48$

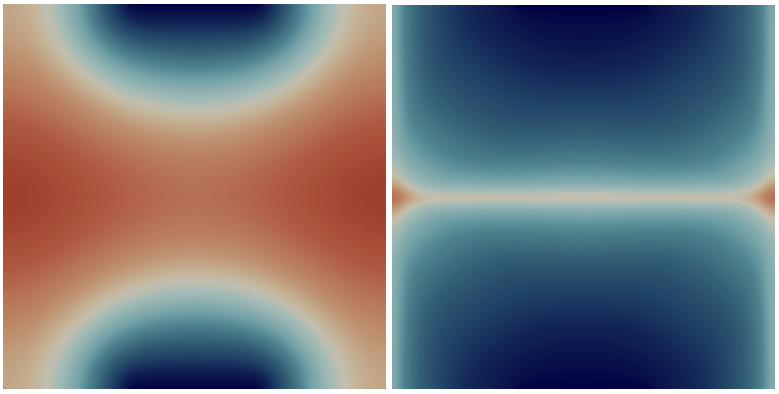


(E) $t = 64$ (F) $t = 80$

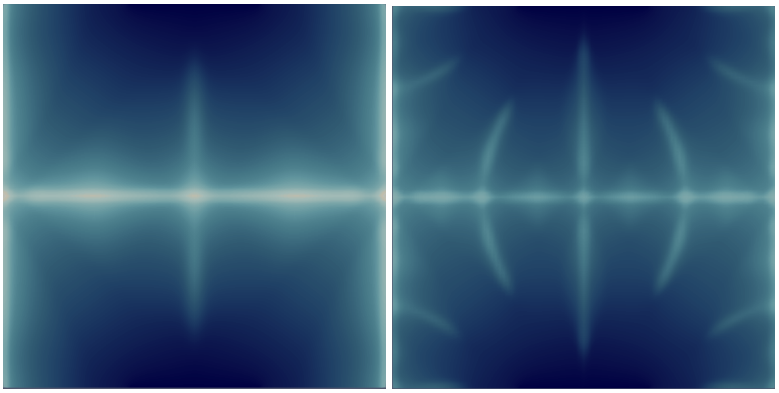


(G) $t = 96$ (H) $t = 127$

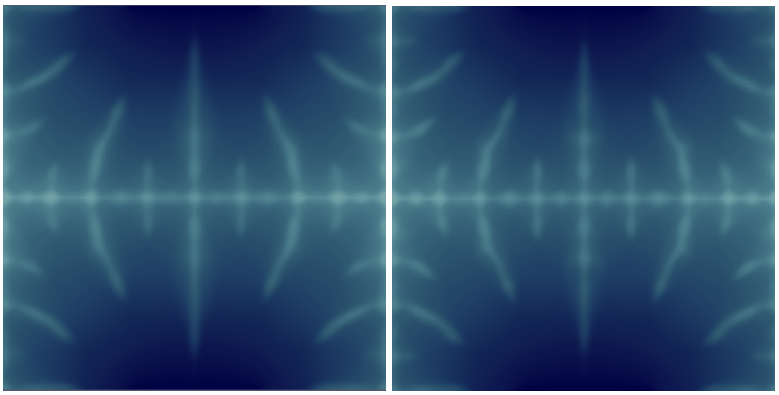
FIGURE 11. Unit Square with Two Sinks Simulation: State Solution



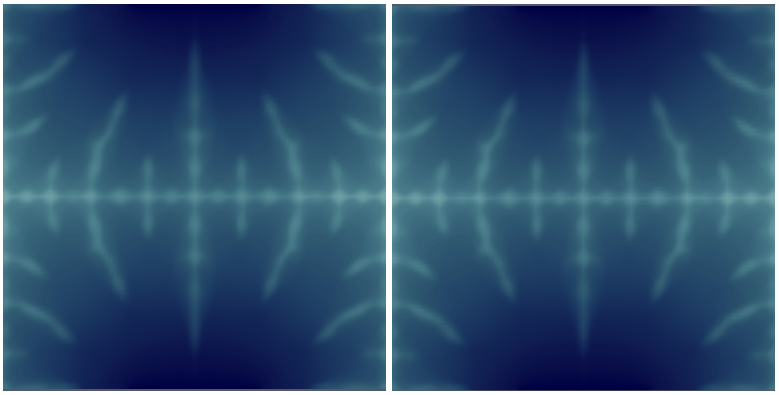
(A) $t = 2$ (B) $t = 16$



(C) $t = 32$ (D) $t = 48$



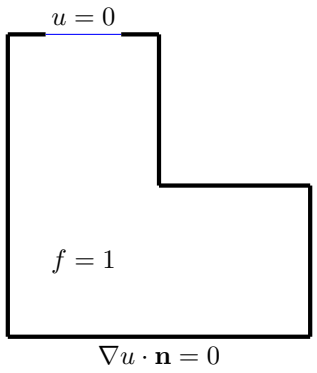
(E) $t = 64$ (F) $t = 80$



(G) $t = 96$ (H) $t = 127$

5.4.3. *L-shaped Domain with a Single Sink.* Here, now over an L-shaped domain, we allow for heat to escape part of the top edge. The interior is also constantly and uniformly heated. The problem remains otherwise the same.

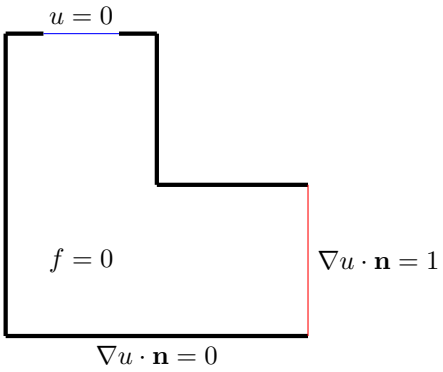
FIGURE 12. L-shaped Ω with a sink.



We again see the tree-like structure here, maximizing surface area; intuitively, that it wraps around the corner is an artifact of the mass constraint and that it needs to reach the south-east corner. The simulation was run with $\theta = 0.5$ and $\alpha = 0.001$.

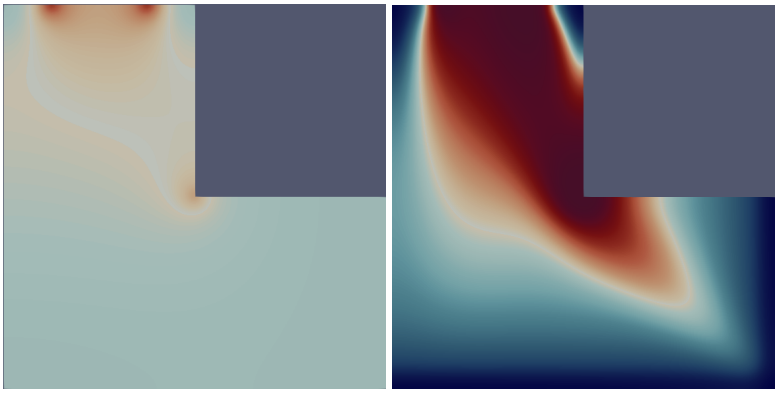
5.4.4. *L-shaped Domain with a Sink and a Source.* We will again allow for heat to escape part of the top edge, but instead of constant heating throughout, we will consider only a source (marked in red; corresponding to given non-homogeneous Neumann boundary conditions) on the right edge.

FIGURE 13. L-shaped Ω with a sink and source.

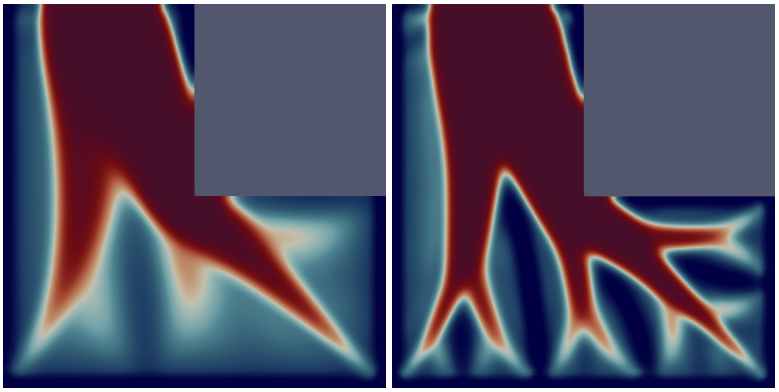


On the physicality of this solution, we can interpret this as simply taking the shortest path to distribute the heat to the sink. This simulation was also run with $\theta = 0.5$ and $\alpha = 0.001$.

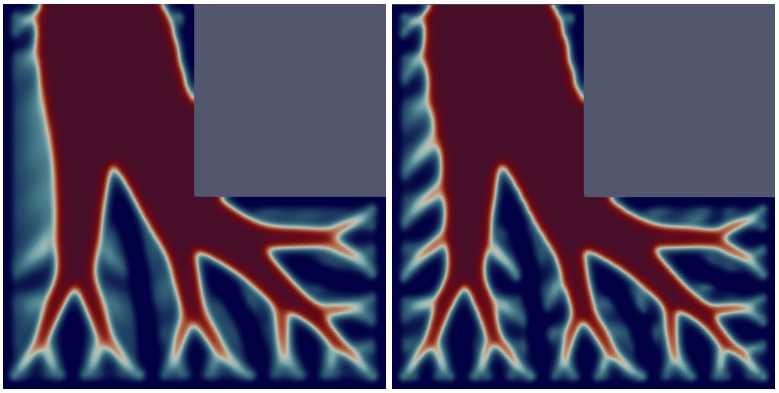
FIGURE 14. L-shape with a Sink: Filtered Distribution



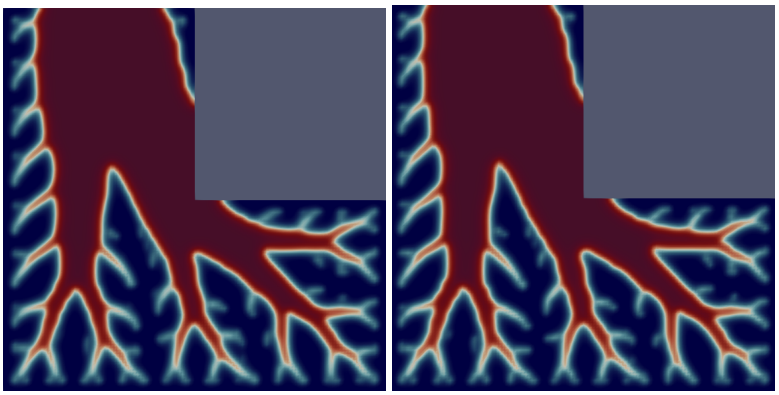
(A) $t = 2$ (B) $t = 25$



(C) $t = 50$ (D) $t = 75$

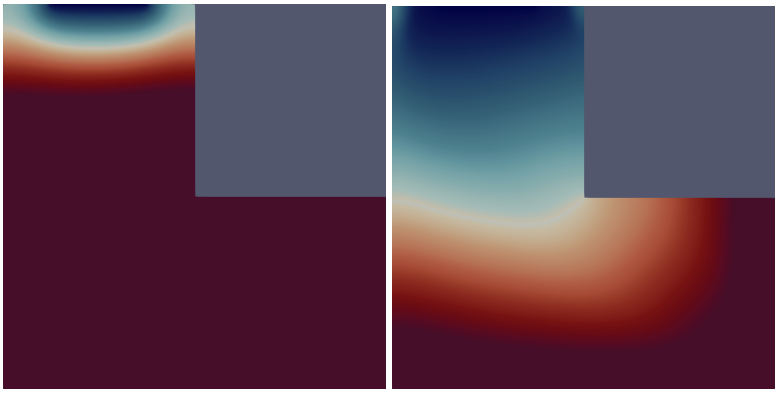


(E) $t = 100$ (F) $t = 125$

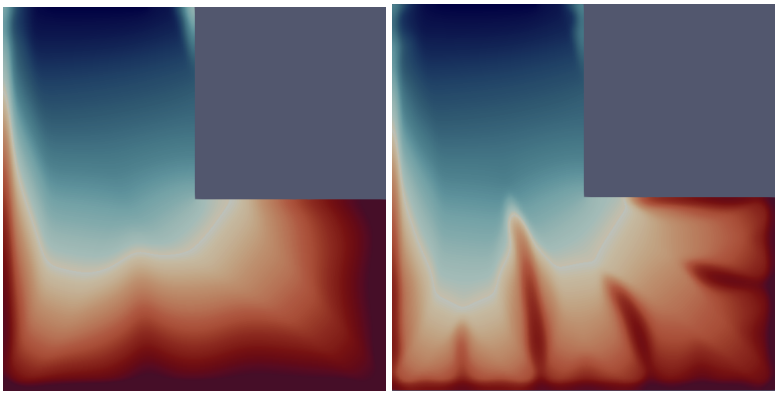


(G) $t = 175$ (H) $t = 225$

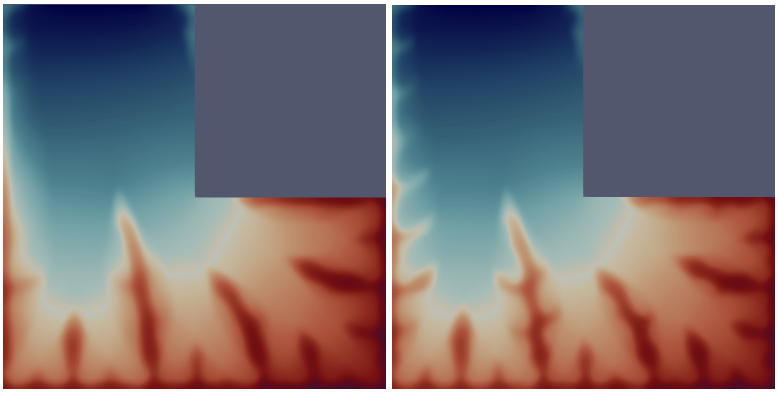
FIGURE 15. L-shape with a Sink: State Solution



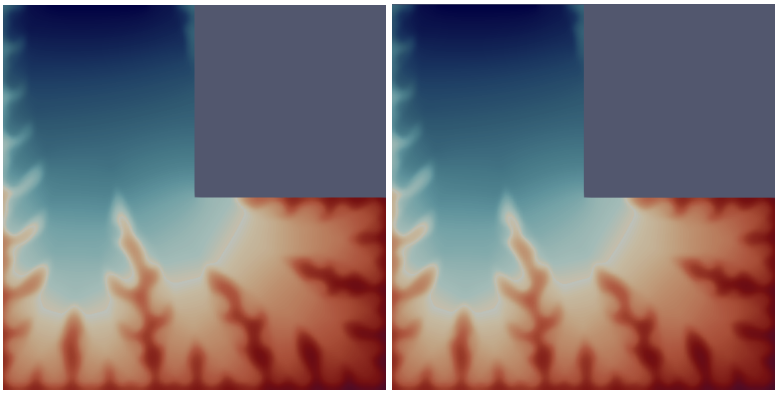
(A) $t = 2$ (B) $t = 25$



(C) $t = 50$ (D) $t = 75$

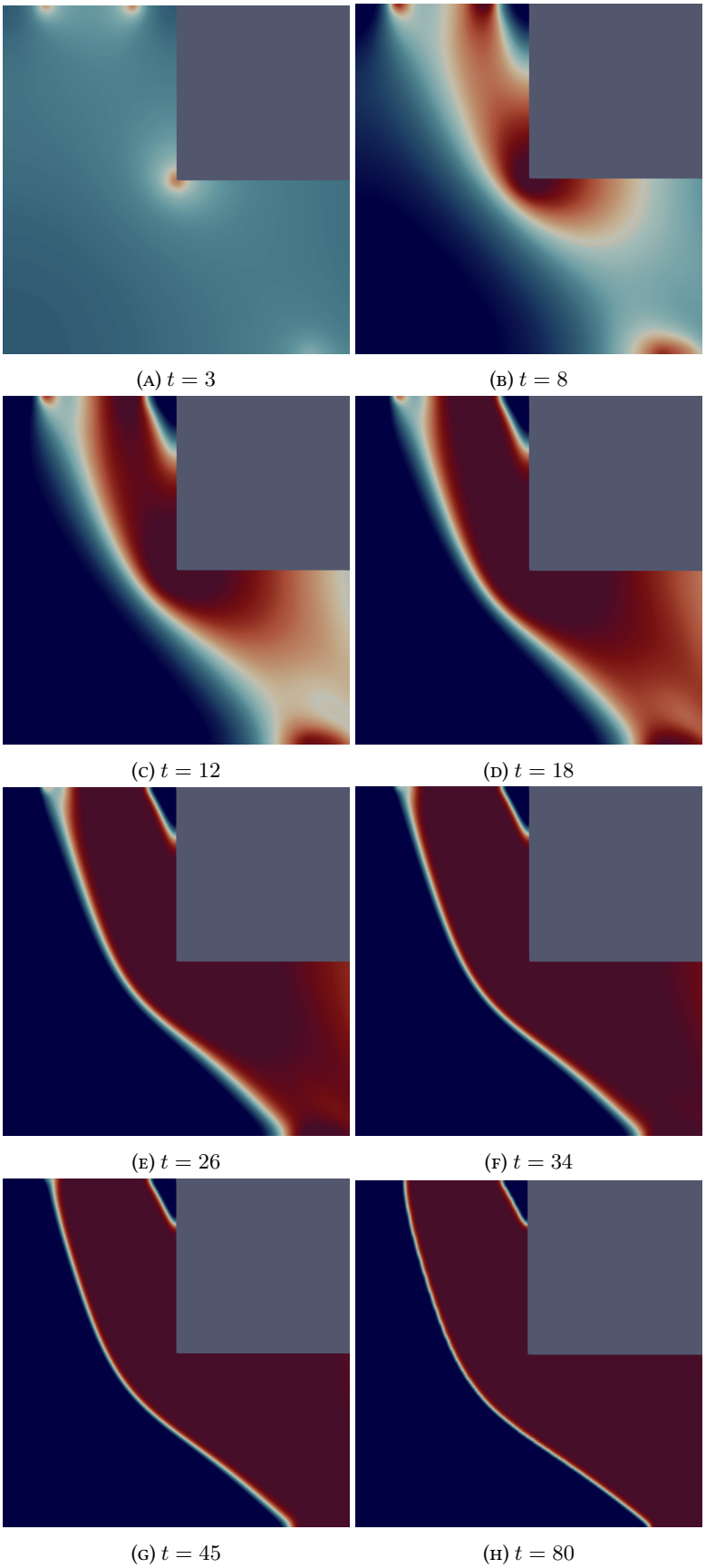


(E) $t = 100$ (F) $t = 125$



(G) $t = 175$ (H) $t = 225$

FIGURE 16. L-shape with a Sink and Source: Filtered Distribution



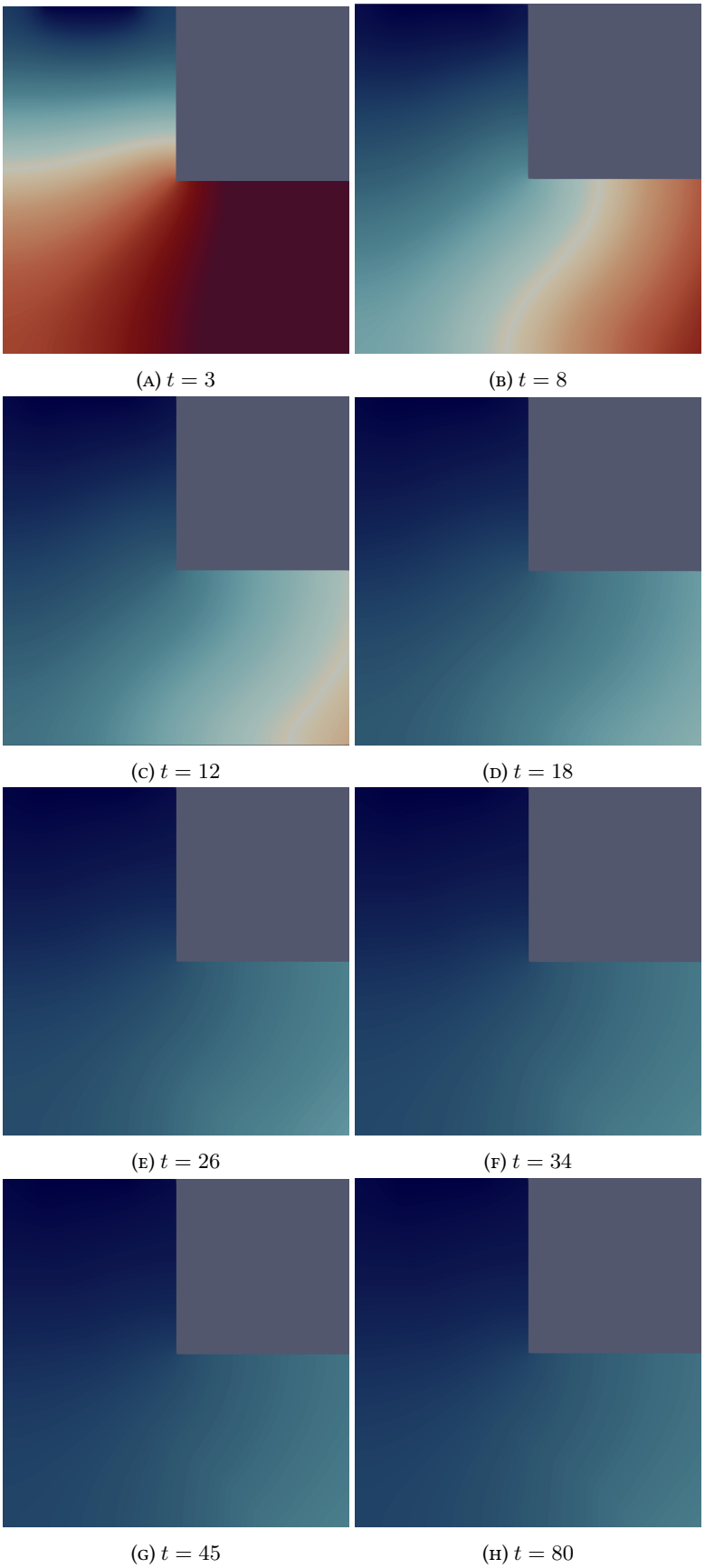
(A) $t = 3$ (B) $t = 8$

(C) $t = 12$ (D) $t = 18$

(E) $t = 26$ (F) $t = 34$

(G) $t = 45$ (H) $t = 80$

FIGURE 17. L-shape with a Sink and Source: State Solution



(A) $t = 3$ (B) $t = 8$

(C) $t = 12$ (D) $t = 18$

(E) $t = 26$ (F) $t = 34$

(G) $t = 45$ (H) $t = 80$

5.4.5. *CPU Heat Sink Problem.* Finally, we will consider a heat sink design problem, where the heat source is embedded within our design space Ω; on the "internal boundary" formed by the presence of this source, eg, a computer processor, we will impose nonhomogeneous Neumann BCs. Heat is allowed to escape on the left and right edges, where we might imagine there is a connection to a larger heatsink.

FIGURE 18. Heat-sink problem.

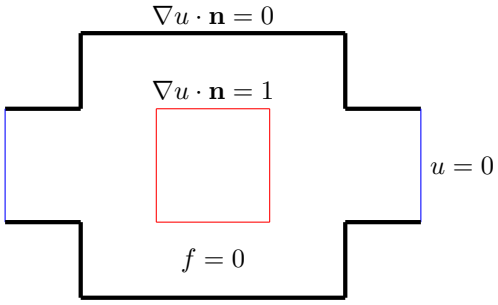
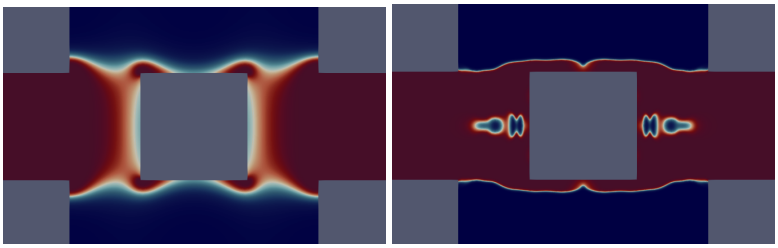
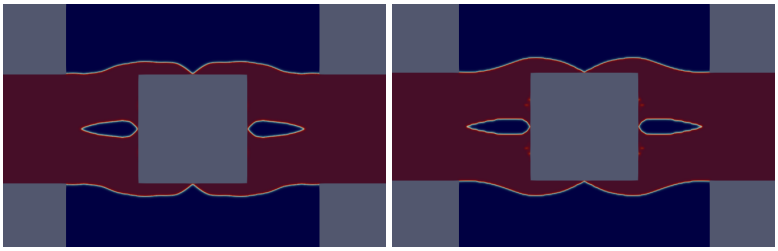


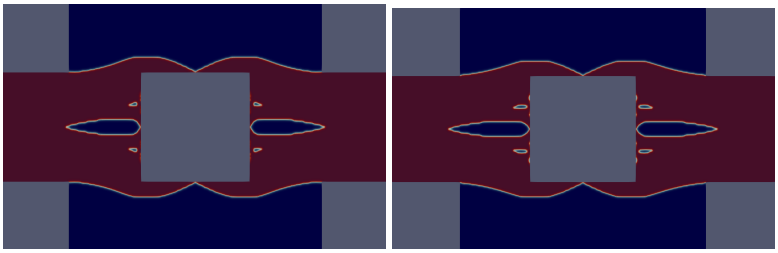
FIGURE 19. Heat sink problem: Filtered Distribution



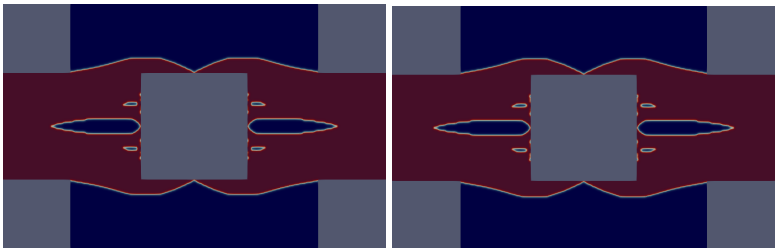
(A) $t = 2$ (B) $t = 5$



(C) $t = 8$ (D) $t = 30$

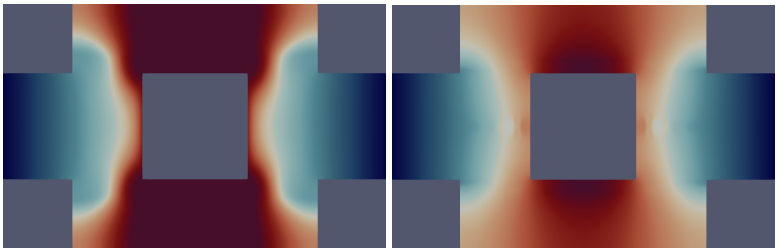


(E) $t = 60$ (F) $t = 90$

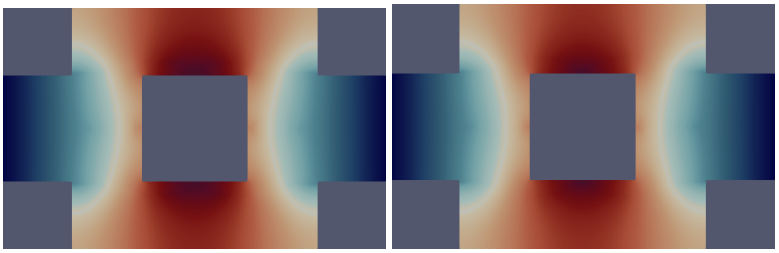


(G) $t = 120$ (H) $t = 150$

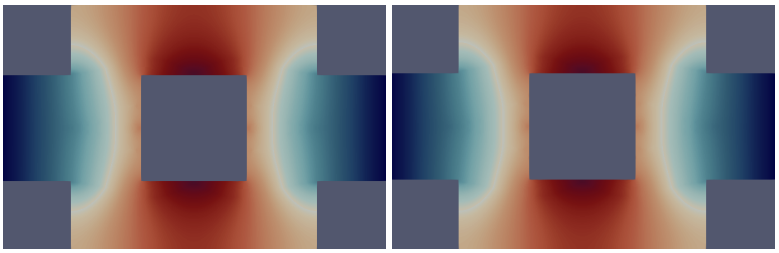
FIGURE 20. Heat sink problem: State Solution



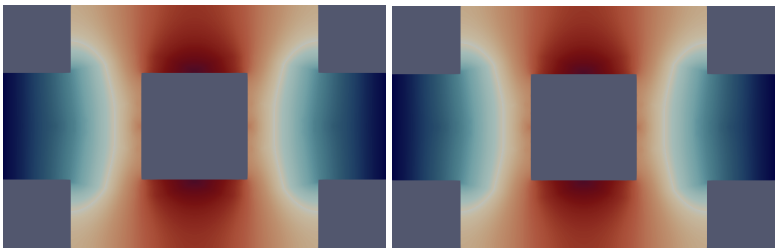
(A) $t = 2$ (B) $t = 5$



(C) $t = 8$ (D) $t = 30$



(E) $t = 60$ (F) $t = 90$



(G) $t = 120$ (H) $t = 150$

5.5. **Parallelization.** The parallelized code is available in `mainp.cxx` and also depends on `main.hxx`; the CMake currently does not build this code, but it can be copied into the MFEM examples directory, added to the Makefile,[5] and built there instead. Little of the topology algorithm is embarassingly parallel (in particular, since we are following an iterative optimization which is dependent on the previous step), so the parallelization is primarily handled behind the scenes by MFEM. The original unit square problem is hard-coded currently.

References

[BS03] Martin P. Bendsøe and O. Sigmund. *Topology Optimization: Theory, Methods, and Applications*. 2nd. Springer, 2003.

[And+10] Erik Andreassen et al. "Efficient topology optimization in MATLAB USING 88 lines of code". In: *Structural and Multidisciplinary Optimization* 43.1 (2010), pp. 1–16. DOI: 10.1007/s00158-010-0594-7.

[SM13] Ole Sigmund and Kurt Maute. "Topology Optimization approaches". In: *Structural and Multidisciplinary Optimization* 48.6 (2013), pp. 1031–1055. DOI: 10.1007/s00158-013-0978-6.

[GW21] Omar Ghattas and Karen Willcox. "Learning physics-based models from data: perspectives from inverse problems and model reduction". In: *Acta Numerica* 30 (2021), pp. 445–554. DOI: 10.1017/S0962492921000064.

Appendix A. `toph.jl` Derivation and Details

See Listing B for the implementation; here, we will work through the details to the code. `toph.jl` is a direct translation into Julia of the `toph` listing from the appendices in [BS03]. Due to Julia's design and syntax, the codes end up being quite similar, though the Julia version may be more familiar to a generally experienced programmer.

The problem is to minimize the thermal compliance in the unit square as was done in subsubsection 5.4.1.

A.1. **Primary Loop.** The main optimization loop is as follows:

Algorithm 3 `toph`: Main loop.

Require: `nelx` (number of x elements), `nely`, θ (volume fraction), p (penalization factor), r_{min} (filter radius)
Ensure: x satisfies the optimization problem.

```
x ← θ · 1                                   ▷ Initial guess is a matrix of θ's
while not converged do
    x_old ← x
    U ← FE; Displacement vector.            ▷ State solution.
    KE ← Element Stiffness Matrix
    c ← objective function calculation
    dc ← check                              ▷ Sensitivity filter.
    x ← OC          ▷ Design update step (by the optimality criteria method).
end while
```

The convergence condition depends on the largest value in the difference $x - x_{\text{old}}$.

[5] Importantly, the file extension must be changed to `.cpp` instead.

Appendix B. Julia Code Listings

The following are Julia versions of `top88` and `toph`.

top88.jl

```julia
module Top88

using LinearAlgebra
using SparseArrays
using Statistics

export top88
export prepare_filter
export OC

"""
    top88(nelx, nely, volfrac, penal, rmin, ft)

A direct, naive Julia port of Andreassen et al. "Efficient topology optimization in MATLAB
using 88 lines of code." By default, this will reproduce the optimized MBB beam from Sigmund
(2001).

# Arguments
- `nelx::S`: Number of elements in the horizontal direction
- `nely::S`: Number of elements in the vertical direction
- `volfrac::T`: Prescribed volume fraction
- `penal::T`: The penalization power
- `rmin::T`: Filter radius divided by the element size
- `ft::Bool`: Choose between sensitivity (if true) or density filter (if false). Defaults
    to sensitivity filter.
- `write::Bool`: If true, will write out iteration number, changes, and density for each
    iteration. Defaults for false.
- `loop_max::Int`: Explicitly set the maximum number of iterations. Defaults to 1000.

# Returns
- `Matrix{T}`: Final material distribution, represented as a matrix
"""
function top88(
    nelx::S=60,
    nely::S=20,
    volfrac::T=0.5,
    penal::T=3.0,
    rmin::T=2.0,
    ft::Bool=true,
    write::Bool=false,
    loop_max::Int=1000
) where {S <: Integer, T <: AbstractFloat}
    # Physical parameters
    E0 = 1; Emin = 1e-9; nu = 0.3;

    # Prepare finite element analysis
    A11 = [12  3 -6 -3;  3 12  3  0; -6  3 12 -3; -3  0 -3 12]
    A12 = [-6 -3  0  3; -3 -6 -3 -6;  0 -3 -6  3;  3 -6  3 -6]
    B11 = [-4  3 -2  9;  3 -4 -9  4; -2 -9 -4 -3;  9  4 -3 -4]
    B12 = [ 2 -3  4 -9; -3  2  9 -2;  4  9  2  3; -9 -2  3  2]
    KE = 1/(1-nu^2)/24*([A11 A12;A12' A11]+nu*[B11 B12;B12' B11])

    nodenrs = reshape(1:(1+nelx)*(1+nely),1+nely,1+nelx)
    edofVec = reshape(2*nodenrs[1:end-1,1:end-1].+1, nelx*nely, 1)
    edofMat = zeros(Int64, nelx*nely, 8)

    offsets = [0 1 2*nely.+[2 3 0 1] -2 -1]
    for i = 1:8
        for j = 1:nelx*nely
            edofMat[j,i]= edofVec[j] + offsets[i]
        end
    end

    iK = reshape(kron(edofMat,ones(8,1))', 64*nelx*nely,1)
    jK = reshape(kron(edofMat,ones(1,8))', 64*nelx*nely,1)

```

```
    # Loads and supports
    F = spzeros(2*(nely+1)*(nelx+1))
    F[2,1] = -1
    U = spzeros(2*(nely+1)*(nelx+1))

    fixeddofs = union(1:2:2*(nely+1), [2*(nelx+1)*(nely+1)])
    alldofs = 1:2*(nely+1)*(nelx+1)
    freedofs = setdiff(alldofs, fixeddofs)

    # Prepare the filter
    H, Hs = prepare_filter(nelx, nely, rmin)

    # Initialize iteration
    x = volfrac*ones(nely,nelx)
    xPhys = x
    loop = 0
    change = 1
    cValues = []

    # Start iteration
    while change > 0.01
        loop += 1
        # FE-Analysis
        sK = [j*((i+Emin)^penal) for i in ((E0-Emin)*xPhys[:]') for j in KE[:]]
        K = sparse(iK[:], jK[:], sK)
        K = (K+K')/2

        KK = cholesky(K[freedofs,freedofs])
        U[freedofs] = KK \ F[freedofs]

        # OLD: edM = [convert(Int64,i) for i in edofMat]
        mat = (U[edofMat]*KE).*U[edofMat]

        # Objective function and sensitivity analysis
        ce = reshape([sum(mat[i,:]) for i = 1:(size(mat)[1])],nely,nelx)
        c = sum(sum((Emin*ones(size(xPhys)).+(xPhys.^penal)*(E0-Emin)).*ce))
        push!(cValues,c)
        dc = -penal*(E0-Emin)*xPhys.^(penal-1).*ce
        dv = ones(nely,nelx)

        # Filtering/ modification of sensitivities
        if ft
            dc[:] = H*(x[:].*dc[:])./Hs./max(1e-3,maximum(x[:]))
        else
            dc[:] = H*(dc[:]./Hs)
            dv[:] = H*(dv[:]./Hs)
        end

        # Optimality criteria update of design variables and physical densities
        xnew = OC(nelx, nely, x, volfrac, dc, dv, xPhys, ft)

        change = maximum(abs.(x-xnew))
        x = xnew

        write && println("Loop = ", loop, ", Change = ", change ,", c = ", c, ", structural density = ", mean(x))
        loop >= loop_max && break
    end

    return x
end


"""
Prepare sensitivity/ density filter
"""
function prepare_filter(nelx::S, nely::S, rmin::T) where {S <: Integer, T <: AbstractFloat}
    iH = ones(nelx*nely*(2*(convert(Int64,ceil(rmin)-1))+1)^2)
    jH = ones(size(iH))
    sH = zeros(size(iH))
    k = 0
    for i1 = 1:nelx
        for j1 = 1:nely
```

```
            e1 = (i1-1)*nely+j1
            for i2 = max(i1-(ceil(rmin)-1),1):min(i1+(ceil(rmin)-1),nelx)
                for j2 = max(j1-(ceil(rmin)-1),1):min(j1+(ceil(rmin)-1),nely)
                    e2 = (i2-1)*nely+j2
                    k += 1
                    iH[k] = e1
                    jH[k] = e2
                    sH[k] = max(0,rmin-sqrt((i1-i2)^2+(j1-j2)^2))
                end
            end
        end
    end
    H = sparse(iH,jH,sH)
    Hs = [sum(H[i,:]) for i = 1:(size(H)[1])]

    return H, Hs
end

"""
Optimality criteria update
"""
function OC(
    nelx::S,
    nely,
    x,
    volfrac,
    dc::Matrix{T},
    dv,
    xPhys::Matrix{T},
    ft::Bool
) where {S <: Integer, T <: AbstractFloat}
    l1 = 0; l2 = 1e9; move = 0.2
    xnew = zeros(nely, nelx)

    while (l2-l1)/(l1+l2) > 1e-3
        lmid = 0.5*(l2+l1)
        RacBe = sqrt.(-dc./dv/lmid)
        XB = x.*RacBe

        for i = 1:nelx
            for j = 1:nely
                xji = x[j,i]
                xnew[j,i]= max(0.000,max(xji-move,min(1,min(xji+move,XB[j,i]))))
            end
        end

        if ft
            xPhys = xnew
        else
            xPhys[:] = (H*xnew[:])./Hs
        end

        if sum(xPhys[:]) > volfrac*nelx*nely
            l1 = lmid
        else
            l2 = lmid
        end
    end

    return xnew
end

end
```

toph.jl

```
module TopH

using LinearAlgebra
using SparseArrays
using Statistics

```

```
export toph
export OC
export check
export FE

"""
    toph(nelx, nely, volfrac, penal, rmin, write, loop_max)

A direct, naive Julia port of the `toph` code listing from "Topology Optimization"
by Martin Bendsoe and Ole Sigmund.

# Arguments
- `nelx::S`: Number of elements in the horizontal direction
- `nely::S`: Number of elements in the vertical direction
- `volfrac::T`: Prescribed volume fraction
- `penal::T`: The penalization power
- `rmin::T`: Filter radius divided by the element size
- `write::Bool`: If true, will write out iteration number, changes, and density
    for each iteration. Defaults to false.
- `loop_max::Int`: Explicitly set the maximum number of iterations. Defaults to 1000.

# Returns
- `Matrix{T}`: Final material distribution, represented as a matrix.
"""
function toph(
    nelx::S,
    nely::S,
    volfrac::T,
    penal::T,
    rmin::T,
    write::Bool=false,
    loop_max::Int=100
) where {S <: Integer, T <: AbstractFloat}
    # Initialization
    x = volfrac * ones(nely,nelx)
    loop = 0
    change = 1.
    dc = zeros(nely,nelx)

    # Start iteration
    while change > 0.01
        loop += 1
        xold = x
        c = 0.

        # FE Analysis
        U = FE(nelx,nely,x,penal)

        KE = [ 2/3 -1/6 -1/3 -1/6
                -1/6  2/3 -1/6 -1/3
                -1/3 -1/6  2/3 -1/6
                -1/6 -1/3 -1/6  2/3 ]

        # Objective function/ sensitivity analysis
        for ely = 1:nely
            for elx = 1:nelx
                n1 = (nely+1)*(elx-1)+ely
                n2 = (nely+1)* elx   +ely
                Ue = U[[n1; n2; n2+1; n1+1]]

                c += (0.001+0.999*x[ely,elx]^penal)*Ue'*KE*Ue
                dc[ely,elx] = -0.999*penal*(x[ely,elx])^(penal-1)*Ue'*KE*Ue
            end
        end

        # Sensitivity filtering
        dc = check(nelx,nely,rmin,x,dc)
        # Design update by optimality criteria method
        x  = OC(nelx,nely,x,volfrac,dc)

        # Print out results if desired
        if write
```

```
            change = maximum(abs.(x-xold))
            println("Change = ", change, " c = ", c)
        end

        loop >= loop_max && break
    end

    return x
end

"""
    OC(nelx, nely, x, volfrac, dc)

Optimality criteria update

# Arguments
- `nelx::S`: Number of elements in the horizontal direction
- `nely::S`: Number of elements in the vertical direction
- `x::Matrix{T}`: Current material distribution
- `volfrac::T`: Prescribed volume fraction
- `dc::Matrix{T}`: Sensitivity filter

# Returns
- `Matrix{T}`: Updated material distribution

"""
function OC(
    nelx::S,
    nely::S,
    x::Matrix{T},
    volfrac::T,
    dc::Matrix{T}
) where {S <: Integer, T <: AbstractFloat}
    l1 = 0; l2 = 100000; move = 0.2
    xnew = zeros(nely,nelx)

    while (l2-l1) > 1e-4
        lmid = 0.5*(l2+l1)
        RacBe = sqrt.(-dc/lmid)
        XB = x .* RacBe

        for i = 1:nelx
            for j = 1:nely
                xji = x[j,i]
                xnew[j,i]= max(0.001,max(xji-move,min(1,min(xji+move,XB[j,i]))))
            end
        end

        if (sum(sum(xnew)) - volfrac*nelx*nely) > 0
            l1 = lmid
        else
            l2 = lmid
        end
    end

    return xnew
end

"""
    check(nelx, nely, rmin, x, dc)

Mesh independency filter

# Arguments
- `nelx::S`: Number of elements in the horizontal direction
- `nely::S`: Number of elements in the vertical direction
- `rmin::T`: Sensitivity filter radius divided by element size
- `x::Matrix{T}`: Current material distribution
- `dc::Matrix{T}`: Compliance derivatives

# Returns
- `Matrix{T}`: Updated dc
```

```
"""
function check(nelx::S,
    nely::S,
    rmin::T,
    x::Matrix{T},
    dc::Matrix{T}
) where {S <: Integer, T <: AbstractFloat}
    dcn=zeros(nely,nelx)

    for i = 1:nelx
        for j = 1:nely
        sum=0.0

        for k = max(i-floor(rmin),1):min(i+floor(rmin),nelx)
            for l = max(j-floor(rmin),1):min(j+floor(rmin),nely)
            l = Int64(l); k = Int64(k)
            fac = rmin-sqrt((i-k)^2+(j-l)^2)
            sum = sum+max(0,fac)
            dcn[j,i] += max(0,fac)*x[l,k]*dc[l,k]
            end
        end

        dcn[j,i] = dcn[j,i]/(x[j,i]*sum)
        end
    end

    return dcn
end

"""
    FE(nelx, nely, x, penal)

Finite element implementation

# Arguments
- `nelx::S`: Number of elements in the horizontal direction
- `nely::S`: Number of elements in the vertical direction
- `x::Matrix{T}`: Current material distribution
- `penal::T`: The penalization power

# Returns
- `Matrix{T}`: Differential equation solution U
"""
function FE(
    nelx::S,
    nely::S,
    x::Matrix{T},
    penal::T
) where {S <: Integer, T <: AbstractFloat}
    KE = [ 2/3 -1/6 -1/3 -1/6
            -1/6  2/3 -1/6 -1/3
            -1/3 -1/6  2/3 -1/6
            -1/6 -1/3 -1/6  2/3 ]

    K = spzeros((nelx+1)*(nely+1), (nelx+1)*(nely+1))
    U = zeros((nely+1)*(nelx+1))
    F = zeros((nely+1)*(nelx+1))
    for elx = 1:nelx
        for ely = 1:nely
            n1 = (nely+1)*(elx-1)+ely
            n2 = (nely+1)* elx   +ely
            edof = [n1; n2; n2+1; n1+1]
            K[edof,edof] += (0.001+0.999*x[ely,elx]^penal)*KE
        end
    end

    F .= 0.01
    fixeddofs = Int64(nely/2+1-(nely/20)):Int64(nely/2+1+(nely/20))
    alldofs = 1:(nely+1)*(nelx+1)
    freedofs = setdiff(alldofs,fixeddofs)

    U[freedofs] = K[freedofs, freedofs] \ F[freedofs]
```

```
    U[fixeddofs] .= 0

    return U
end

end
```