

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

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

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

1. Introduction	1
2. Classical Methods	2
2.1. Density Approach	3
2.2. Other Major Algorithms	4
2.3. Discrete Approaches: ESO	4
3. Structural Compliance with top88	5
3.1. Problem Specification	5
3.2. Result	6
4. Thermal Compliance with toph	7
4.1. Results	7
5. Thermal Compliance with the Entropic Finite Element Method (EFEM)	9
5.1. Method	9
5.2. Deriving the Lagrangian	9
5.3. The Gradient	10

5.4. Results	13
5.5. Parallelization	27
References	27
Appendix A. <code>toph.jl</code> Derivation and Details	27
A.1. Primary Loop	27
A.2. Finite Element Analysis	28
A.3. OC based optimizer	28
A.4. Mesh-independency filtering	28
Appendix B. Julia Code Listings	29

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

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

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

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

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 problems¹ 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 intuition.

¹as opposed to having several materials.

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 sensitivity 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 the domain Ω . It’s easy to see a natural extension of the explicit penalization [Equation 3](#) of the form

$$(4) \quad \alpha \int_{\Omega} \bar{\rho}(1 - \bar{\rho}),$$

where $\bar{\rho}$ 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 $\bar{\rho}$ over a length parameter r which can be related to r_{\min} from above. This can be written as a Helmholtz type diffusion

$$(5) \quad -r^2 \Delta \bar{\rho} + \bar{\rho} = \rho,$$

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 $\bar{\rho}$. 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 ramifications thereof are not well understood.

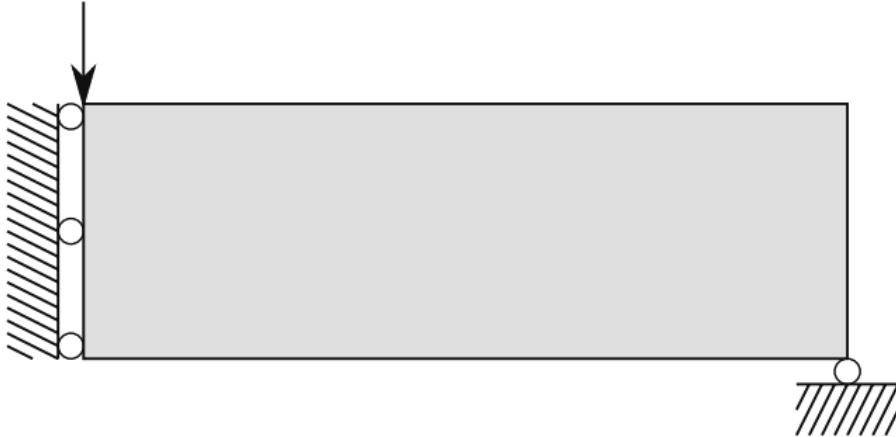
The level set approach’s central idea is to construct the geometry via the definition of a solid-void interface, rather than explicitly indicating where the material should be placed as is done with the density approach. Specifically, the design boundary will be specified by the zero level counter of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

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

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

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



considered rewarding in the analogous way (BESO). More recently came the genetic evolutionary structural optimization strategy (GESO), which combined the genetic algorithm with ESO.

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

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

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 singular stiffness matrix. [And+10] titles this a “modified SIMP approach.” Following the format of Equation 2, the problem may be written

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

Here, c defines the compliance, \mathbf{U} and \mathbf{F} denote resp. the global displacement and force vectors, \mathbf{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.

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

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

$$(10) \quad B_e = \frac{-\frac{\partial c}{\partial x_e}}{\lambda \frac{\partial V}{\partial x_e}},$$

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

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

and

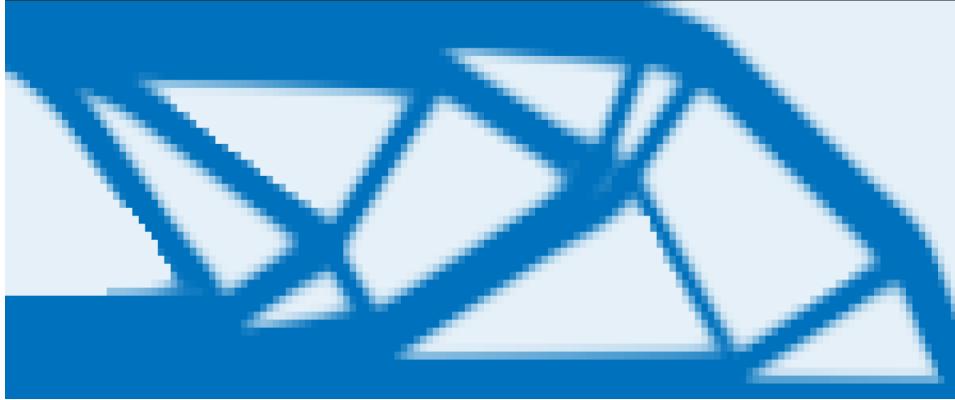
$$(12) \quad \frac{\partial V}{\partial x_e} = 1,$$

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.

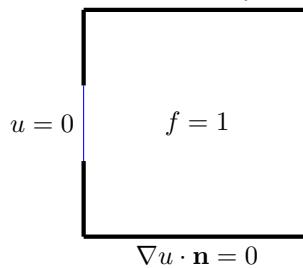
3.2. Result. Confirming roughly our intuition, we see a number of triangles forming. The following was run on a rectangle of 150x50 elements with a volume fraction of 0.5, $r_{\min}/V(e) = 2.0$ (ie, radius divided by element volume), and using a density filter.

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 40×40 square discretization, mass fraction $\theta = 0.4$, SIMP penalization exponent of 3.0, and r_{\min} of 1.2, and secondly with an 80×80 square.

Keeping the constant even heating in mind, the solution reached is fairly natural: the goal is essentially to 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.

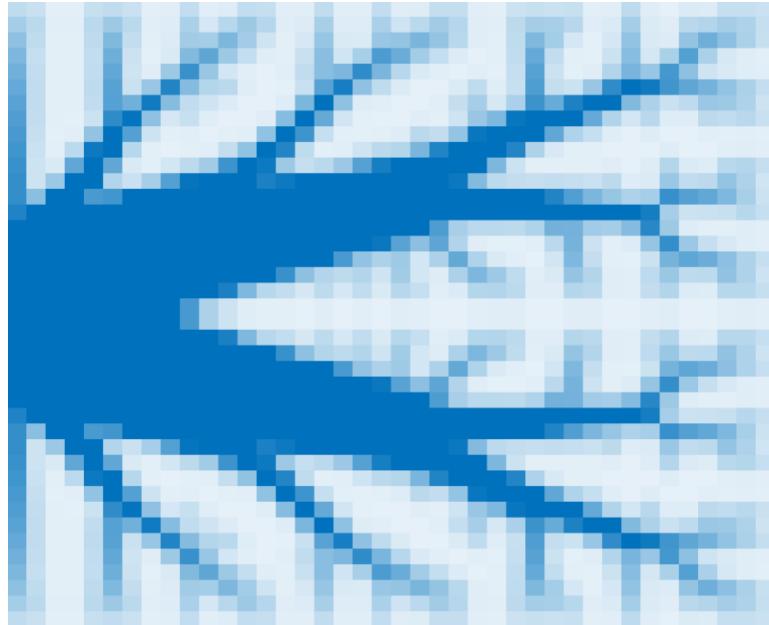
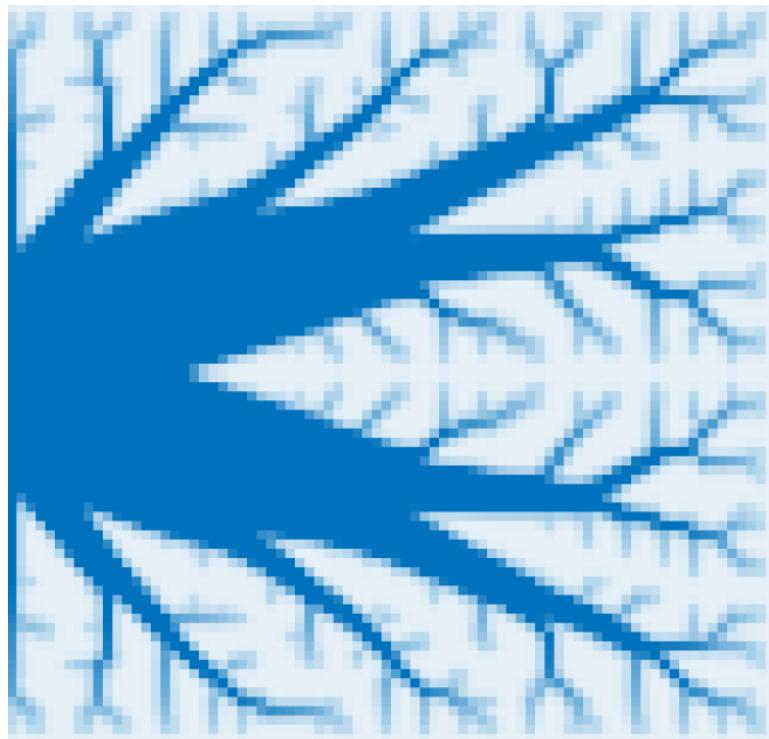


FIGURE 5. 80x80 toph simulation.



5. THERMAL COMPLIANCE WITH THE ENTROPIC FINITE ELEMENT METHOD (EFEM)

We would like to solve the thermal compliance problem

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

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, $\text{clip}(y) := \min(\max_val, \max(\min_val, y))$, and $\text{projit}(\psi)$ 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: $\rho = \arg \min \int_{\Omega} f \cdot u \, dx$

$\psi \leftarrow \sigma^{-1}(\theta)$	\triangleright So that $\int \sigma(\psi) = \theta \cdot \text{vol}(\Omega)$
while not converged do	
$\tilde{\rho} \leftarrow$ solution to $\partial_{\tilde{w}} \mathcal{L} = 0$,	\triangleright Filter solve
$u \leftarrow$ state solution; solution to $\partial_w \mathcal{L} = 0$,	\triangleright 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 $v \in L^2$,	
$\psi \leftarrow \text{clip}(\text{projit}(\psi - \alpha G))$.	\triangleright 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 $\text{projit}(\psi)$ Nonlinear Projection

Require: $\psi \in L^2(\Omega)$, $0 < \theta < 1$
Ensure: $\int_{\Omega} \sigma(\psi + c) \, dx = \theta \cdot \text{vol}(\Omega)$

$c \leftarrow$ Newton-Raphson result for $f(c) := \int_{\Omega} \sigma(\psi + c) \, dx - \theta \text{vol}(\Omega)$	\triangleright Performed in-place
$\psi \leftarrow \psi + c$	

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

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

Now, integrating by parts², we expand the left hand side to reach

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

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

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

Integrating by parts similarly, we reach

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

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

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

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 $F : X \rightarrow Y$, the Gateaux differential of F at $u \in U$ in the direction $\psi \in X$ is defined as

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

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*

$$(20) \quad \partial_u(v, w) = 0.$$

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

³For 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

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

□

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

$$(21) \quad \partial_u(v, u) = (v, \psi).$$

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

$$(22) \quad \partial_u(u, v) = (\psi, v).$$

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

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

but this is

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

as desired. □

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

$$(23) \quad \partial_u(\nabla v, \nabla u) = (\nabla v, \nabla \psi)$$

Moreover,

$$(24) \quad \partial_u(\nabla u, \nabla v) = (\nabla u, \nabla \psi)$$

holds also by symmetry.

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

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

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). □

⁴Noticing 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

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

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

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

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

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

We would now like to find u such that

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

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

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

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

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

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,

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

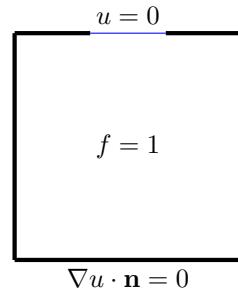
by the equivalence of the primal and dual problems (ergo their solutions; it is from this that we deduce the final equivalence). Setting the derivative equal to zero, we would like to find $\tilde{\rho}$ satisfying

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

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.

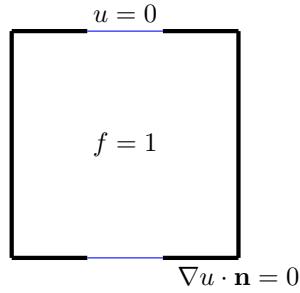


FIGURE 8. Unit Square with Single Sink: Filtered Distribution

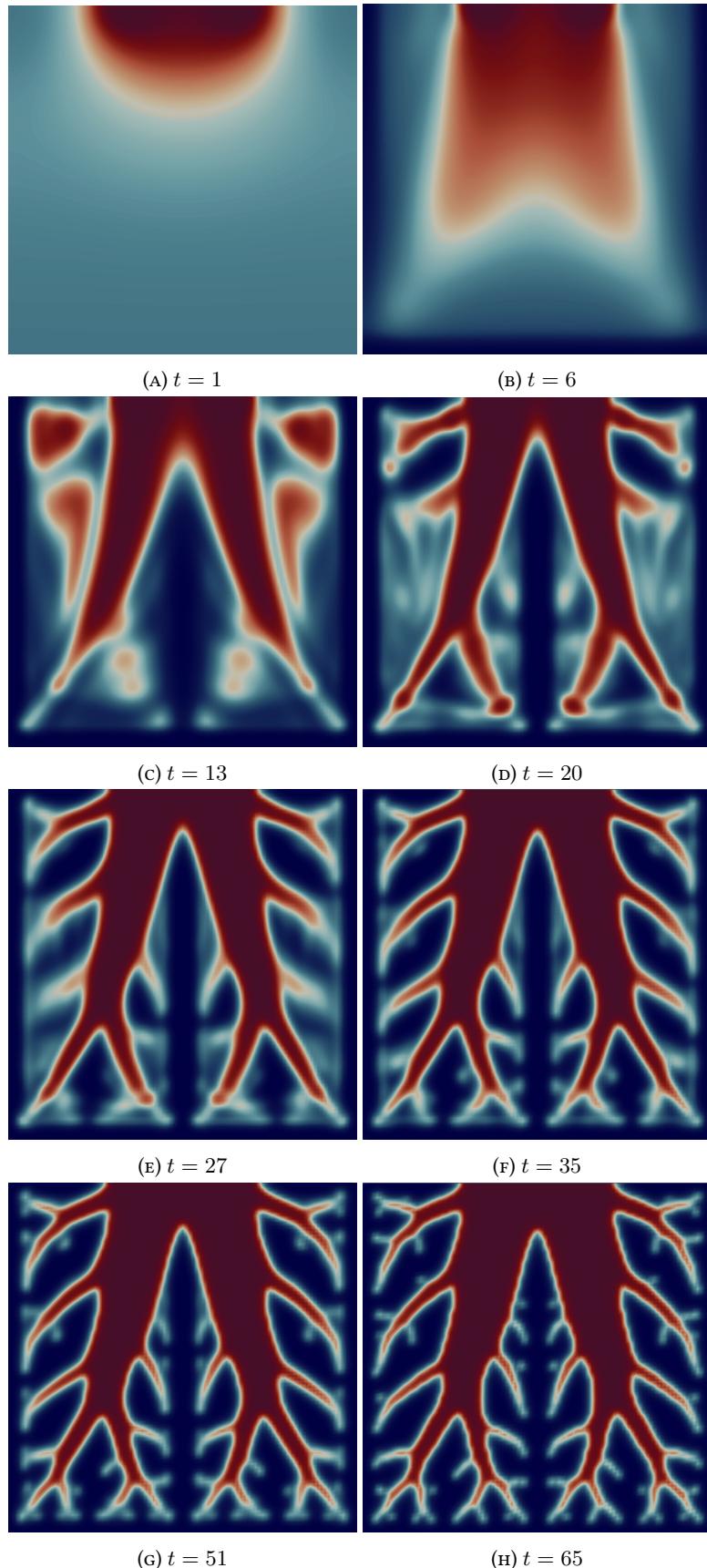


FIGURE 9. Unit Square with Single Sink: State Solution

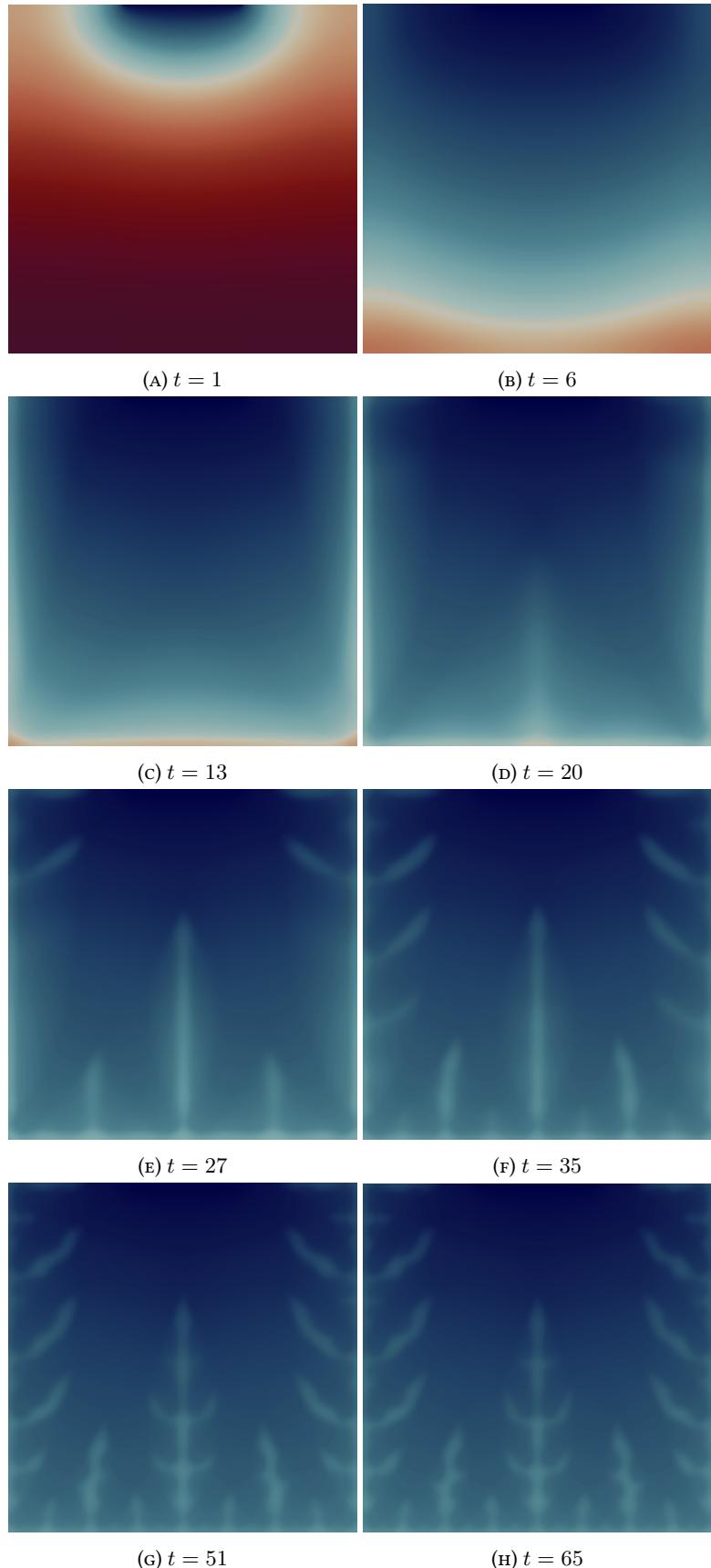


FIGURE 10. Unit Square with Two Sinks: Filtered Distribution

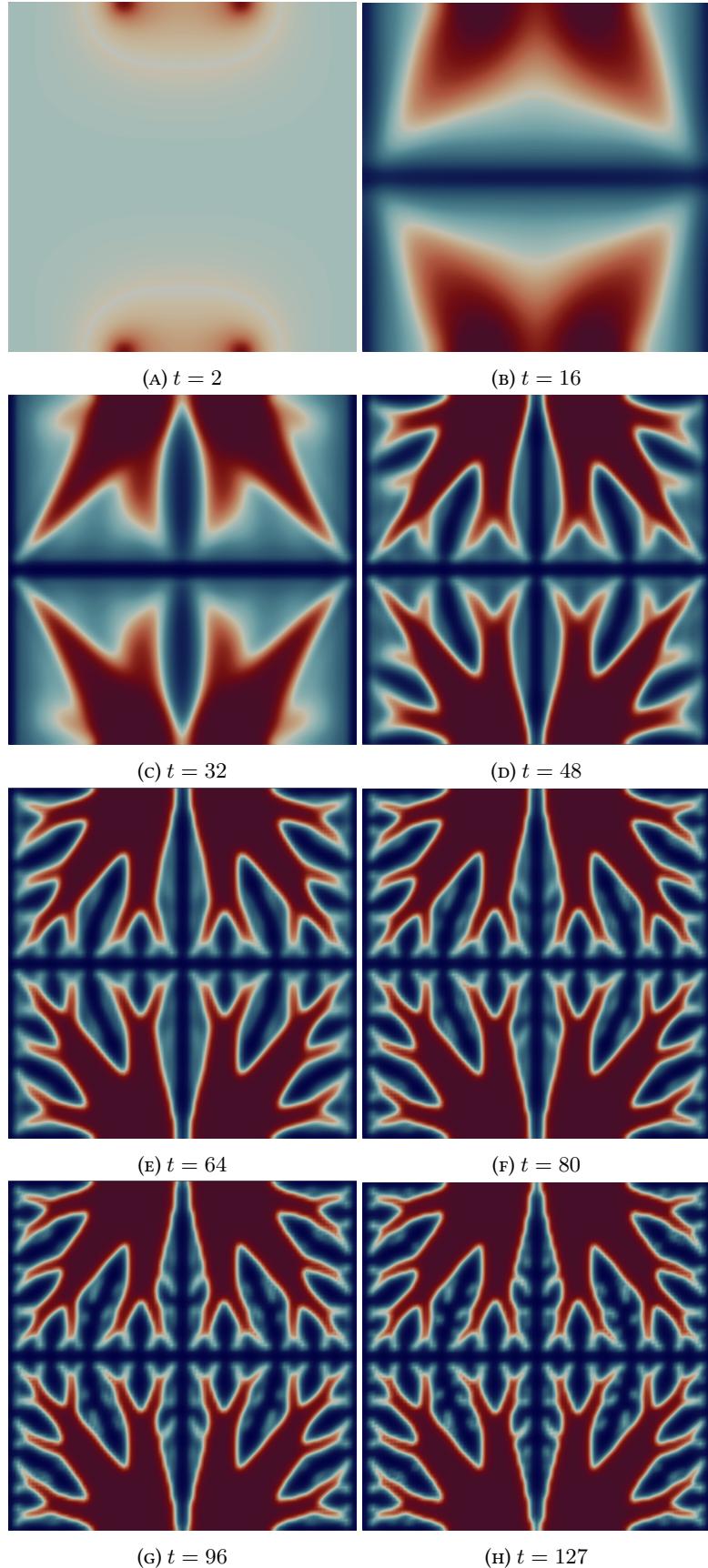
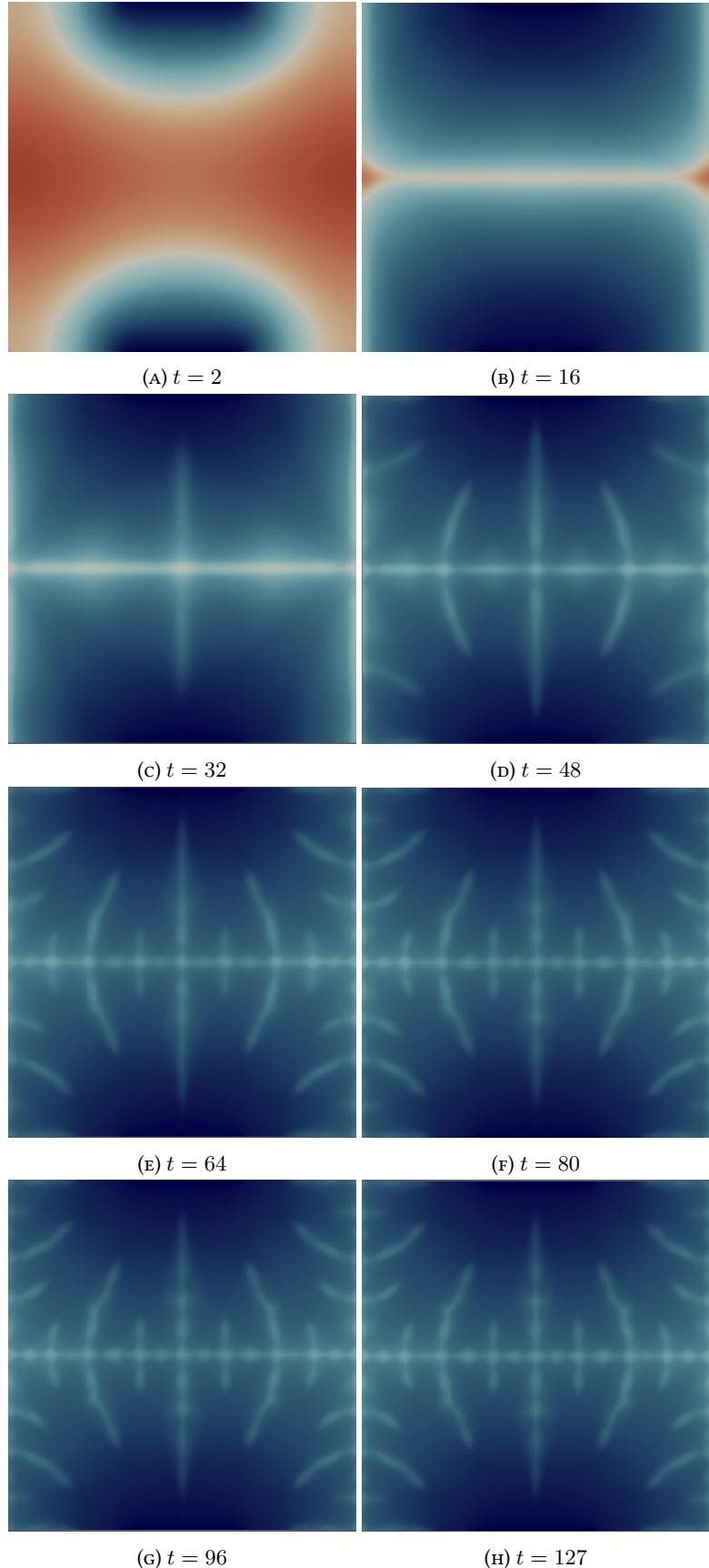
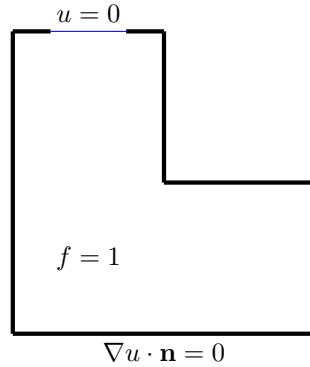


FIGURE 11. Unit Square with Two Sinks Simulation: State Solution



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.

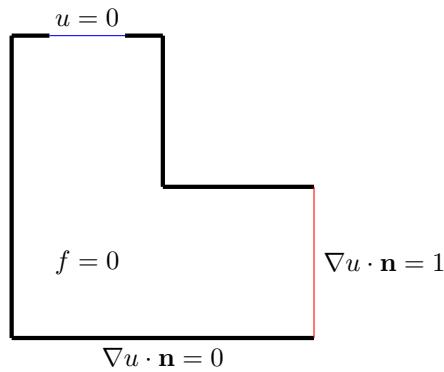
FIGURE 12. *L*-shaped Ω with a sink.



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 $\alpha = 0.001$.

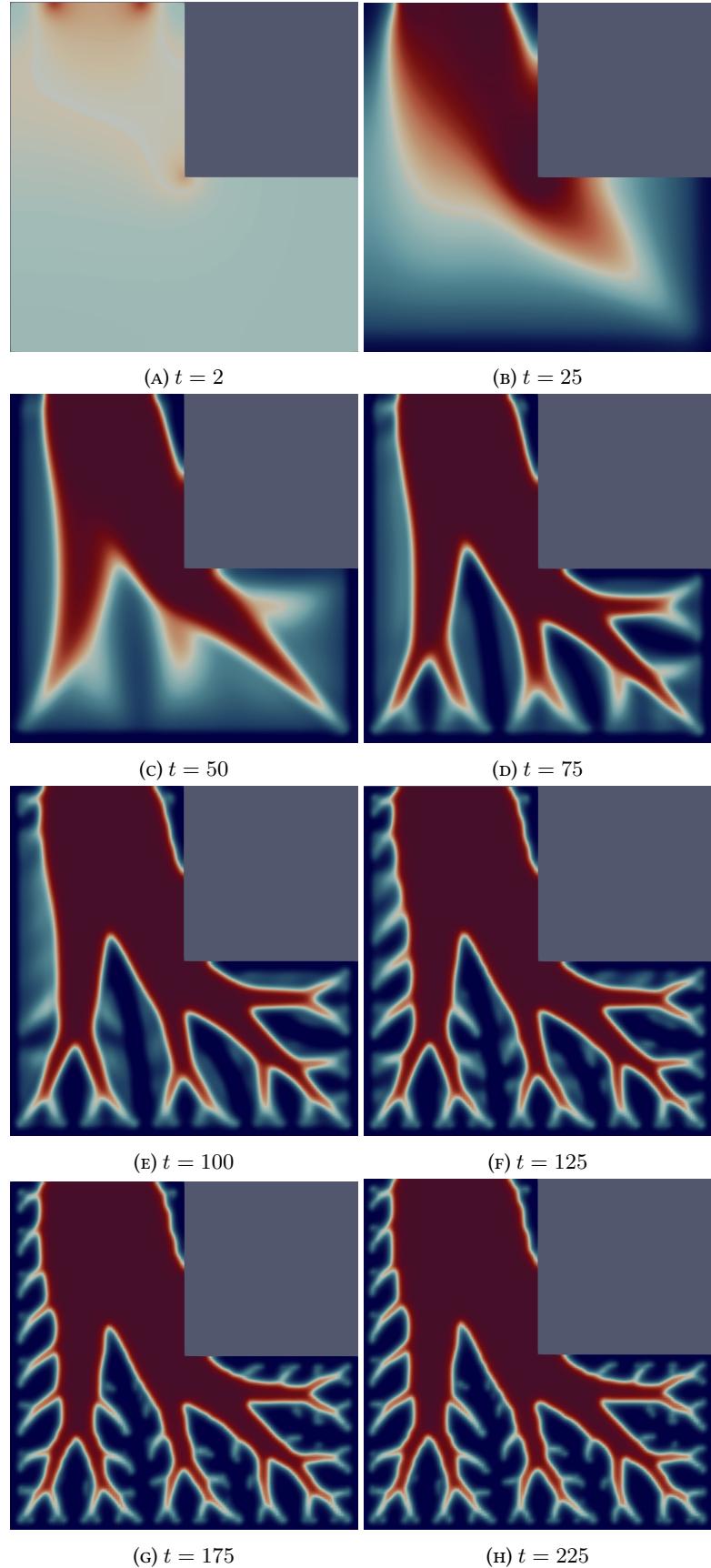
FIGURE 14. *L*-shape with a Sink: Filtered Distribution

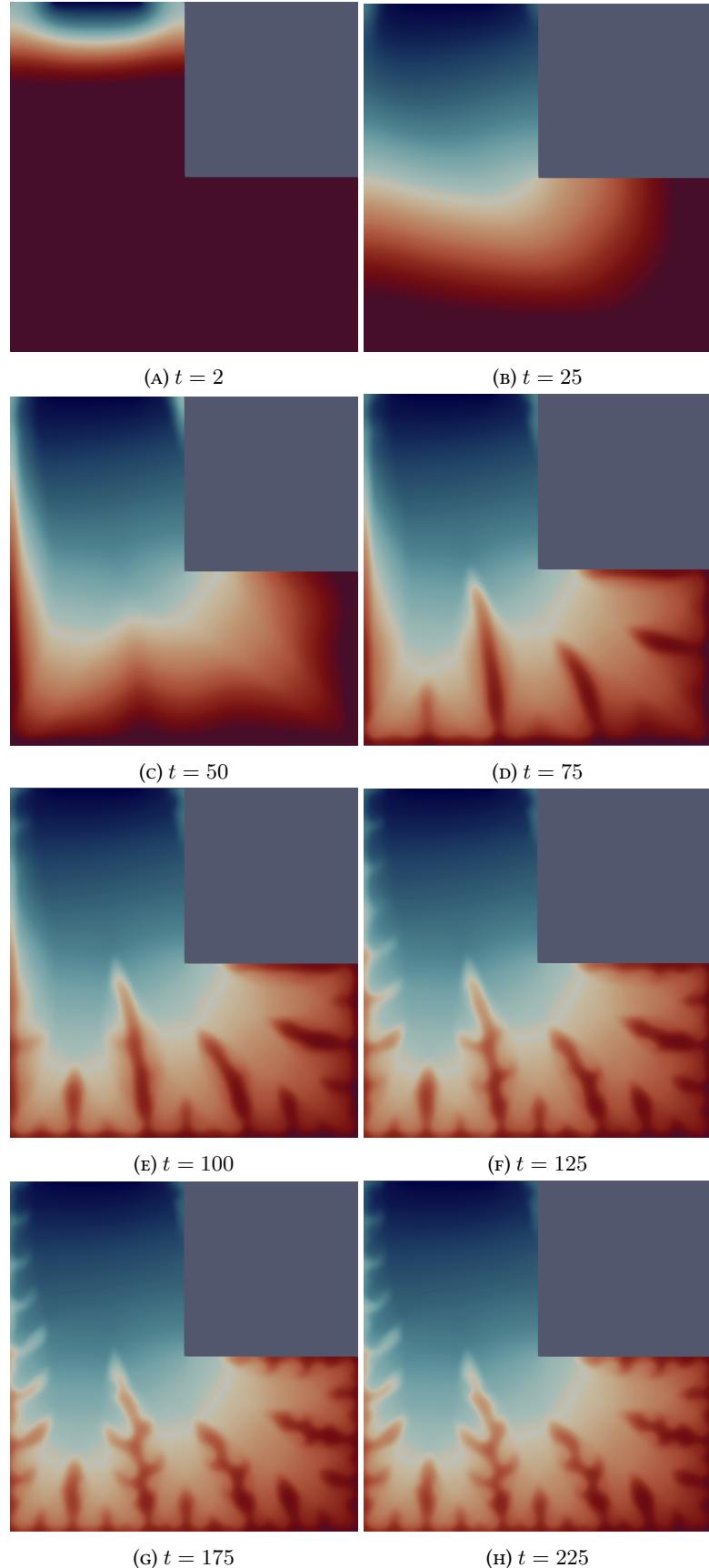
FIGURE 15. *L*-shape with a Sink: State Solution

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

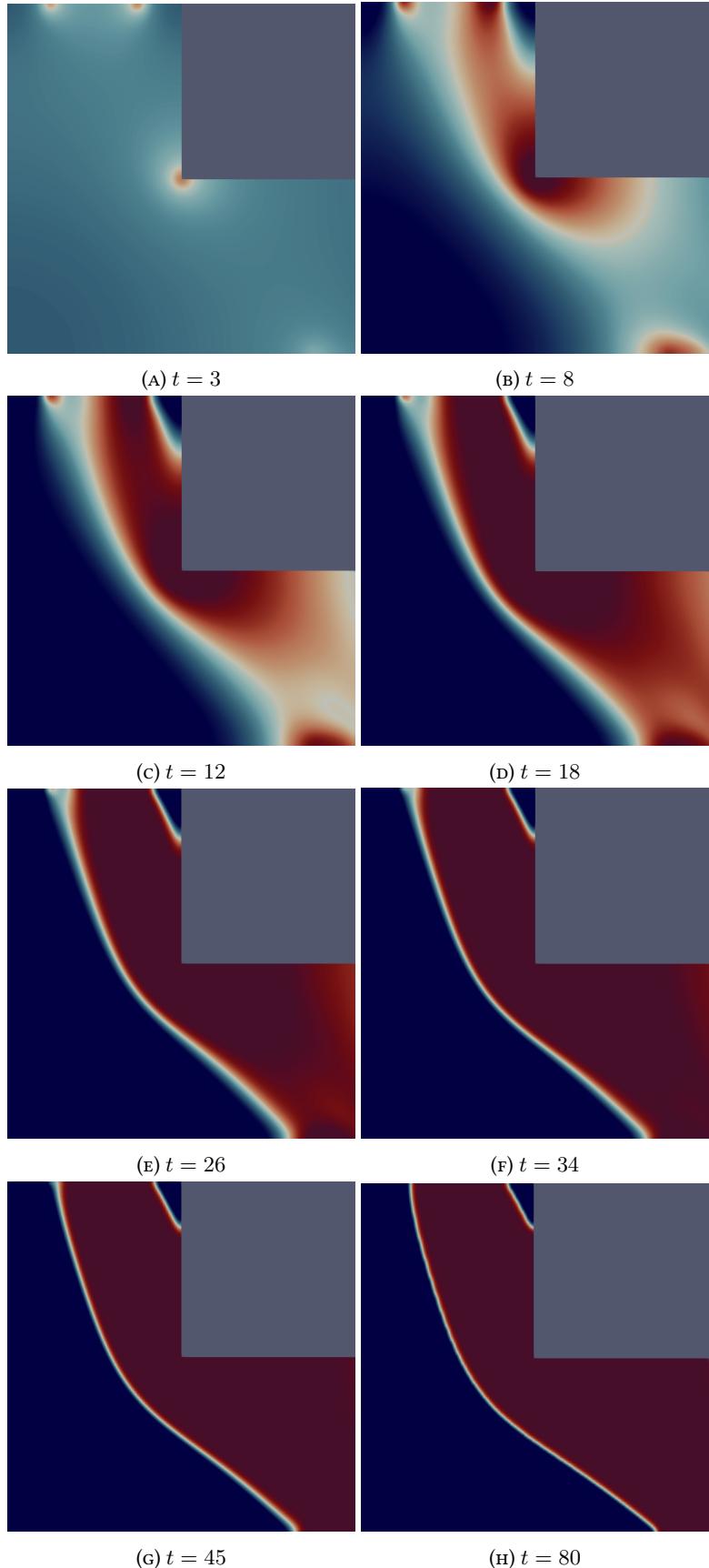
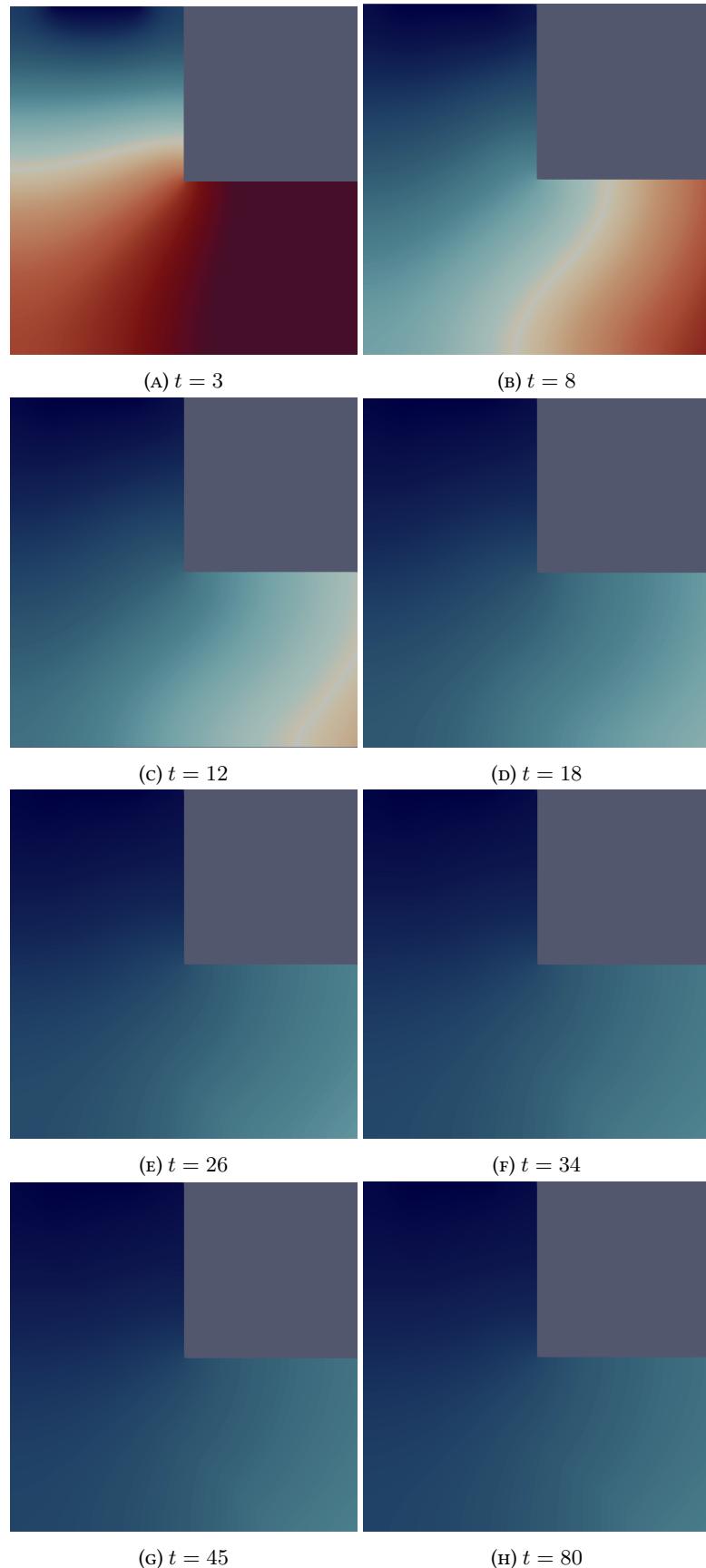


FIGURE 17. *L*-shape with a Sink and Source: State Solution

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.

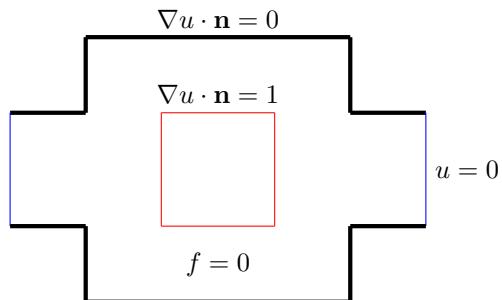


FIGURE 19. Heat sink problem: Filtered Distribution

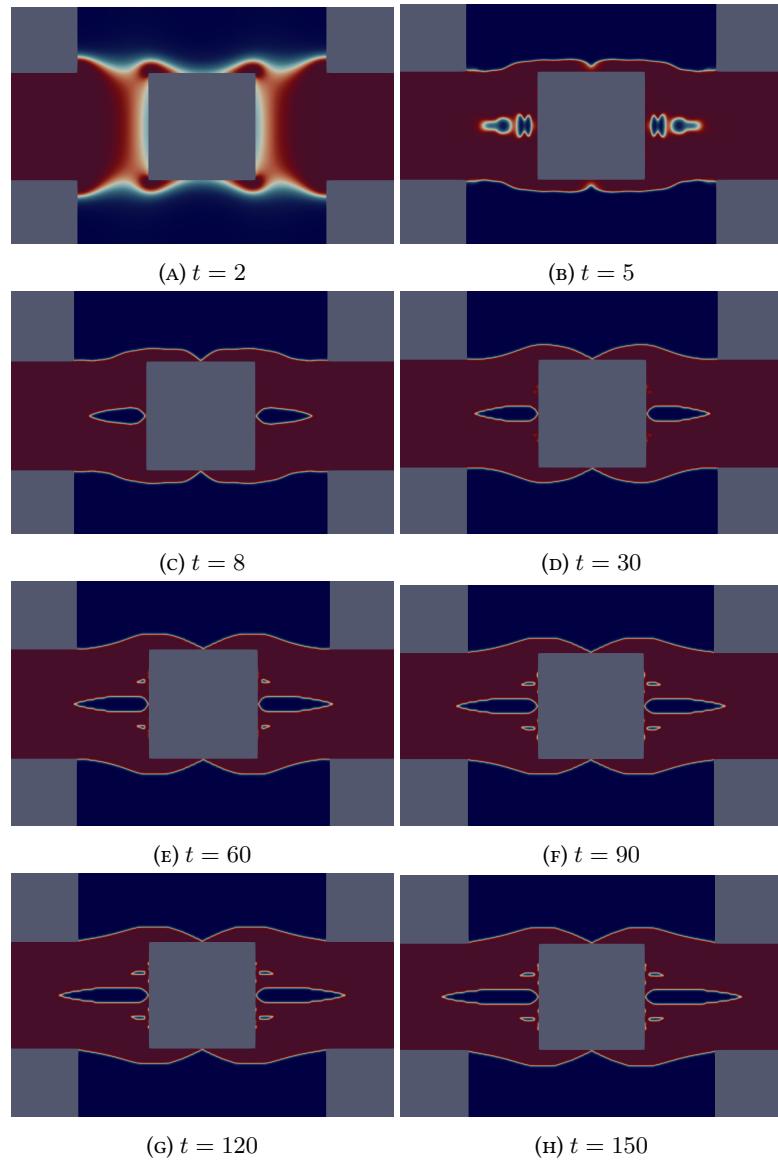
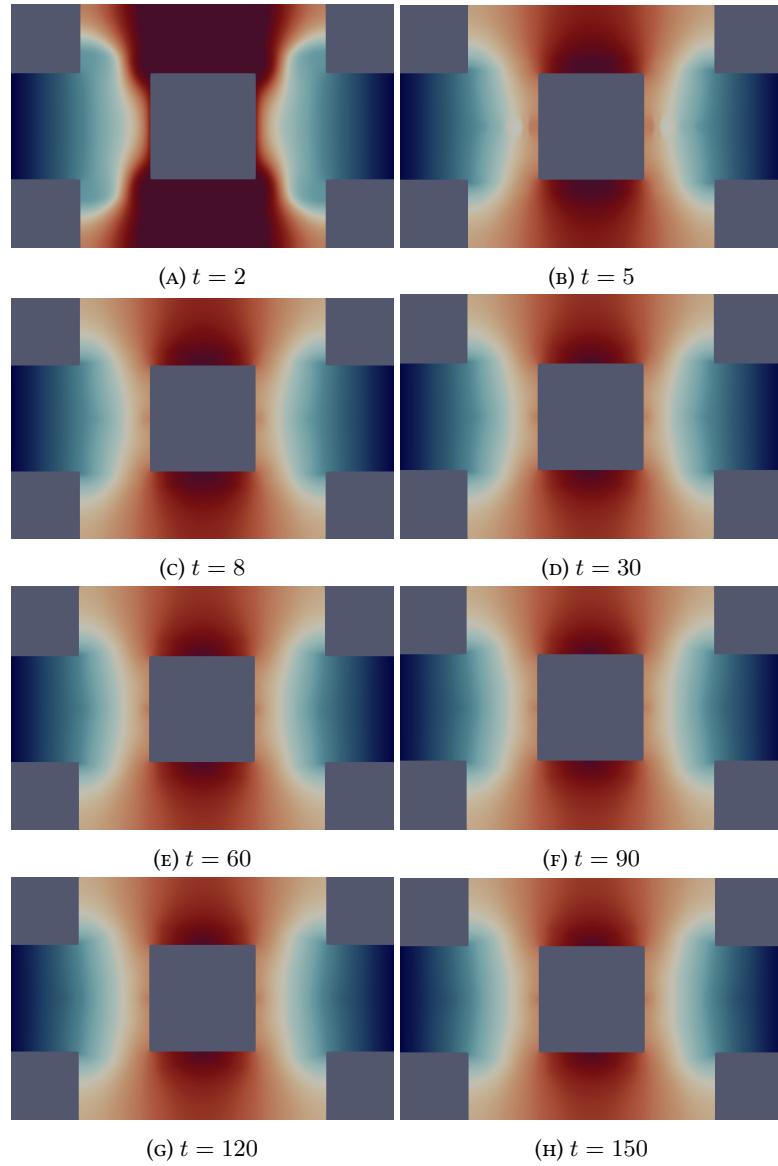


FIGURE 20. Heat sink problem: State Solution



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 embarrassingly 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](https://doi.org/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](https://doi.org/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](https://doi.org/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. We are interested in the distribution of two material phases with isotropic conductivities of 1 and 0.001 for material and void, respectively.

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.

```

 $x \leftarrow \theta \cdot \mathbf{1}$                                      ▷ Initial guess is a matrix of  $\theta$ 's
while not converged do
     $x_{\text{old}} \leftarrow x$ 
     $U \leftarrow \text{FE};$  Displacement vector.                  ▷ State solution.
     $KE \leftarrow \text{Element Stiffness Matrix}$ 
     $c \leftarrow \text{objective function calculation}$ 
     $dc \leftarrow \text{check}$                                      ▷ Sensitivity filter.
     $x \leftarrow \text{OC}$                                          ▷ Design update step (by the optimality criteria method).
end while

```

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

The convergence condition depends on the largest value in the difference $x - x_{\text{old}}$ or the maximum number of allowed iterations. Here, the only calculation that requires motivation is that of the objective function (the large for loop). We have seen that this is $U^T K U = \sum E_e(x_e) u_e^T k_0 u_e$. Due to the symmetry between all the square elements, for solid material, k_0 (the element stiffness matrix) is identical,⁶ so we need only construct it once and can maintain it as e.g. a const.

$$(25) \quad KE = \begin{pmatrix} k_{11} & k_{12} & k_{13} & k_{14} \\ k_{21} & k_{22} & k_{23} & k_{24} \\ k_{31} & k_{32} & k_{33} & k_{34} \\ k_{41} & k_{42} & k_{43} & k_{44} \end{pmatrix} = \begin{pmatrix} 2/3 & -1/6 & -1/3 & -1/6 \\ -1/6 & 2/3 & -1/6 & -1/6 \\ -1/3 & -1/6 & 2/3 & -1/6 \\ -1/6 & -1/3 & -1/6 & 2/3 \end{pmatrix}.$$

We calculate also the sensitivities $\frac{\partial c}{\partial x_e}$, the formula for which we have essentially seen; in particular, notice that the derivatives are always negative, meaning that increasing the material reduces the thermal compliance of the structure.

$$(26) \quad \frac{\partial c}{\partial x_e} = -0.999 \cdot p \cdot x_e^{p-1} u^T k_0 u$$

U_e denotes the element displacement; obtaining this is a matter of vertex indexation once we have the global displacement vector U . The elements are numbered column-wise from left to right. This constructed sensitivity matrix is passed into the sensitivity filter subroutine as well as the relevant parameters and current material distribution.

A.2. Finite Element Analysis. We form the global stiffness matrix K with a loop over all elements. Then, the load is set to all 0.01, meaning that entire domain is uniformly heated. Finally, the matrix problem $Ku = f$ is solved and the solution U is returned.

A.3. OC based optimizer. Here, we follow the algorithm outlined in the discussion of top88; nothing changes in this case with the physics hot-swap. The primary loop is the bisectioning algorithm for finding the Lagrange multiplier, and while we wait for the bisectioning algorithm, at each step, we update the distribution in x_{new} , which is finally returned upon convergence.

A.4. Mesh-independency filtering. This is an implementation of the sensitivity filter, which was skipped for the sake of brevity. The scheme modifies the element sensitivities of the compliance with [BS03]

$$(27) \quad \frac{\hat{\partial}f}{\partial \rho_k} = \frac{1}{\rho_k \sum \hat{H}_i} \sum \hat{H}_i \rho_i \frac{f}{\rho_i},$$

where the weight factor \hat{H}_i is

$$(28) \quad \hat{H}_i = r_{\min} - \text{dist}(k, i), \quad \{i \in N | \text{dist}(k, i) \leq r_{\min}\}, k = 1, \dots, N,$$

where N is the number of elements in the mesh.

⁶The MATLAB code keeps this as a subroutine that returns a constant value, but this is inefficient in general for several reasons, especially for a JIT language like Julia.

APPENDIX B. JULIA CODE LISTINGS

The following are Julia versions of top88 and toph.

top88.jl

```

1 module Top88
2
3 using LinearAlgebra
4 using SparseArrays
5 using Statistics
6
7 export top88
8 export prepare_filter
9 export OC
10
11 """
12     top88(nelx, nely, volfrac, penal, rmin, ft)
13
14 A direct, naive Julia port of Andreassen et al. "Efficient topology optimization in MATLAB
15 using 88 lines of code." By default, this will reproduce the optimized MBB beam from Sigmund
16 (2001).
17
18 # Arguments
19 - 'nelx::S': Number of elements in the horizontal direction
20 - 'nely::S': Number of elements in the vertical direction
21 - 'volfrac::T': Prescribed volume fraction
22 - 'penal::T': The penalization power
23 - 'rmin::T': Filter radius divided by the element size
24 - 'ft::Bool': Choose between sensitivity (if true) or density filter (if false). Defaults
25     to sensitivity filter.
26 - 'write::Bool': If true, will write out iteration number, changes, and density for each
27     iteration. Defaults for false.
28 - 'loop_max::Int': Explicitly set the maximum number of iterations. Defaults to 1000.
29
30 # Returns
31 - 'Matrix{T}': Final material distribution, represented as a matrix
32 """
33 function top88(
34     nelx::S=60,
35     nely::S=20,
36     volfrac::T=0.5,
37     penal::T=3.0,
38     rmin::T=2.0,
39     ft::Bool=true,
40     write::Bool=false,
41     loop_max::Int=1000
42 ) where {S <: Integer, T <: AbstractFloat}
43     # Physical parameters
44     E0 = 1; Emin = 1e-9; nu = 0.3;
45
46     # Prepare finite element analysis
47     A11 = [12 3 -6 -3; 3 12 3 0; -6 3 12 -3; -3 0 -3 12]
48     A12 = [-6 -3 0 3; -3 -6 -3 -6; 0 -3 -6 3; 3 -6 3 -6]
49     B11 = [-4 3 -2 9; 3 -4 -9 4; -2 -9 -4 -3; 9 4 -3 -4]
50     B12 = [ 2 -3 4 -9; -3 2 9 -2; 4 9 2 3; -9 -2 3 2]
51     KE = 1/(1-nu^2)/24*([A11 A12; A12' A11]+nu*[B11 B12; B12' B11])
52
53     nodens = reshape(1:(1+nelx)*(1+nely),1+nely,1+nelx)
54     edofVec = reshape(2*nodenrs[1:end-1,1:end-1].+1, nelx*nely, 1)
55     edofMat = zeros(Int64, nelx*nely, 8)
56
57     offsets = [0 1 2*nely.+[2 3 0 1] -2 -1]
58     for i = 1:8
59         for j = 1:nelx*nely
60             edofMat[j,i]= edofVec[j] + offsets[i]
61         end
62     end
63
64     iK = reshape(kron(edofMat,ones(8,1))', 64*nelx*nely,1)
65     jK = reshape(kron(edofMat,ones(1,8))', 64*nelx*nely,1)
66

```

```

67      # Loads and supports
68      F = spzeros(2*(nely+1)*(nelx+1))
69      F[2,1] = -1
70      U = spzeros(2*(nely+1)*(nelx+1))
71
72      fixeddofs = union(1:2:2*(nely+1), [2*(nelx+1)*(nely+1)])
73      alldofs = 1:2*(nely+1)*(nelx+1)
74      freedofs = setdiff(alldofs, fixeddofs)
75
76      # Prepare the filter
77      H, Hs = prepare_filter(nelx, nely, rmin)
78
79      # Initialize iteration
80      x = volfrac*ones(nely,nelx)
81      xPhys = x
82      loop = 0
83      change = 1
84      cValues = []
85
86      # Start iteration
87      while change > 0.01
88          loop += 1
89          # FE-Analysis
90          sK = [j*((i+Emin)^penal) for i in ((E0-Emin)*xPhys[:])' for j in KE[:]]
91          K = sparse(iK[:,], jK[:,], sK)
92          K = (K+K')/2
93
94          KK = cholksy(K[freedofs, freedofs])
95          U[freedofs] = KK \ F[freedofs]
96
97          # OLD: edM = [convert(Int64,i) for i in edofMat]
98          mat = (U[edofMat]*KE).*U[edofMat]
99
100         # Objective function and sensitivity analysis
101         ce = reshape([sum(mat[i,:]) for i = 1:(size(mat)[1])],nely,nelx)
102         c = sum((Emin*ones(size(xPhys)).+(xPhys.^penal)*(E0-Emin)).*ce)
103         push!(cValues,c)
104         dc = -penal*(E0-Emin)*xPhys.^((penal-1).*ce)
105         dv = ones(nely,nelx)
106
107         # Filtering/ modification of sensitivities
108         if ft
109             dc[:] = H*(x[:].*dc[:])./Hs./max(1e-3,maximum(x[:]))
110         else
111             dc[:] = H*(dc[:]./Hs)
112             dv[:] = H*(dv[:]./Hs)
113         end
114
115         # Optimality criteria update of design variables and physical densities
116         xnew = OC(nelx, nely, x, volfrac, dc, dv, xPhys, ft)
117
118         change = maximum(abs.(x-xnew))
119         x = xnew
120
121         write && println("Loop = ", loop, ", Change = ", change ,", c = ", c, ", structural density = ", mean(x))
122         loop >= loop_max && break
123     end
124
125     return x
126 end
127
128 """
129 """
130 Prepare sensitivity/ density filter
131 """
132 function prepare_filter(nelx::S, nely::S, rmin::T) where {S <: Integer, T <: AbstractFloat}
133     iH = ones(nelx*nely*(2*(convert(Int64,ceil(rmin)-1))+1)^2)
134     jH = ones(size(iH))
135     sH = zeros(size(iH))
136     k = 0
137     for i1 = 1:nelx
138         for j1 = 1:nely

```

```

139         e1 = (i1-1)*nely+j1
140         for i2 = max(i1-(ceil(rmin)-1),1):min(i1+(ceil(rmin)-1),nelx)
141             for j2 = max(j1-(ceil(rmin)-1),1):min(j1+(ceil(rmin)-1),nely)
142                 e2 = (i2-1)*nely+j2
143                 k += 1
144                 iH[k] = e1
145                 jH[k] = e2
146                 sH[k] = max(0,rmin-sqrt((i1-i2)^2+(j1-j2)^2))
147             end
148         end
149     end
150 H = sparse(iH,jH,sH)
152 Hs = [sum(H[i,:]) for i = 1:(size(H)[1])]
153
154     return H, Hs
155 end
156 """
157 """
158 Optimality criteria update
159 """
160 function OC(
161     nelx::S,
162     nely,
163     x,
164     volfrac,
165     dc::Matrix{T},
166     dv,
167     xPhys::Matrix{T},
168     ft::Bool
169 ) where {S <: Integer, T <: AbstractFloat}
170     l1 = 0; l2 = 1e9; move = 0.2
171     xnew = zeros(nely, nelx)
172
173     while (l2-l1)/(l1+l2) > 1e-3
174         lmid = 0.5*(l2+l1)
175         RacBe = sqrt.(dc./dv/lmid)
176         XB = x.*RacBe
177
178         for i = 1:nelx
179             for j = 1:nely
180                 xji = x[j,i]
181                 xnew[j,i] = max(0.000,max(xji-move,min(1,min(xji+move,XB[j,i]))))
182             end
183         end
184
185         if ft
186             xPhys = xnew
187         else
188             xPhys[:] = (H*xnew[:])./Hs
189         end
190
191         if sum(xPhys[:]) > volfrac*nelx*nely
192             l1 = lmid
193         else
194             l2 = lmid
195         end
196     end
197
198     return xnew
199 end
200
201 end

```

toph.jl

```

1 module TopH
2
3 using LinearAlgebra
4 using SparseArrays
5 using Statistics
6

```

```

7 export toph
8 export OC
9 export check
10 export FE
11 """
12 """
13     toph(nelx, nely, volfrac, penal, rmin, write, loop_max)
14
15 A direct, naive Julia port of the 'toph' code listing from "Topology Optimization"
16 by Martin Bendsoe and Ole Sigmund.
17
18 # Arguments
19 - 'nelx::S': Number of elements in the horizontal direction
20 - 'nely::S': Number of elements in the vertical direction
21 - 'volfrac::T': Prescribed volume fraction
22 - 'penal::T': The penalization power
23 - 'rmin::T': Filter radius divided by the element size
24 - 'write::Bool': If true, will write out iteration number, changes, and density
25     for each iteration. Defaults to false.
26 - 'loop_max::Int': Explicitly set the maximum number of iterations. Defaults to 1000.
27
28 # Returns
29 - 'Matrix{T}': Final material distribution, represented as a matrix.
30 """
31 function toph(
32     nelx::S,
33     nely::S,
34     volfrac::T,
35     penal::T,
36     rmin::T,
37     write::Bool=false,
38     loop_max::Int=100
39 ) where {S <: Integer, T <: AbstractFloat}
40     # Initialization
41     x = volfrac * ones(nely,nelx)
42     loop = 0
43     change = 1.
44     dc = zeros(nely,nelx)
45
46     # Start iteration
47     while change > 0.01
48         loop += 1
49         xold = x
50         c = 0.
51
52         # FE Analysis
53         U = FE(nelx,nely,x,penal)
54
55         KE = [ 2/3 -1/6 -1/3 -1/6
56                -1/6 2/3 -1/6 -1/3
57                -1/3 -1/6 2/3 -1/6
58                -1/6 -1/3 -1/6 2/3 ]
59
60         # Objective function/ sensitivity analysis
61         for ely = 1:nely
62             for elx = 1:nelx
63                 n1 = (nely+1)*(elx-1)+ely
64                 n2 = (nely+1)* elx +ely
65                 Ue = U[[n1; n2; n2+1; n1+1]]
66
67                 c += (0.001+0.999*x[ely,elx]^penal)*Ue'*KE*Ue
68                 dc[ely,elx] = -0.999*penal*(x[ely,elx])^(penal-1)*Ue'*KE*Ue
69             end
70         end
71
72         # Sensitivity filtering
73         dc = check(nelx,nely,rmin,x,dc)
74         # Design update by optimality criteria method
75         x = OC(nelx,nely,x,volfrac,dc)
76
77         # Print out results if desired
78         if write

```

```

79         change = maximum(abs.(x-xold))
80         println("Change = ", change, " c = ", c)
81     end
82
83     loop >= loop_max && break
84 end
85
86     return x
87 end
88 """
89     OC(nelx, nely, x, volfrac, dc)
90
91 Optimality criteria update
92
93 # Arguments
94 - 'nelx::S': Number of elements in the horizontal direction
95 - 'nely::S': Number of elements in the vertical direction
96 - 'x::Matrix{T}': Current material distribution
97 - 'volfrac::T': Prescribed volume fraction
98 - 'dc::Matrix{T}': Sensitivity filter
99
100 # Returns
101 - 'Matrix{T}': Updated material distribution
102
103 """
104 """
105 function OC(
106     nelx::S,
107     nely::S,
108     x::Matrix{T},
109     volfrac::T,
110     dc::Matrix{T}
111 ) where {S <: Integer, T <: AbstractFloat}
112     l1 = 0; l2 = 100000; move = 0.2
113     xnew = zeros(nely,nelx)
114
115     while (l2-l1) > 1e-4
116         lmid = 0.5*(l2+l1)
117         RacBe = sqrt.(-dc/lmid)
118         XB = x .* RacBe
119
120         for i = 1:nelx
121             for j = 1:nely
122                 xji = x[j,i]
123                 xnew[j,i] = max(0.001,max(xji-move,min(1,min(xji+move,XB[j,i]))))
124             end
125         end
126
127         if (sum(sum(xnew)) - volfrac*nelx*nely) > 0
128             l1 = lmid
129         else
130             l2 = lmid
131         end
132     end
133
134     return xnew
135 end
136 """
137 """
138     check(nelx, nely, rmin, x, dc)
139
140 Mesh independency filter
141
142 # Arguments
143 - 'nelx::S': Number of elements in the horizontal direction
144 - 'nely::S': Number of elements in the vertical direction
145 - 'rmin::T': Sensitivity filter radius divided by element size
146 - 'x::Matrix{T}': Current material distribution
147 - 'dc::Matrix{T}': Compliance derivatives
148
149 # Returns
150 - 'Matrix{T}': Updated dc

```

```

151 """
152 function check(nelx::S,
153     nely::S,
154     rmin::T,
155     x::Matrix{T},
156     dc::Matrix{T}
157 ) where {S <: Integer, T <: AbstractFloat}
158     dcn=zeros(nely,nelx)
159
160     for i = 1:nelx
161         for j = 1:nely
162             sum=0.0
163
164             for k = max(i-floor(rmin),1):min(i+floor(rmin),nelx)
165                 for l = max(j-floor(rmin),1):min(j+floor(rmin),nely)
166                     l = Int64(l); k = Int64(k)
167                     fac = rmin-sqrt((i-k)^2+(j-l)^2)
168                     sum = sum+max(0,fac)
169                     dcn[j,i] += max(0,fac)*x[l,k]*dc[l,k]
170                 end
171             end
172             dcn[j,i] = dcn[j,i]/(x[j,i]*sum)
173         end
174     end
175
176     return dcn
177 end
178 """
180 """
181     FE(nelx, nely, x, penal)
182
183 Finite element implementation
184
185 # Arguments
186 - 'nelx::S': Number of elements in the horizontal direction
187 - 'nely::S': Number of elements in the vertical direction
188 - 'x::Matrix{T}': Current material distribution
189 - 'penal::T': The penalization power
190
191 # Returns
192 - 'Matrix{T}': Differential equation solution U
193 """
194 function FE(
195     nelx::S,
196     nely::S,
197     x::Matrix{T},
198     penal::T
199 ) where {S <: Integer, T <: AbstractFloat}
200     KE = [ 2/3 -1/6 -1/3 -1/6
201             -1/6  2/3 -1/6 -1/3
202             -1/3 -1/6  2/3 -1/6
203             -1/6 -1/3 -1/6  2/3 ]
204
205     K = spzeros((nelx+1)*(nely+1), (nelx+1)*(nely+1))
206     U = zeros((nely+1)*(nelx+1))
207     F = zeros((nely+1)*(nelx+1))
208
209     for elx = 1:nelx
210         for ely = 1:nely
211             n1 = (nely+1)*(elx-1)+ely
212             n2 = (nely+1)* elx +ely
213             edof = [n1; n2; n2+1; n1+1]
214             K[edof,edof] += (0.001+0.999*x[ely,elx]^penal)*KE
215         end
216     end
217
218     F .= 0.01
219     fixeddofs = Int64(nely/2+1-(nely/20)):Int64(nely/2+1+(nely/20))
220     alldofs = 1:(nely+1)*(nelx+1)
221     freedofs = setdiff(alldofs,fixeddofs)
222
223     U[freedofs] = K[freedofs, freedofs] \ F[freedofs]

```

```
223     U[fixeddofs] .= 0
224
225     return U
226 end
227
228 end
```