

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. One typically uses a finite element method to assess the design and a gradient-based optimization method (e.g. the optimality criteria algorithm or a descent algorithm) to discover a solution ρ , though gradient-free methods (e.g. using genetic algorithms) have been studied.

We will first focus on linear elasticity, for historical reasons, and the heuristic development of the foremost methodologies, turning then to a brief exposition of modern methods and implementations of a few heat conduction problems.

CONTENTS

1. Introduction	1
2. Classical Methods and Linear Elasticity	2
3. Thermal Compliance with the Entropic Finite Element Method (EFEM)	2
3.1. Method	2
3.2. Deriving the Lagrangian	2
3.3. The Gradient	3
3.4. Examples	6
References	17
Appendix A. toph.jl Derivation and Details	17
Appendix B. Julia Code Listings	17

2. CLASSICAL METHODS AND LINEAR ELASTICITY

Like the finite element method itself, Topology Optimization found its first motivations in mechanical engineering; for this reason, the classic problem is to construct a stiff structure in the design domain that minimizes the *compliance* (or, strain energy) under the given boundary conditions.

In the ideal, we would be able to

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

We would like to solve the thermal compliance problem

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

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 .

3.1. 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: $\rho = \arg \min \int_{\Omega} f \cdot u \, dx$

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

3.2. 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 2) will be covered here, as they are incorporated into and satisfied automatically as a result of the projection algorithm.

Algorithm 2 $\text{projit}(\psi)$ Nonlinear Projection

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

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

$$\psi \leftarrow \psi + c \quad \triangleright \text{Performed in-place}$$

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

$$(3) \quad \int_{\Omega} -\nabla \cdot (r(\tilde{\rho}) \nabla u) \cdot w = \int_{\Omega} f \cdot w.$$

Now, integrating by parts¹, we expand the left hand side to reach

$$(4) \quad - \int_{\partial\Omega} w(\nabla u) \cdot \mathbf{n} + \int_{\Omega} (r(\tilde{\rho}) \nabla u) \cdot (\nabla w) = \int_{\Omega} f \cdot v.$$

3.2.2. *Weak Form for Helmholtz type Diffusion.* We assume homogeneous Dirichlet boundary conditions, so it is safe to omit the boundary term as in [Equation 4](#).² 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

$$(5) \quad (-\epsilon^2 \Delta \tilde{\rho}, \tilde{w})_{\Omega} + (\tilde{\rho}, \tilde{w})_{\Omega} = (\rho, \tilde{w})_{\Omega}.$$

Integrating by parts similarly, we reach

$$(6) \quad (\epsilon^2 \nabla \tilde{\rho}, \nabla \tilde{w})_{\Omega} + (\tilde{\rho}, \tilde{w})_{\Omega} = (\rho, \tilde{w})_{\Omega}$$

and are done.

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

$$(7) \quad \begin{aligned} \mathcal{L}(u, \rho, \tilde{\rho}, w, \tilde{w}) = & (f, u)_{\Omega} - (r(\tilde{\rho}) \nabla u, \nabla w)_{\Omega} + (\nabla u \cdot \mathbf{n}, w)_{\partial\Omega} \\ & + (f, w)_{\Omega} - (\epsilon^2 \nabla \tilde{\rho}, \nabla \tilde{w})_{\Omega} - (\tilde{\rho}, \tilde{w})_{\Omega} + (\rho, \tilde{w})_{\Omega} \end{aligned}$$

3.3. **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

$$(8) \quad \frac{d}{dt} F(u + t\psi) \Big|_{t=0}.$$

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 $\partial_u F$.

¹Problem set 4; this is a result of the divergence theorem and the product rule for gradients.

²For 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.

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

$$(9) \quad \partial_u(v, w) = 0.$$

Proof. Observe that

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

□

Lemma 3.2 (Simple inner product formula). *Assuming that $u, v \in L^2$,³ It holds that*

$$(10) \quad \partial_u(v, u) = (v, \psi).$$

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

$$(11) \quad \partial_u(u, v) = (\psi, v).$$

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

$$(12) \quad \partial_u(\nabla v, \nabla u) = (\nabla v, \nabla \psi)$$

Moreover,

$$(13) \quad \partial_u(\nabla u, \nabla v) = (\nabla u, \nabla \psi)$$

holds also by symmetry.

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

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

³Noticing our discretization choices from above, this will be sufficient for application.

where the second equality holds by 3.1 and the third by bilinearity and evaluation of the derivative and its restriction (notice t disappears in the derivative). \square

3.3.1. *Filter Equation.* Taking the derivative $\partial_{\tilde{w}} \mathcal{L}$, we will obtain the filter equation. Skipping the application of 3.1, 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 3.3 and 3.2. Setting this equal to zero and moving terms around, solving the filter equation is to find $\tilde{\rho}$ such that

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

3.3.2. *Primal Problem.* Taking the derivative $\partial_w \mathcal{L}$, we will obtain the primal problem (state evaluation). Again skipping the application of 3.1, 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

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

3.3.3. *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.

3.3.4. *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,

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

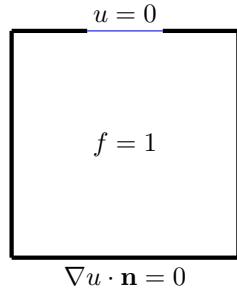
by the equivalence of the primal and dual problems (ergo their solutions; it is from this that we deduce the final equivalence). Setting the derivative equal to zero, we would like to find $\tilde{\rho}$ satisfying

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

3.4. Examples. 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.

3.4.1. Unit Square with Single Sink. We will first implement the problem solved in `toph.jl`, that is, solve [Equation 2](#) 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 1. 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.

3.4.2. 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 2. Ω with two sinks.

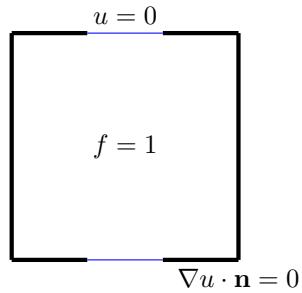


FIGURE 3. Unit Square with Single Sink: Filtered Distribution

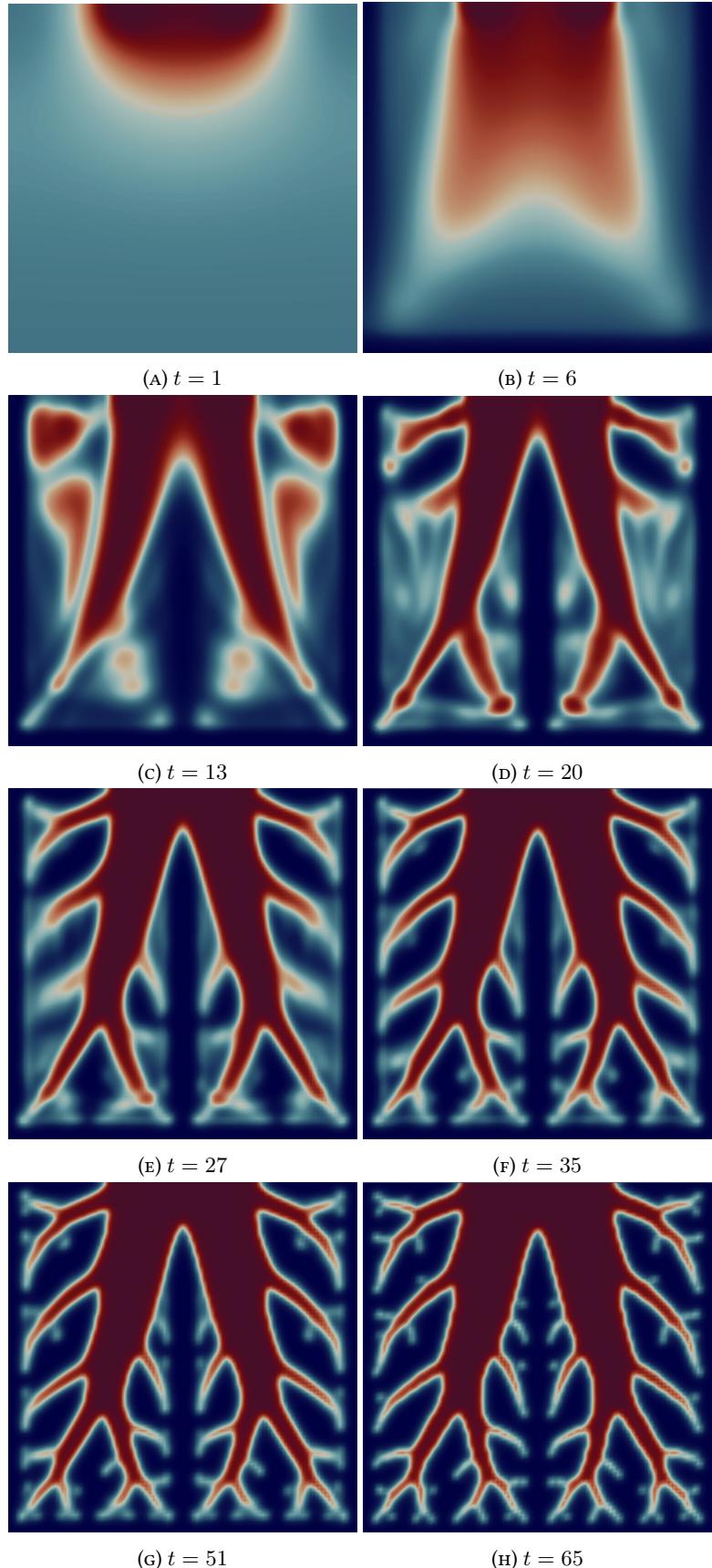


FIGURE 4. Unit Square with Single Sink: State Solution

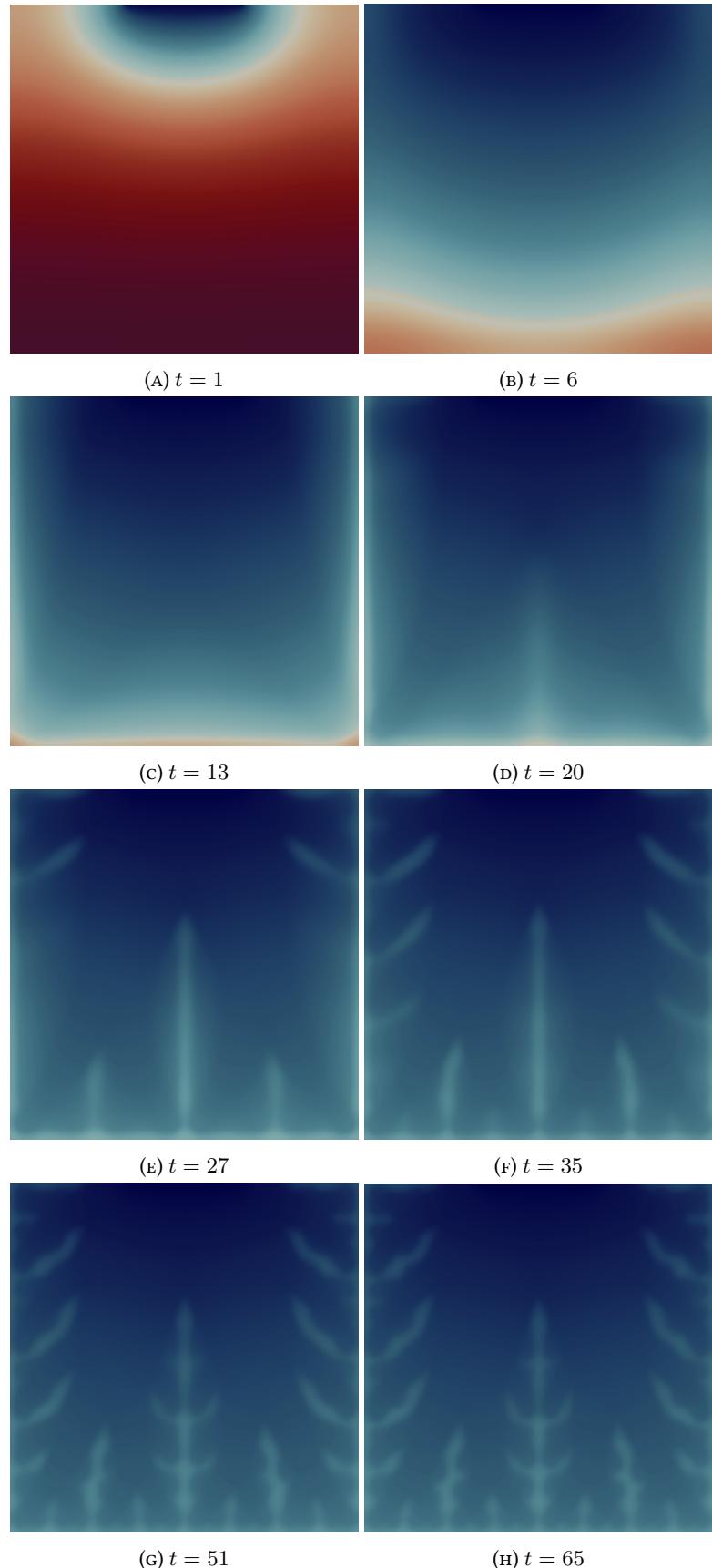


FIGURE 5. Unit Square with Two Sinks: Filtered Distribution

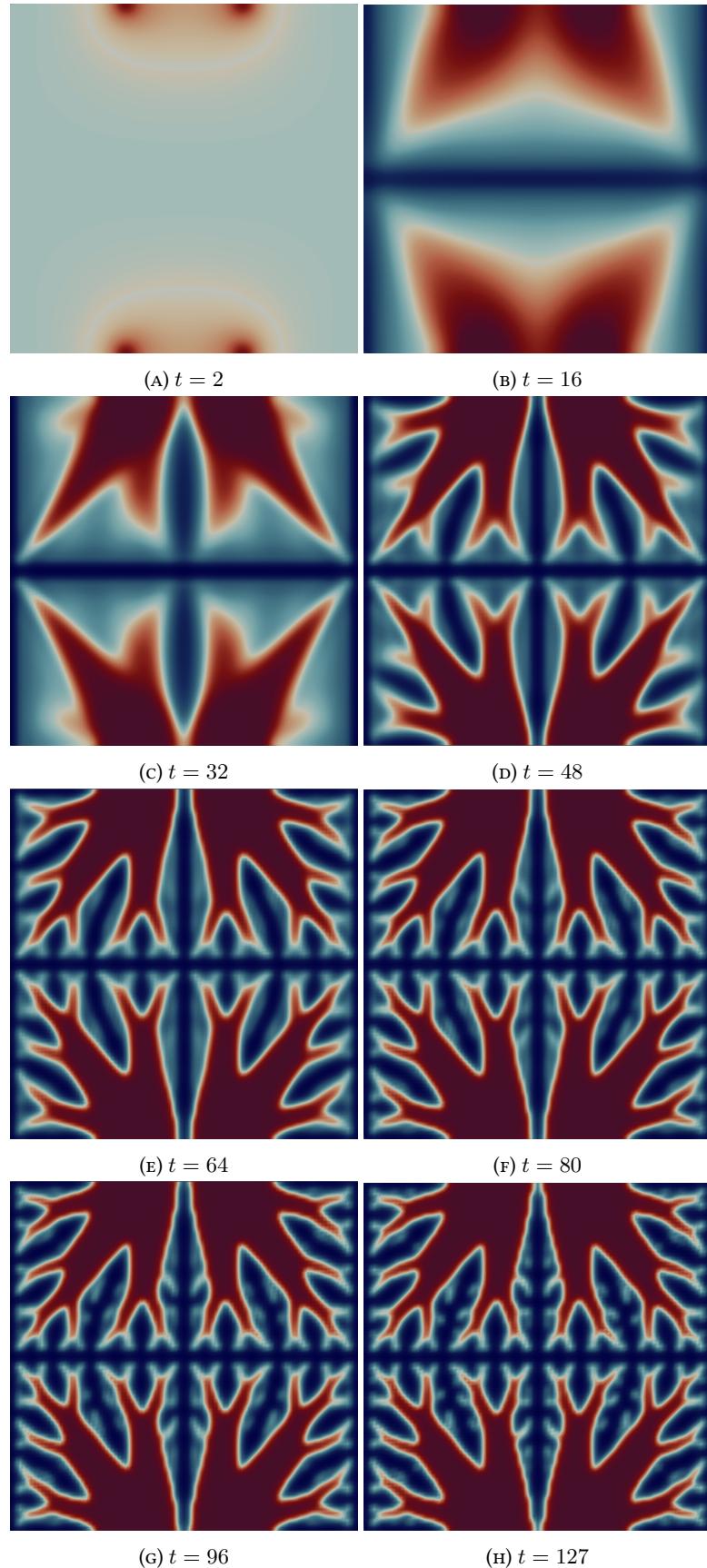
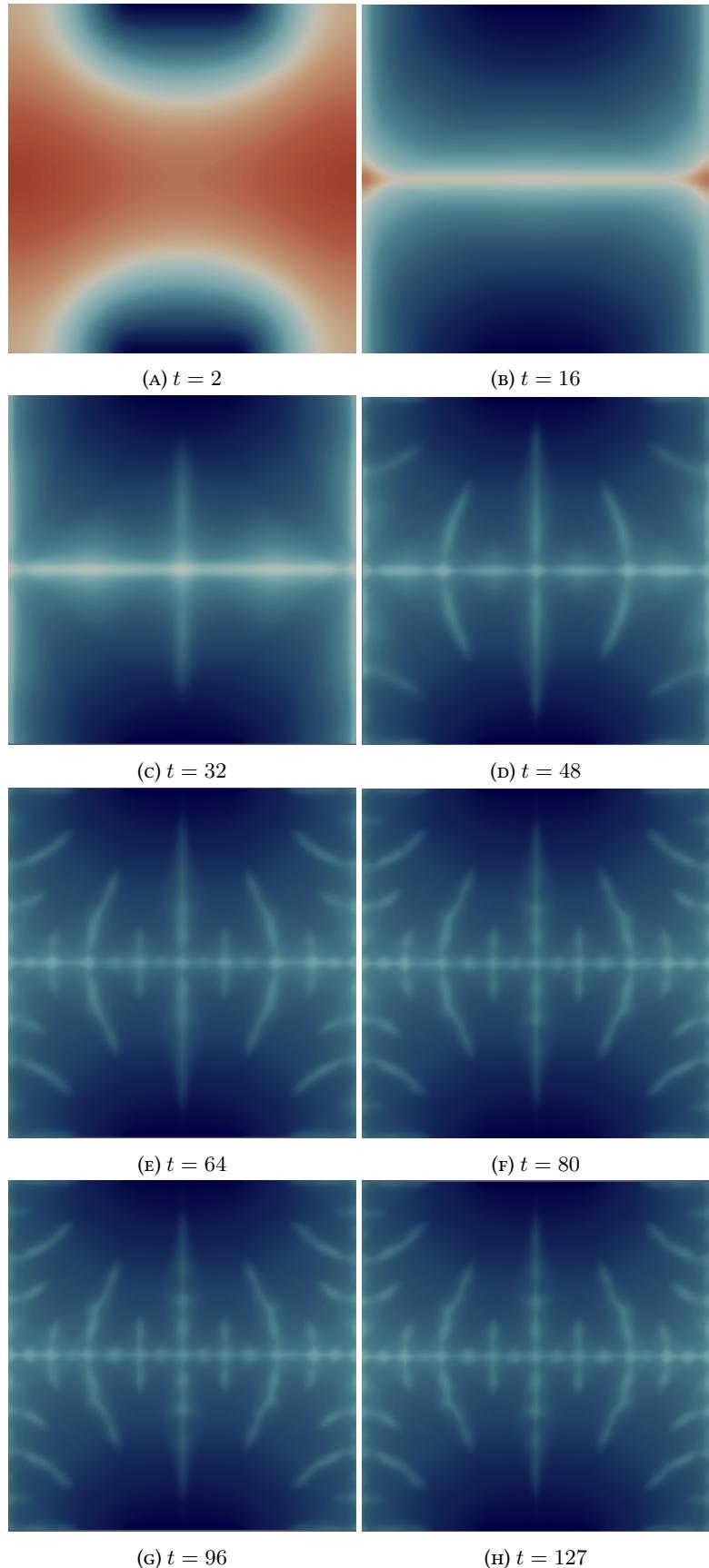
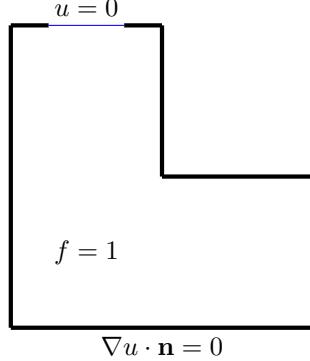


FIGURE 6. Unit Square with Two Sinks Simulation: State Solution



3.4.3. *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.

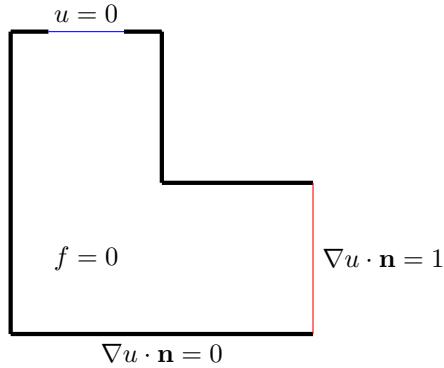
FIGURE 7. *L*-shaped Ω with a sink.



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 $\alpha = 0.001$.

3.4.4. *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 8. *L*-shaped Ω with a sink.



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 $\alpha = 0.001$.

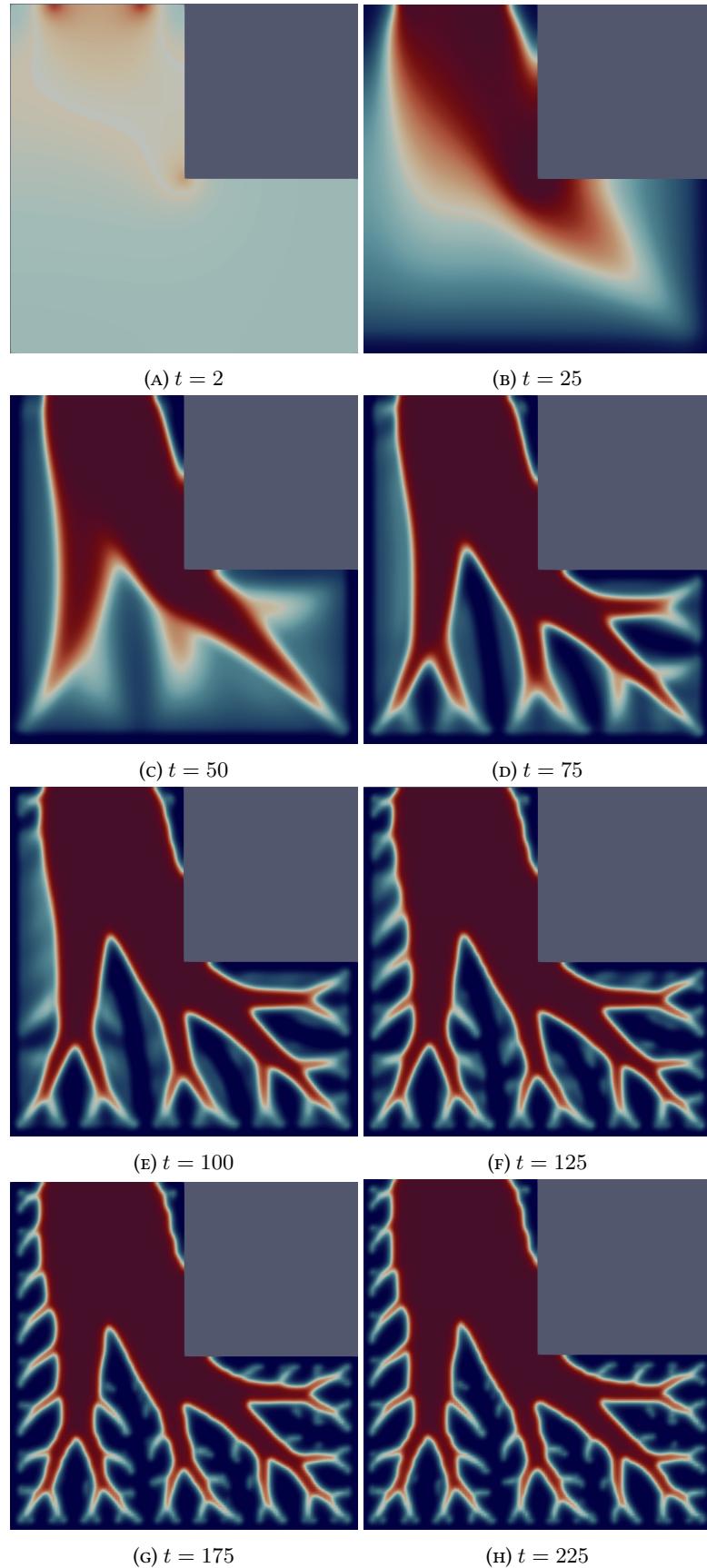
FIGURE 9. *L*-shape with a Sink: Filtered Distribution

FIGURE 10. *L*-shape with a Sink: State Solution

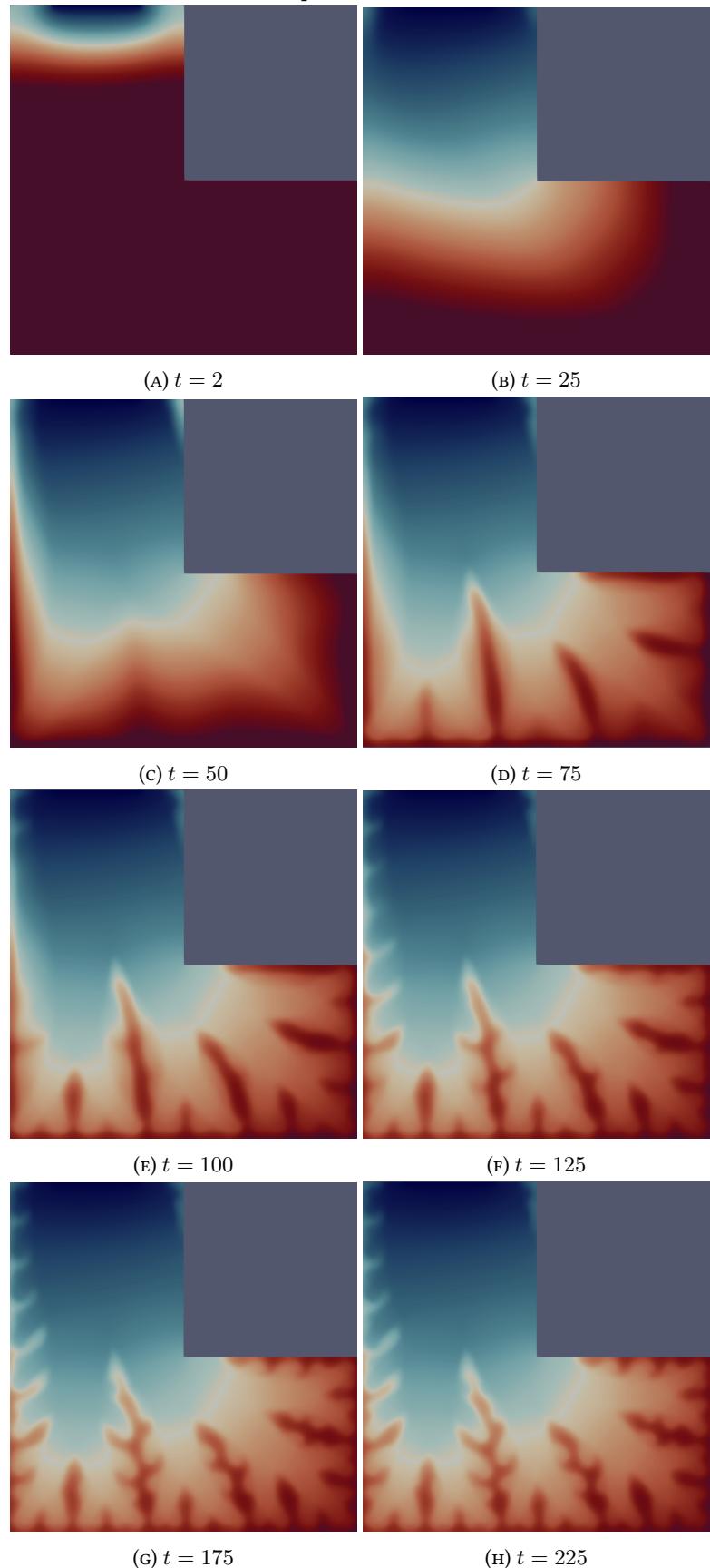


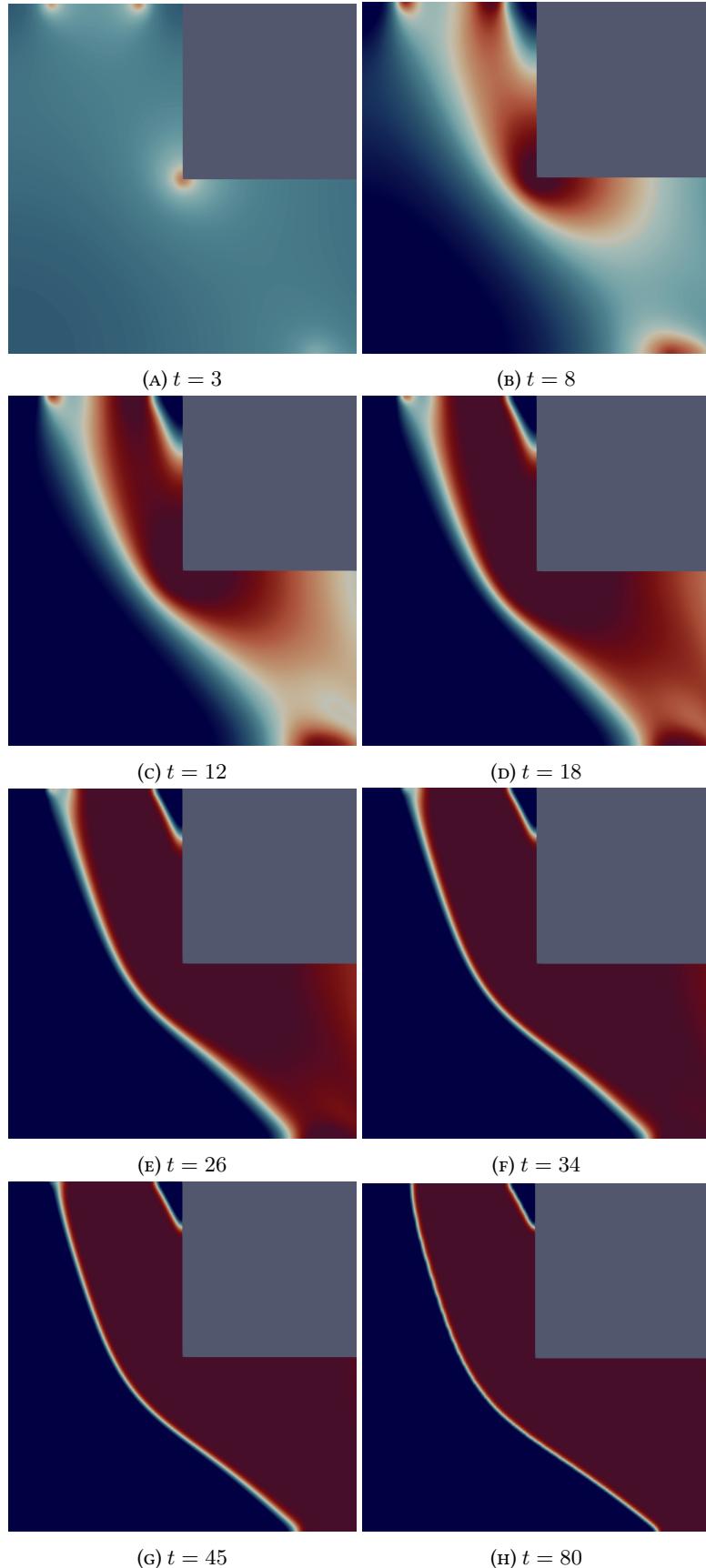
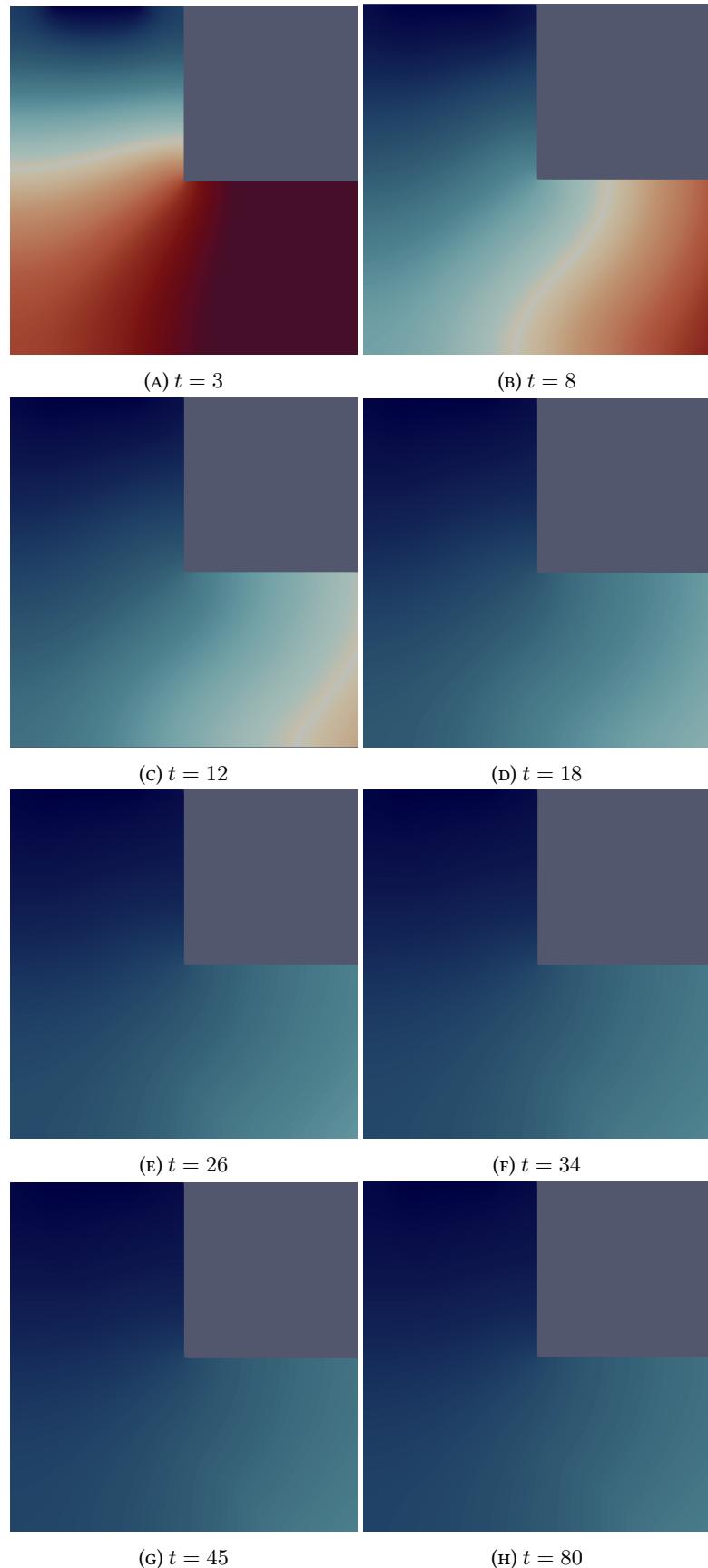
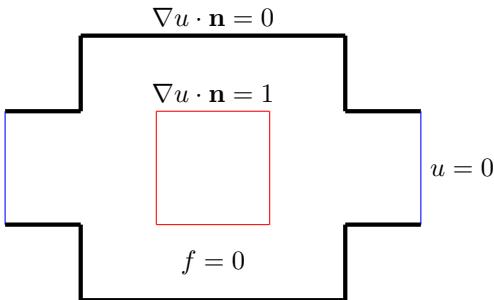
FIGURE 11. *L*-shape with a Sink and Source: Filtered Distribution

FIGURE 12. *L*-shape with a Sink and Source: State Solution

3.4.5. *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 13. L -shaped Ω with a sink.



REFERENCES

- [BS03] Martin P. Bendsøe and O. Sigmund. *Topology Optimization: Theory, Methods, and Applications*. 2nd. Springer, 2003.
- [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 3.4.1.

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     nodenrs = 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 = cholesky(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     end
151     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 topf(
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 """
90     OC(nelx, nely, x, volfrac, dc)
91
92 Optimality criteria update
93
94 # Arguments
95 - 'nelx::S': Number of elements in the horizontal direction
96 - 'nely::S': Number of elements in the vertical direction
97 - 'x::Matrix{T}': Current material distribution
98 - 'volfrac::T': Prescribed volume fraction

```

```

99 - 'dc::Matrix{T}': Sensitivity filter
100
101 # Returns
102 - 'Matrix{T}': Updated material distribution
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
173     dcn[j,i] = dcn[j,i]/(x[j,i]*sum)
174     end
175 end
176
177     return dcn
178 end
179
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
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     for elx = 1:nelx
209         for ely = 1:nely
210             n1 = (nely+1)*(elx-1)+ely
211             n2 = (nely+1)* elx +ely
212             edof = [n1; n2; n2+1; n1+1]
213             K[edof,edof] += (0.001+0.999*x[ely,elx]^penal)*KE
214         end
215     end
216
217     F .= 0.01
218     fixeddofs = Int64(nely/2+1-(nely/20)):Int64(nely/2+1+(nely/20))
219     alldofs = 1:(nely+1)*(nelx+1)
220     freedofs = setdiff(alldofs,fixeddofs)
221
222     U[freedofs] = K[freedofs, freedofs] \ F[freedofs]
223     U[fixeddofs] .= 0
224
225     return U
226 end
227
228 end

```