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. 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

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

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

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, clip(y) $:=$ $\min(\text{max\_val}, \max(\text{min\_val}, y))$, and 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: $\rho = \arg\min \int_\Omega f \cdot u \, \mathrm{d}x$
  $\psi \leftarrow \sigma^{-1}(\theta)$ ▷ 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

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 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

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

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

Now, integrating by parts[1], we expand the left hand side to reach

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

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

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

Integrating by parts similarly, we reach

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

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

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

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

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

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 $\partial_u F$.

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

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*

$$\partial_u(v, w) = 0. \tag{9}$$

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*

$$\partial_u(v, u) = (v, \psi). \tag{10}$$

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

$$\partial_u(u, v) = (\psi, v). \tag{11}$$

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

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

as desired. □

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

$$\partial_u(\nabla v, \nabla u) = (\nabla v, \nabla \psi) \tag{12}$$

Moreover,

$$\partial_u(\nabla u, \nabla v) = (\nabla u, \nabla \psi) \tag{13}$$

holds also by symmetry.

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

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

[3]Noticing 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). □

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

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

by 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

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

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

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

We would now like to find u such that

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

3.3.3. *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.

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,

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

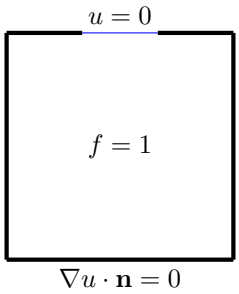
by the equivalence of the primal and dual problems (ergo their solutions; it is from this that we deduce the final equivalence). Setting the derivative equal to zero, we would like to find $\tilde{\rho}$ satisfying

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

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.

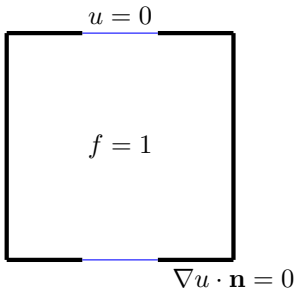
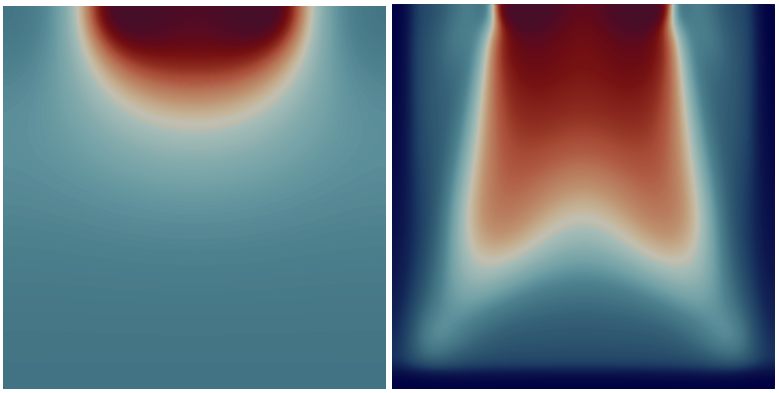
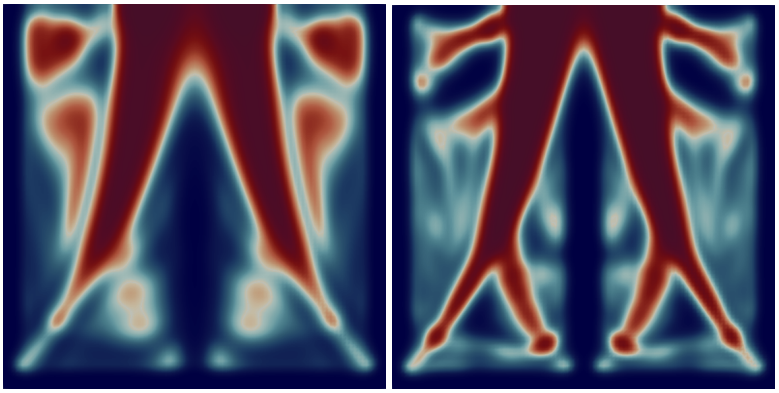


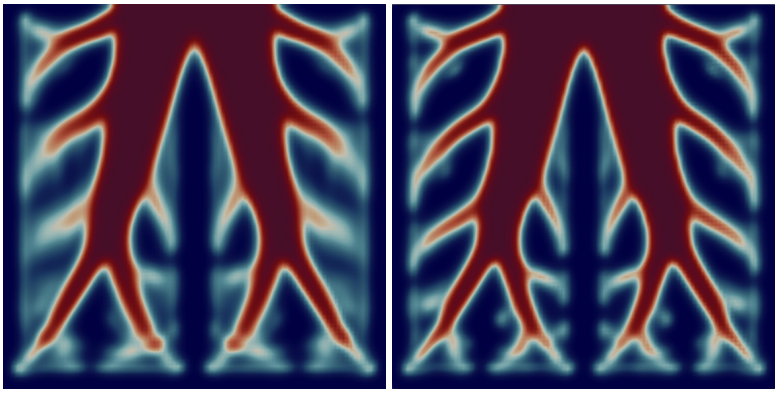
FIGURE 3. Unit Square with Single Sink: Filtered Distribution



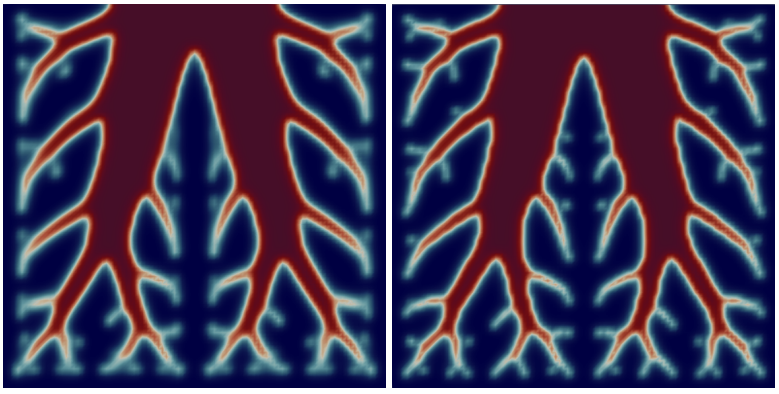
(A) $t = 1$ (B) $t = 6$



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

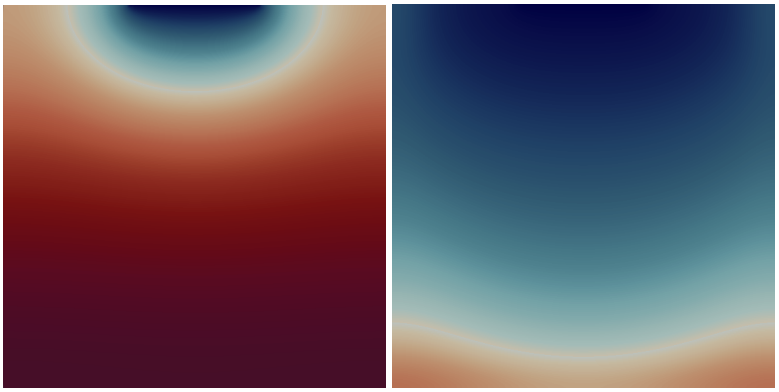


(E) $t = 27$ (F) $t = 35$

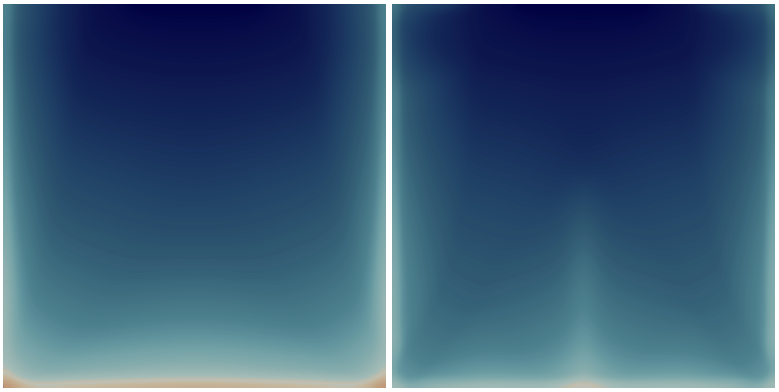


(G) $t = 51$ (H) $t = 65$

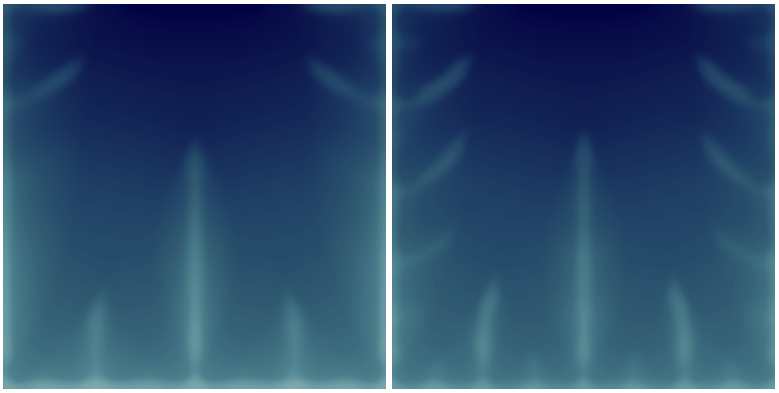
FIGURE 4. Unit Square with Single Sink: State Solution



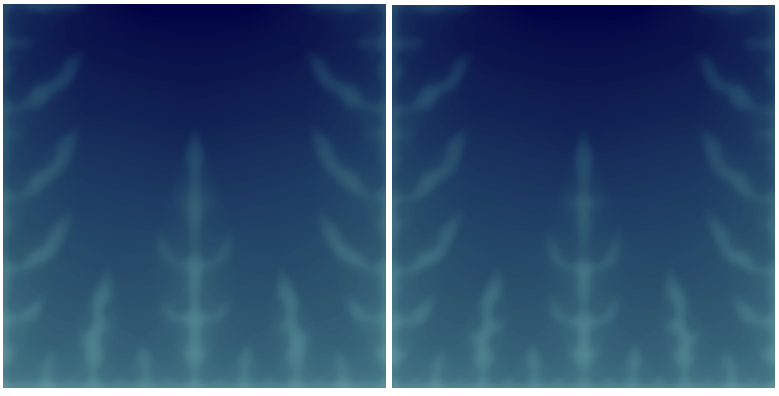
(A) $t = 1$ (B) $t = 6$



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

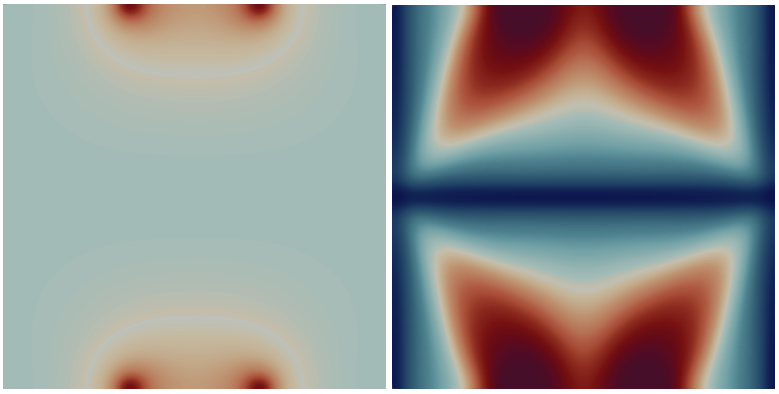


(E) $t = 27$ (F) $t = 35$

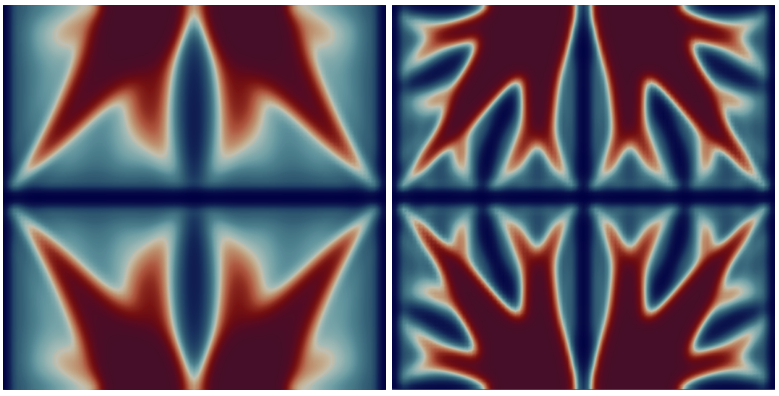


(G) $t = 51$ (H) $t = 65$

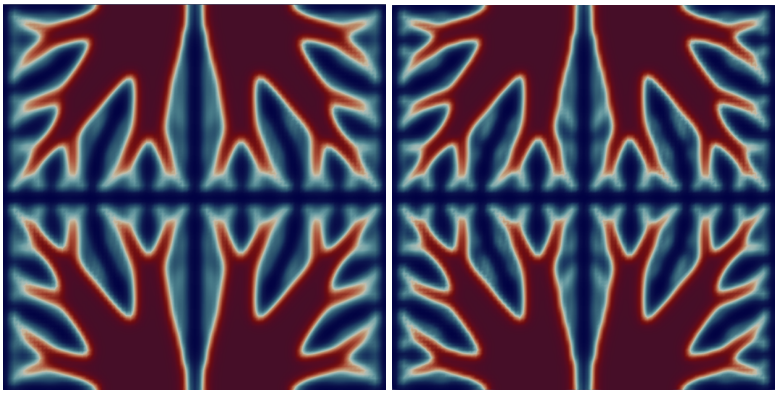
FIGURE 5. Unit Square with Two Sinks: Filtered Distribution



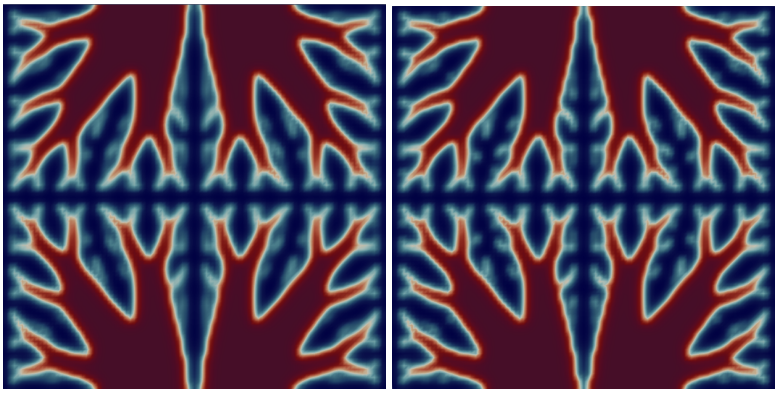
(A) $t = 2$ (B) $t = 16$



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

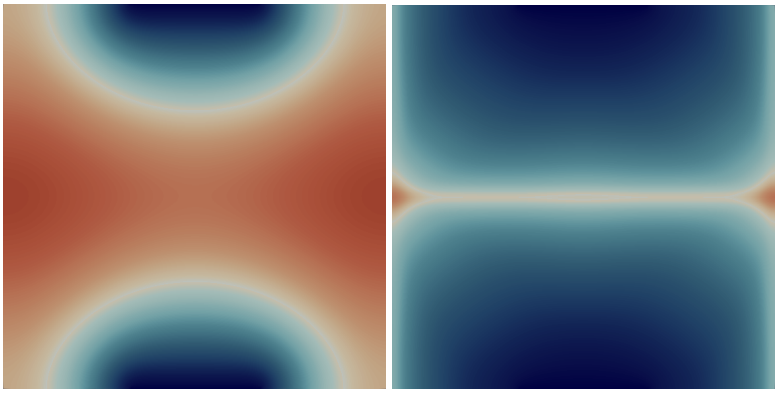


(E) $t = 64$ (F) $t = 80$

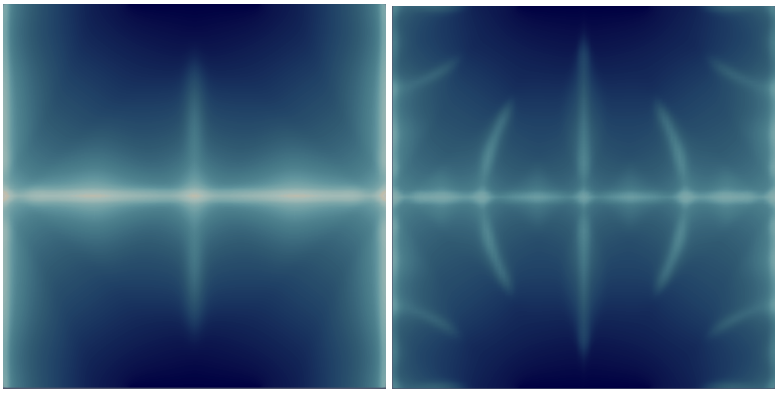


(G) $t = 96$ (H) $t = 127$

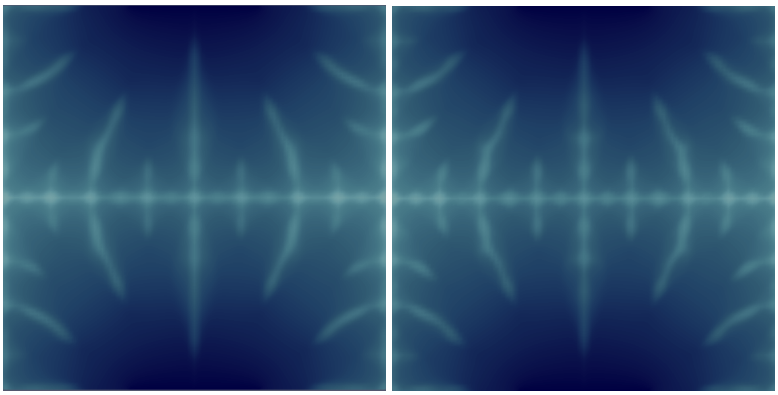
FIGURE 6. Unit Square with Two Sinks Simulation: State Solution



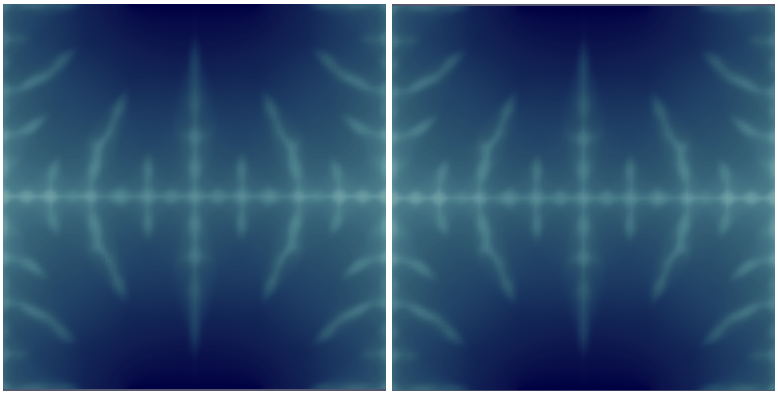
(A) $t = 2$ (B) $t = 16$



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



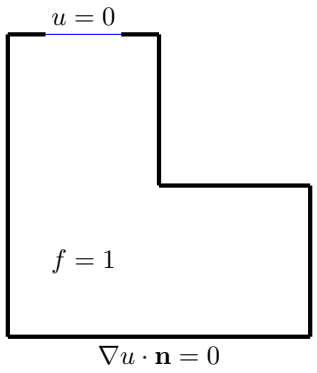
(E) $t = 64$ (F) $t = 80$



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

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.

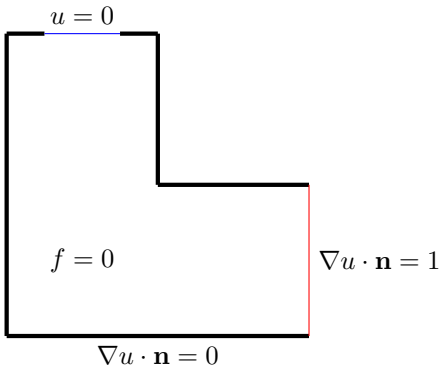
FIGURE 7. L-shaped Ω with a sink.



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 $\alpha = 0.001$.

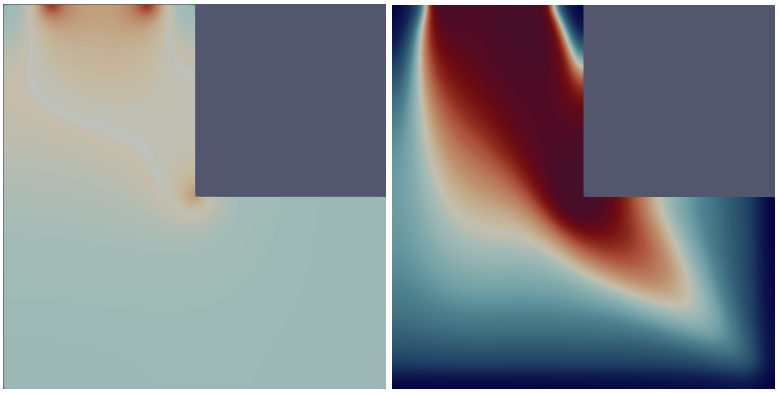
3.4.4. *L-shaped Domain with a Sink and a Source.* We will again allow for heat to escape part of the top edge, but instead of constant heating throughout, we will consider only a source (marked in red; corresponding to given non-homogeneous Neumann boundary conditions) on the right edge.

FIGURE 8. L-shaped Ω with a sink and source.

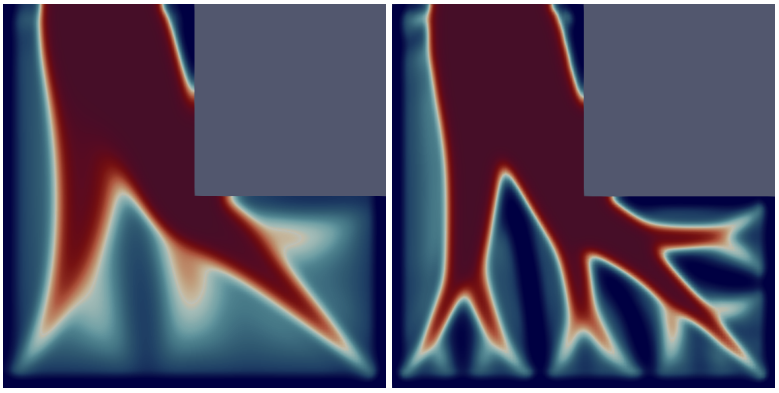


On the physicality of this solution, we can interpret this as simply taking the shortest path to distribute the heat to the sink. This simulation was also run with $\theta = 0.5$ and $\alpha = 0.001$.

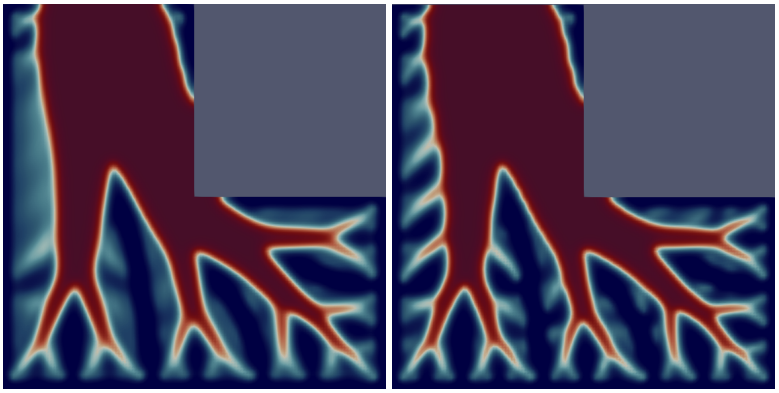
FIGURE 9. L-shape with a Sink: Filtered Distribution



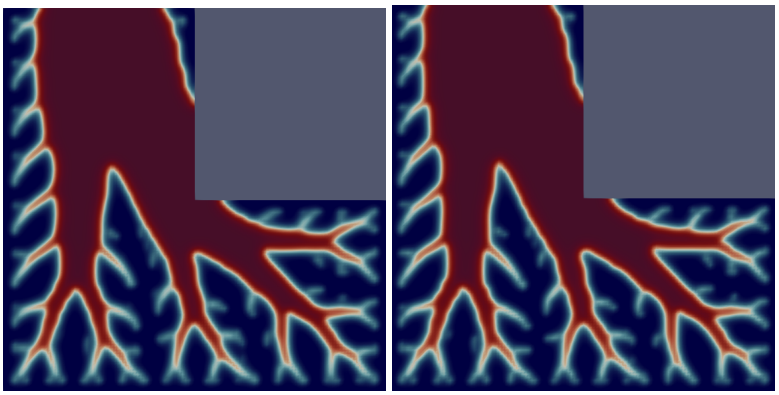
(A) $t = 2$ (B) $t = 25$



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

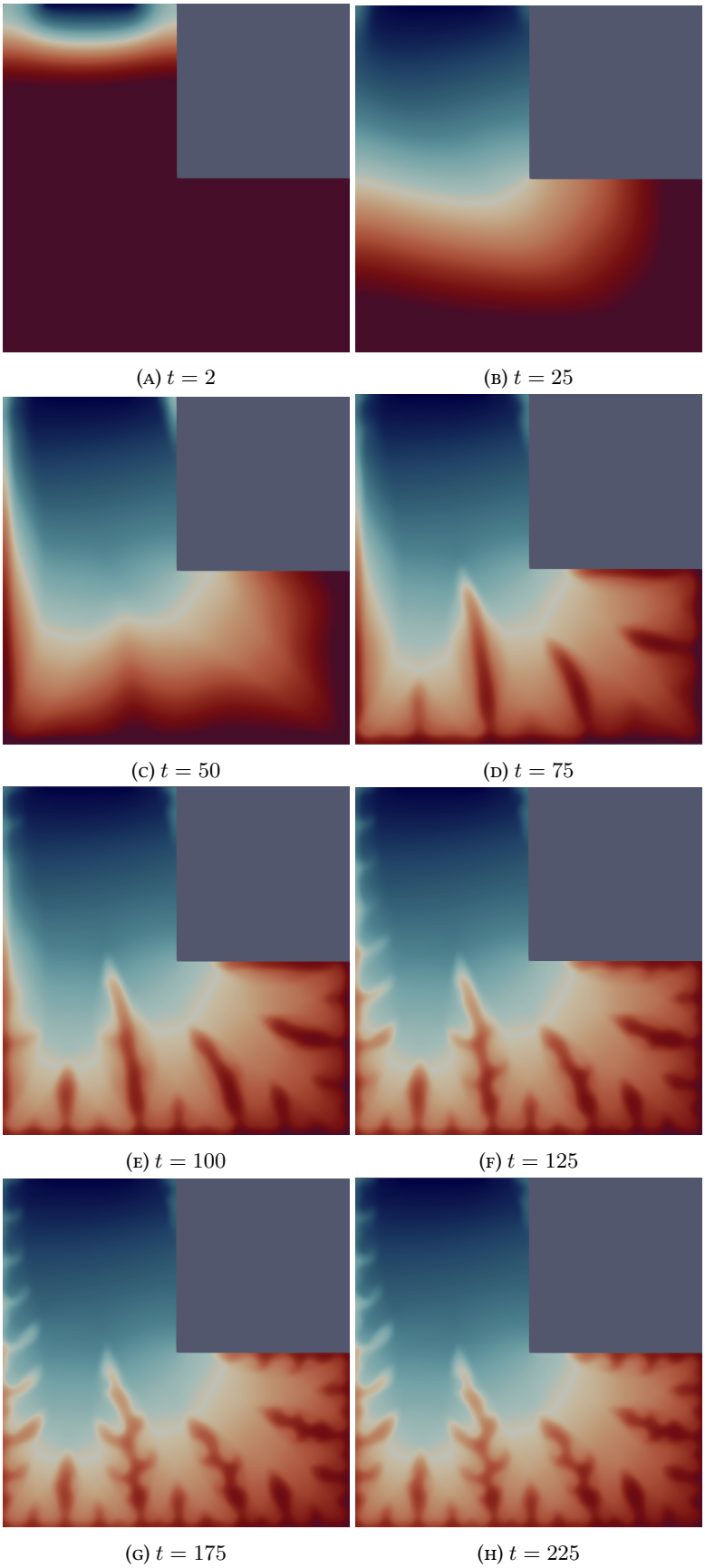


(E) $t = 100$ (F) $t = 125$



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

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



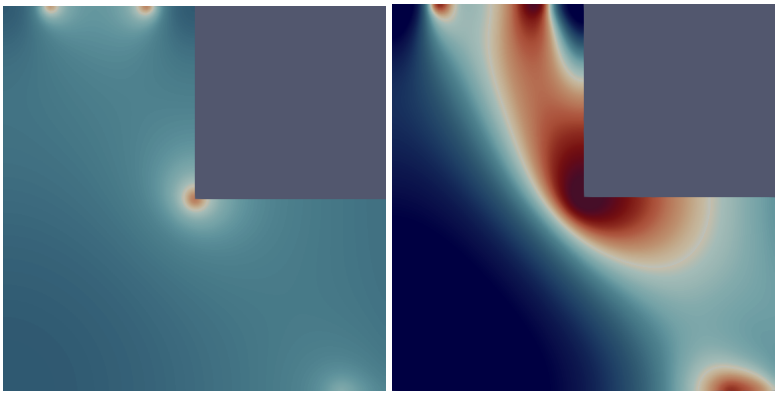
(A) $t = 2$ (B) $t = 25$

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

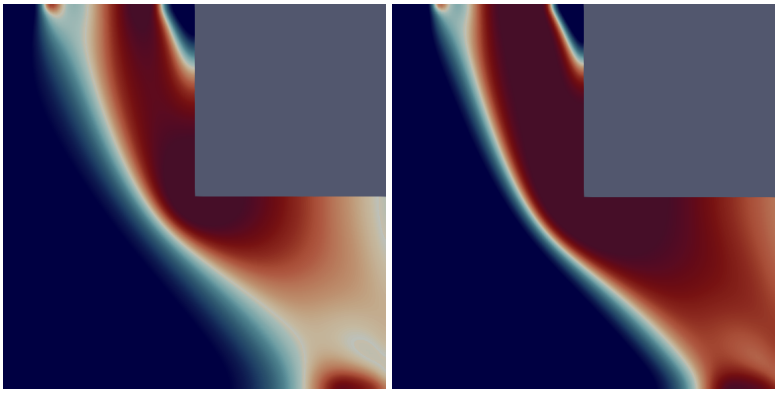
(E) $t = 100$ (F) $t = 125$

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

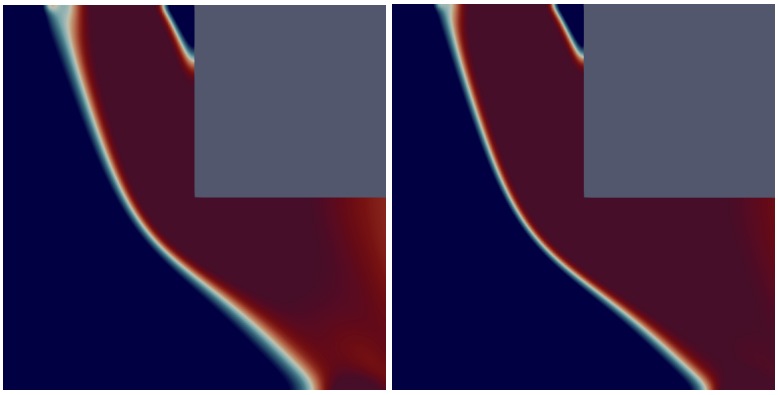
FIGURE 11. L-shape with a Sink and Source: Filtered Distribution



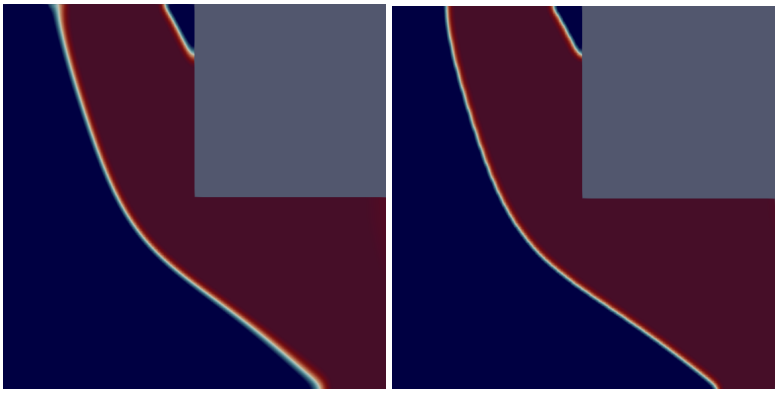
(A) $t = 3$ (B) $t = 8$



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

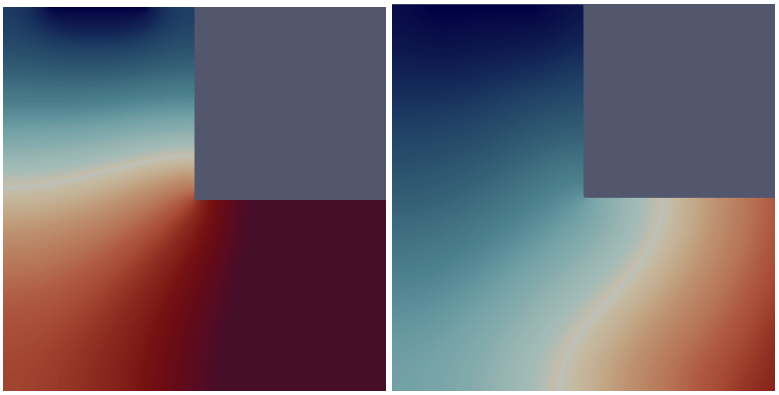


(E) $t = 26$ (F) $t = 34$

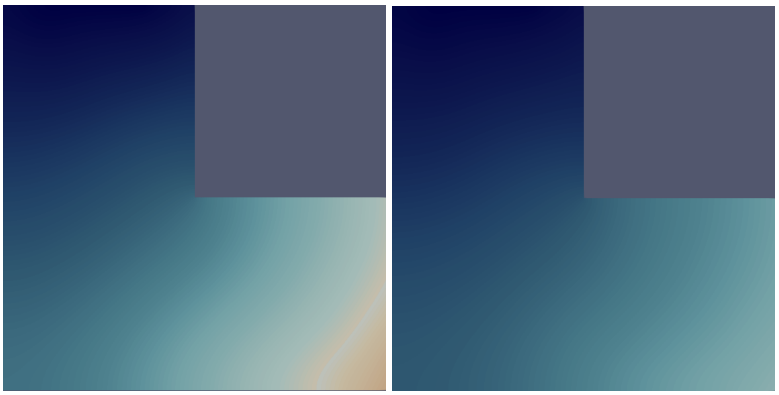


(G) $t = 45$ (H) $t = 80$

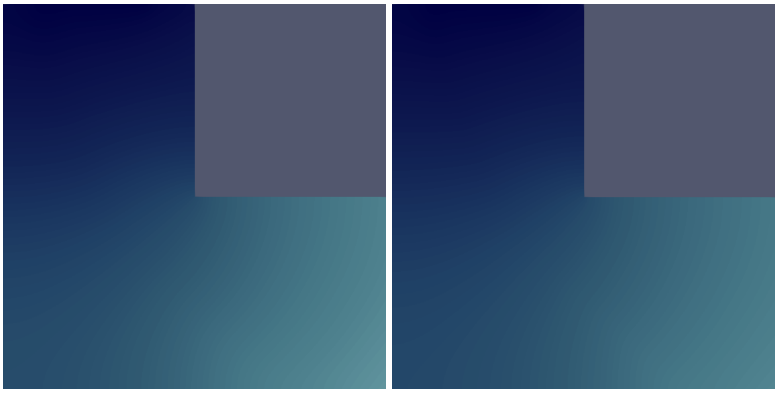
FIGURE 12. L-shape with a Sink and Source: State Solution



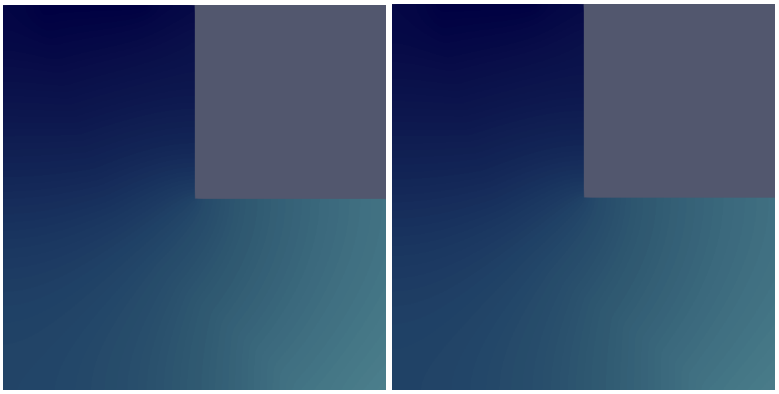
(A) $t = 3$ (B) $t = 8$



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



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



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

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. Heat-sink problem.

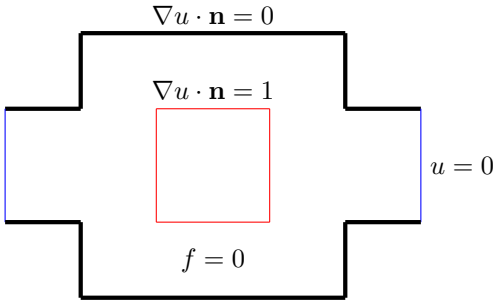
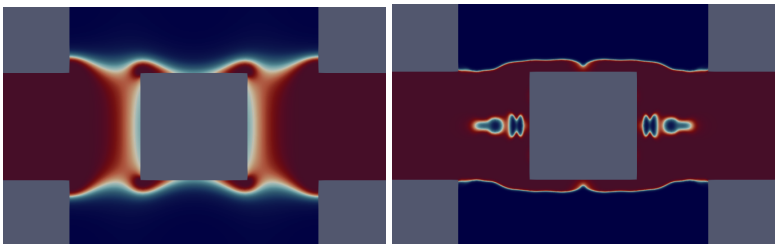
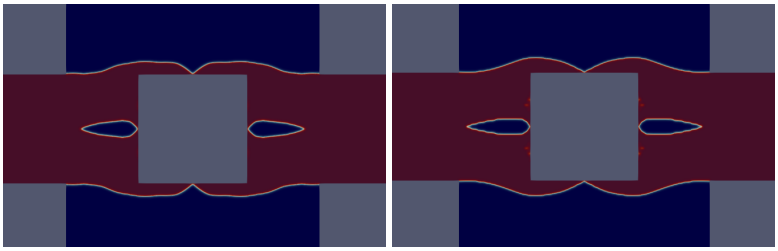


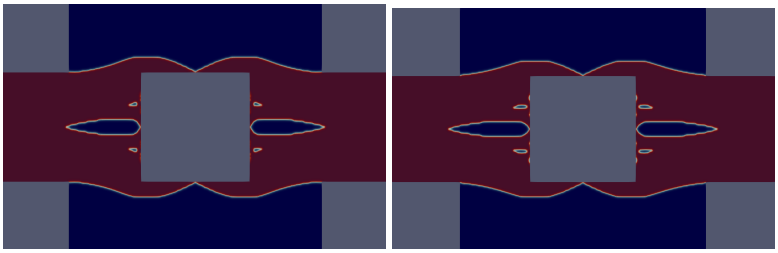
FIGURE 14. Heat sink problem: Filtered Distribution



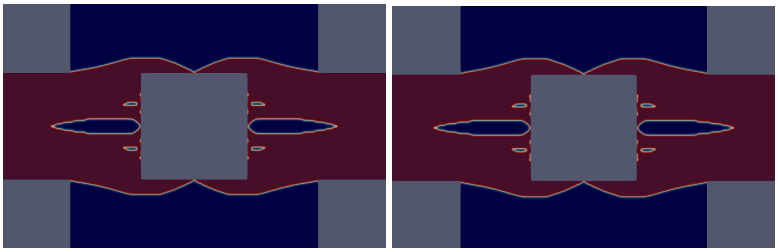
(A) $t = 2$ (B) $t = 5$



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

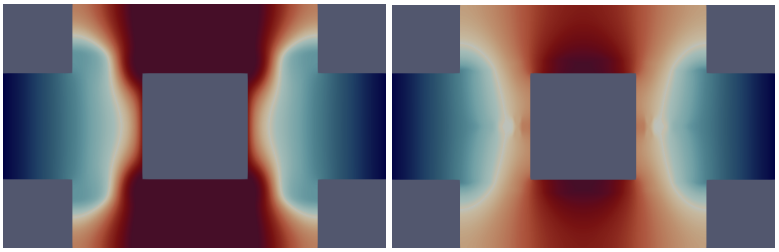


(E) $t = 60$ (F) $t = 90$

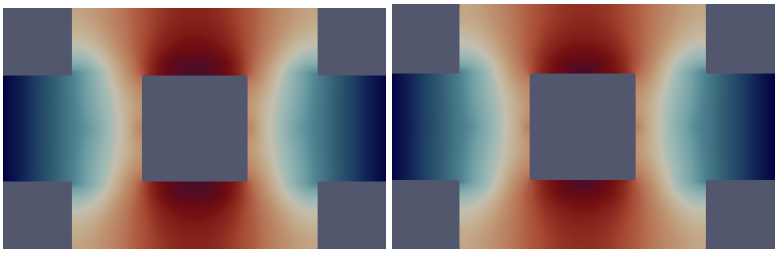


(G) $t = 120$ (H) $t = 150$

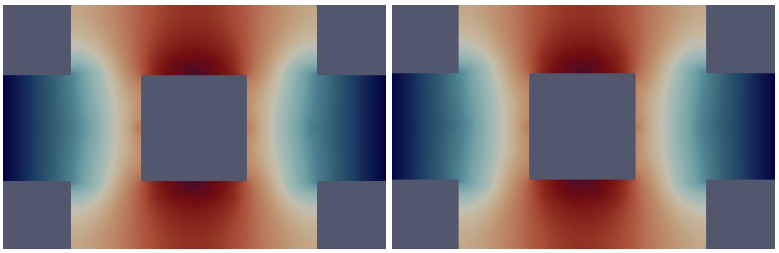
FIGURE 15. Heat sink problem: State Solution



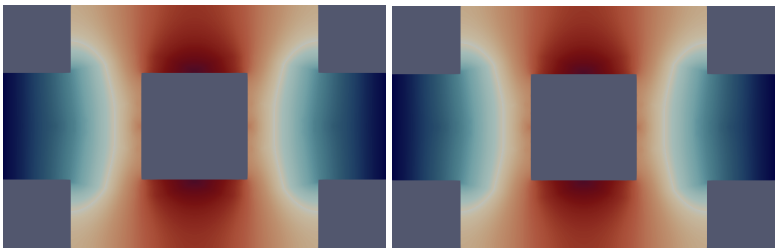
(A) $t = 2$ (B) $t = 5$



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



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



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

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.

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`

```
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
```