UNIVERSITY
OF OSLO

Master's thesis

A Comparison of Classical and Machine Learning Based Approaches to Topology Optimization

Emilie Dørum

Computational Physics
60 ECTS study points

Department of Physics
Faculty of Mathematics and Natural Sciences

Spring 2024

Emilie Dørum

A Comparison of Classical and Machine Learning Based Approaches to Topology Optimization

Supervisors:
Morten Hjorth-Jensen
Thomas M. Surowiec
Henrik Finsberg

Abstract

How to create designs that are optimal with respect to some metric is an important topic in the field of engineering. This is called topology optimization, and is typically done using an iterative minimization algorithm, where, for each iteration, a set of partial differential equations (PDEs) are solved using the finite element method (FEM). However, this approach is computationally expensive for large-scale topology optimization. For this reason, with the rise of the field of artificial intelligence, a new approach might be to use a neural network to approximate the PDEs, hopefully making each iteration less computationally expensive. To see if this new approach shows promise, we have developed a fast and flexible program that can solve topology optimization problems, available from the GitHub repository linked to in appendix A. We have used this program to compare the FEM to a specific neural network based approach called the deep energy method (DEM) by comparing their performance on topology optimization problems based on both the equations of linear elasticity and the Stokes equations. We have made, to the best of our knowledge, three novel contributions; we have extended the DEM implementation in [24] to work with the Stokes equations, we have used a new topology optimization algorithm described in [25] to solve Stokes based topology optimization, and we have combined the DEM implementation with the new algorithm. Using our program, we found that, for linear elasticity, our FEM implementation is faster than our DEM implementation for a rough mesh, but significantly slower for a fine mesh. We have also found that increasing the mesh size improves the FEM result, but makes the DEM result worse. For Stokes flow, our novel FEM-based solver proved to be very fast and gave results comparable to those from the existing literature, while our novel DEM-based solver did not work well enough to be useful in any way. The only clear advantage we have found the DEM to have is that it has an almost constant time complexity, so it can solve topology optimization problems on very fine meshes significantly faster than the FEM. This benefit is however rendered moot by the fact that the DEM gives worse results as the mesh becomes finer, making the usefulness of the DEM for topology optimization questionable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of topology optimization is figuring out how to place material within a domain in order to optimize some aspect of the resulting design. This can for instance be how to design a pipe that minimizes dissipated power, or how to design a truss that is as stiff as possible without wasting any material. These kinds of questions have great importance for the field of engineering, particularly in situations where weight plays a large role, such as in aircraft and aerospace design [47]. The shapes you get from topology optimization might be hard to manufacture with traditional means, but work well combined with additive manufacturing [30].

Topology optimization is an improvement over shape optimization, which optimized shapes without the ability to add or remove holes, and therefore without the ability to change the initial topology. The field of topology optimization has a long history, dating back to the seminal paper on numerical topology optimization *Generating optimal topologies in structural design using a homogenization method* by Bendsøe and Kikuchi in 1988 [5]. Later on, many approaches developed including density [4], level set [2], topological derivatives [41] and phase field [9]. Traditionally, topology optimization is done using an iterative minimization algorithm, and each iteration requires solving a system of partial differential equations (PDEs), typically with the finite element method (FEM). This becomes computationally expensive for large-scale topology optimization. For this reason, with the rise of the field of artificial intelligence, attempts have been made to use machine learning based methods instead. The idea is to use a neural network to approximate the PDEs, making each iteration less computationally expensive, or even developing a network that can solve topology optimization problems in one step, removing the need for an iterative approach. Such iteration-free approaches have had limited success [45], but approaches that replace the classic PDE solver with a neural network have shown promise. One such approach is using a physics informed neural network (PINN) [34], which uses the strong form of the governing PDE as the cost function directly by evaluating the PDE at discrete points. This approach has been successfully applied to linear elasticity [46] and fluid mechanics [11]. In the case where the governing PDE arises from an energy minimization problem, an alternative approach that avoids calculating the higher order gradients present in the strong form by using the energy functional, called the deep energy method (DEM) [36], can be used.

In this thesis we will compare the performance of FEM based topology optimization with the DEM based approach described in the paper *Deep energy method in topology optimization applications* by He et al. [24]. We will compare performance in two applications; finding optimal structures based on the equations of linear elasticity, and finding optimal fluid pathways based on the Stokes equations. For both of these

applications we will use three examples to test the performance of the methods. To the best of our knowledge, this thesis includes three novel contributions. We are going to use a newly developed topology optimization algorithm described in the paper *Proximal Galerkin: A structure-preserving finite element method for pointwise bound constraints* by Keith et al. [25], and apply it to fluid optimization for the first time and combine it with the DEM for the first time. We will also apply the approach by He et al. to fluid optimization for the first time.

This thesis is structured as follows; chapter 2 contains some mathematical foundation and describes some notation, chapter 3 first formulates topology optimization abstractly, and then goes into detail on the two applications we are using. We also give some physical background for the PDEs we are working with. In chapter 4 we describe the methods we will use, namely the iterative solver, the FEM, and the DEM. Chapter 5 shows some important code snippets from the program we have developed, together with short explanations of what the code does. We also show some of the tests we developed to ensure that the program works as intended. In chapter 6 we present our results and use them to compare the two methods, and chapter 7 contains our conclusion. Appendix A contains a link to the GitHub repository where the code developed for this thesis is available.

Chapter 2

Preliminary Results

In this chapter we will cover some of the mathematical foundation that is necessary to understand topology optimization, namely the weak form and the energy functional.

2.1 Notation

To write down the strong form of PDEs, we will use the nabla differential operator $\nabla = \hat{\boldsymbol{x}}_1 \frac{\partial}{\partial x_1} + \cdots + \hat{\boldsymbol{x}}_d \frac{\partial}{\partial x_d}$, where $\hat{\boldsymbol{x}}_i$ is the i-th basis vector, and d is the number of dimensions. We write the gradient and divergence as $\operatorname{grad}(\phi) = \nabla\phi$ and $\operatorname{div}(\boldsymbol{u}) = \nabla \cdot \boldsymbol{u}$, where ϕ is a scalar field and $\boldsymbol{u}$ is a vector field. We also use $\nabla \cdot \nabla = \nabla^2$ as the Laplacian operator. Further, we apply ∇ row-wise when acting on higher order tensors. For instance, the gradient of a 2D vector field $\boldsymbol{u} = (u_1, u_2)$ is

$$\nabla \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} (\nabla u_1)^T \\ (\nabla u_2)^T \end{bmatrix} = \begin{bmatrix} \frac{\partial u_1}{\partial x} & \frac{\partial u_1}{\partial y} \\ \frac{\partial u_2}{\partial x} & \frac{\partial u_2}{\partial y} \end{bmatrix},$$

and the divergence of a 2D matrix field $M = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$ is

$$\nabla \cdot \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} = \begin{bmatrix} \nabla \cdot \begin{bmatrix} m_{11} & m_{12} \end{bmatrix}^T \\ \nabla \cdot \begin{bmatrix} m_{21} & m_{22} \end{bmatrix}^T \end{bmatrix} = \begin{bmatrix} \frac{\partial m_{11}}{\partial x} + \frac{\partial m_{12}}{\partial y} \\ \frac{\partial m_{21}}{\partial x} + \frac{\partial m_{22}}{\partial y} \end{bmatrix}.$$

Note that $\nabla \boldsymbol{u}$ is the Jacobian matrix of $\boldsymbol{u}$.

2.2 The Weak Form

The weak form of a PDE is useful as it lets us decrease the order of the equation by using partial integration. Let u solve the PDE $\mathcal{D}u = f$ on some domain Ω, where $\mathcal{D}$ is a differential operator and f is a function. Provided $\mathcal{D}u$ is regular enough, u also solves the modified equation

$$\int_\Omega (\mathcal{D}u) \cdot v \, \mathrm{d}\boldsymbol{x} = \int_\Omega f \cdot v \, \mathrm{d}\boldsymbol{x}$$

for some sufficiently regular function v, as both multiplication and integration preserves the equality. With the equation in this form, we can use partial integration to reduce the order of the differential operator. For instance, given Poisson's equation $\nabla^2 u + f = 0$ on Ω, we get

$$\int_\Omega \nabla^2 u \cdot v \, \mathrm{d}\boldsymbol{x} + \int_\Omega f \cdot v \, \mathrm{d}\boldsymbol{x} = 0.$$

We can then formally use partial integration on $\int_\Omega \nabla^2 u \cdot v \, \mathrm{d}\boldsymbol{x}$:

$$\int_\Omega \nabla^2 u \cdot v \, \mathrm{d}\boldsymbol{x} = \int_{\delta\Omega} (\nabla u \cdot \boldsymbol{n}) v \, \mathrm{d}s - \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}\boldsymbol{x} \,,$$

where $\delta\Omega$ is the boundary of Ω, and $\mathrm{d}s$ is the surface measure. To ensure a solution exists, we must define some boundary conditions. We define the Dirichlet boundary condition $u = g_D$ on $\Gamma_D \subset \delta\Omega$ and the Neumann boundary condition $\nabla u \cdot \boldsymbol{n} = g_N$ on $\Gamma_N = \delta\Omega \setminus \Gamma_D$. We must also add a requirement on v, namely that $v = 0$ where there are Dirichlet boundary conditions. With this, we get

$$\int_{\delta\Omega} (\nabla u \cdot \boldsymbol{n}) v \, \mathrm{d}s = \int_{\Gamma_N} g_N \cdot v \, \mathrm{d}s \,.$$

It is common to use the notation

$$(u, v) = \int_\Omega u \cdot v \, \mathrm{d}\boldsymbol{x} \,, \;\; (u, v)_\Gamma = \int_\Gamma u \cdot v \, \mathrm{d}s \,,$$

which lets us write the weak form of Poisson's equation as

$$(\nabla u, \nabla v) = (f, v) + (g_N, v)_{\Gamma_N}.$$

With the PDE in its weak form, the solution u can be defined as the function for which the weak form holds for all v. To make this well-defined, v must be an element of some function space V, in which case "all v" means $\forall v \in V$. There are some obvious requirements on V, namely that all functions in V are integrable over Ω. It might seem like another requirement should be that the functions are differentiable, but such a strict requirement is often not suitable in practice [17]. To construct the weak form, we only need partial integration to hold. This leads to the use of the so-called weak derivative, and is the motivation for the construction of the Sobolev space $W^{1,p}(\Omega)$, which consists of all locally summable functions $v : \Omega \to \mathbb{R}$ such that every partial derivative of v exists in a weak sense and belongs to $L^p(\Omega)$ [17]. The function v being locally summable means that $\int_K |v| \, \mathrm{d}\boldsymbol{x} < \infty$ for all compact subsets K of Ω. The function space $L^p(\Omega)$ is called the Lebesgue space, and it is defined as the set of all functions v such that $(\int_\Omega |v|^p \, \mathrm{d}\boldsymbol{x})^{1/p} < \infty$. In the special case $p = 2$, the notation $W^{1,2}(\Omega) = H^1(\Omega)$ is used. We can now define some useful function spaces:

$$\begin{aligned} H_0^1(\Omega, \Gamma) &= \Big\{ v \in H^1(\Omega) \mid v = 0 \text{ on } \Gamma \Big\}, \\ U_g(\Omega, \Gamma) &= \Big\{ u \in H^1(\Omega) \mid u = g \text{ on } \Gamma \Big\}, \\ L_0^2(\Omega) &= \Big\{ q \in L^2(\Omega) \mid \int_\Omega q \, \mathrm{d}\boldsymbol{x} = 0 \Big\}. \end{aligned}$$

Note that $U_g(\Omega, \Gamma)$ is not actually a vector space; it does not contain 0 when $g \neq 0$, and is therefore an affine space. This distinction does however not matter in this context. Typically, the function v is called a test function, and u is called a trial function. For Poisson's equation, the set of test functions V is then $H_0^1(\Omega, \Gamma_D)$, and the set of trial functions U is $U_{g_D}(\Omega, \Gamma_D)$. The weak form of Poisson's equation can then be formulated as follows: *Given* $f \in L^2(\Omega)$*, find* $u \in U_{g_D}(\Omega, \Gamma_D)$ *such that*

$$(\nabla u, \nabla v) = (f, v) + (g_N, v)_{\Gamma_N} \;\; \forall v \in H_0^1(\Omega, \Gamma_D).$$

2.3 The Energy Functional

In many instances, PDEs arise from energy minimization problems, which are studied in the calculus of variations. These kinds of PDEs are called variational problems, and are defined by the following property: the PDE is equal to the derivative of some functional, called the energy functional. From calculus, we remember Fermat's theorem, which states that the value x that minimizes some function $f(x)$ is a solution to $f'(x) = 0$. This also applies to functionals, so the function that minimizes the energy functional is a solution to the original PDE [18].

For a more robust definition of a variational problem, we first define the Lagrangian $\mathcal{L}(\boldsymbol{x}, u(\boldsymbol{x}), \nabla u(\boldsymbol{x}))$ for some differentiable function $u : \Omega \subset \mathbb{R}^n \to \mathbb{R}$. Finding the minimum of the energy functional

$$\psi(u) = \int_\Omega \mathcal{L}(\boldsymbol{x}, u(\boldsymbol{x}), \nabla u(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x}$$

is then the same as finding the solution to the Euler-Lagrange equation

$$\frac{\partial \mathcal{L}}{\partial u} - \sum_{i=1}^{n} \frac{\partial}{\partial x_i}\frac{\partial \mathcal{L}}{\partial u_i'} = 0,$$

where $\boldsymbol{x} = (x_1, \ldots, x_n)$ and $\nabla u = (u_1', \ldots, u_n')$ [18]. We see that if $\mathcal{L} = \frac{1}{2}||\nabla u||^2 - f \cdot u$, then

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial u} &= -f, \\ \frac{\partial}{\partial x_i}\frac{\partial \mathcal{L}}{\partial u_i'} = \frac{\partial}{\partial x_i}\frac{\partial}{\partial u_i'}\left(\frac{1}{2}\sum_{i=1}^{n}(u_i')^2\right) &= \frac{\partial^2 u}{\partial x_i^2}, \end{aligned}$$

thus the Euler-Lagrange equation is equal to Poisson's equation. This means the energy functional for the Poisson's equation is equal to

$$\psi(u) = \frac{1}{2}\int_\Omega ||\nabla u||^2\,\mathrm{d}\boldsymbol{x} - \int_\Omega f \cdot u\,\mathrm{d}\boldsymbol{x}\,.$$

Finding the energy functional this way can be difficult for more complicated PDEs, especially when boundary conditions are involved. Luckily, for linear elliptic second-order PDEs, there is a simpler approach where you start with the weak form of the PDE and set $v = u$. This is not actually allowed when the Dirichlet boundary function $g \neq 0$, but it will typically give you an expression close to the true energy functional. With Poisson's equation, this gives

$$\int_\Omega ||\nabla u||^2\,\mathrm{d}\boldsymbol{x} - \int_\Omega f \cdot u\,\mathrm{d}\boldsymbol{x} - \int_{\Gamma_N} g_N \cdot u\,\mathrm{d}s = 0$$

This is close to $\psi(u) = 0$, but the factor in front of $||\nabla u||^2$ is missing. To get the true functional, you must correct factors to ensure that the derivative of the functional is equal to the original PDE. This gives the true functional

$$\psi(u) = \frac{1}{2}\int_\Omega ||\nabla u||^2\,\mathrm{d}\boldsymbol{x} - \int_\Omega f \cdot u\,\mathrm{d}\boldsymbol{x} - \int_{\Gamma_N} g_N \cdot u\,\mathrm{d}s\,.$$

A solution to Poisson's equation is then found by finding

$$u = \underset{v \in U_{g_D}(\Omega, \Gamma_D)}{\operatorname{argmin}} \psi(v).$$

Note that not all energy functionals have a minimum, so there are conditions the Lagrangian must fulfill for it to be useful. Going into detail about those conditions is outside the scope of this thesis, see [18] for more details. Luckily, all the PDEs we are going to work with in this thesis do fulfill the conditions.

Chapter 3

Introduction to Topology Optimization

The goal of topology optimization is to find the material distribution ρ in a design domain Ω that minimizes an objective function $\phi(\boldsymbol{u}_\rho, \rho)$, where $\boldsymbol{u}_\rho$ is a state function that satisfies a state equation $\mathcal{D}_\rho \boldsymbol{u} = \boldsymbol{f}$, for some differential operator $\mathcal{D}_\rho$ that depends on ρ and some function $\boldsymbol{f}$. The material distribution can take two values; 0 representing absence of material, and 1 representing presence of material. While solving the problem with a discrete material distribution is possible, there are many solvers for discrete problems, none of them are suited for the large scale problems encountered in topology optimization. Instead, it is common to let ρ be continuous, and then penalize values that are not 0 or 1 using a function $r(\rho)$. What this function is depends on the type of topology optimization you are doing, so we will go into further detail in section 3.1 and 3.2. This way of defining topology optimization is called the density approach. Some other approaches are the level set approach, where ρ is defined with respect to a so-called level set function χ by $\rho = (1 + \text{sign}(\chi))/2$, the topological derivatives approach, which is too complicated for us to summarize here, and the phase field approach, which is similar to the density approach except it penalizes intermediate values by adding a term to the objective function [37]. All of these approaches are similar to each other [37], so we will use the density approach as it is one of the most popular approaches.

The optimal topology often includes many small elements, like very thin beams or many small holes. This is a problem when solving the state equation numerically, as numerical solvers need to discretize the equation, making the small elements unrepresentable. This typically results in checkerboard patterns that are nonphysical [28]. Small elements can also cause the simplifying assumptions made for a problem to no longer be valid, making the optimized topology perform poorly in practice. To avoid these problems, it is common to filter ρ to enforce a minimum length scale. A common filter is called the Helmholtz filter [28], also called the screened Poisson equation, which is defined as

$$-\epsilon^2 \nabla^2 \tilde{\rho} + \tilde{\rho} = \rho,$$

where ϵ is the minimum length scale and $\tilde{\rho}$ is the filtered ρ. The filter equation is subject to the Neumann boundary conditions $\nabla \tilde{\rho} \cdot \boldsymbol{n} = 0$ on $\delta\Omega$. The maximum principle guarantees that $0 < \tilde{\rho} < 1$, so filtering ρ preserves its bounds. The weak form of this equation reads: *Given* $\epsilon \in \mathbb{R}_{\geq 0}$ *and* $\rho \in L^2(\Omega)$*, find* $\tilde{\rho} \in H^1(\Omega)$ *such that*

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

The minimization problem is commonly subject to a volume constraint limiting the amount of material present, so we require

$$\int_\Omega \rho(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \leq \gamma|\Omega|, \tag{3.1}$$

where $|\Omega| = \int_\Omega \mathrm{d}\boldsymbol{x}$, and $0 < \gamma < 1$ is the largest fraction of the volume the material is allowed to fill. γ cannot be 0 or 1, as that would necessitate $\rho = 0$ and $\rho = 1$ respectively.

We define $S(\rho)$ to be a function that returns the solution to the state equation, and $F(\rho)$ to be a function that returns the solution to the filter equation. The topology optimization problem can then formally be written as

$$\begin{aligned} \min_{\rho} \quad & \phi(S(F(\rho)), F(\rho)) \\ \text{s.t} \quad & \int_\Omega \rho\,\mathrm{d}\boldsymbol{x} \leq \gamma|\Omega| \\ & 0 \leq \rho \leq 1 \end{aligned}$$

3.1 Topology Optimization of Elastic Materials

3.1.1 Linear Elasticity

In topology optimization of elastic materials, we typically want to find the shape of a body, conforming to a volume constraint, that deforms as little as possible under an applied load. The applied load, called a body force $\boldsymbol{f}$, is a vector field that describes the forces that act on the body at every point. While this does include gravity from the body's own weight, we are often interested in large external forces acting on small areas of the body, so we can ignore the gravitational component.

To describe how a body deforms, we must apply the principles of continuum mechanics. When applying a load to a body, it experiences both stress and strain. Stress describes the internal forces that counteract the applied load, the average restorative internal force per area, while strain describes the relative deformation that is caused by the load. The relationship between stress and strain, called the stress-strain curve, is important to understand how a material acts when a load is applied. See figure 3.1 for en example of a stress-strain curve. For most ductile materials, the stress-strain curve has three parts. For small applied loads, the stress and strain are proportional to each other. This is called elastic deformation, meaning that all deformation is reversed when the load is removed. If the load is large enough, the strain starts to increase faster than the stress. This is called plastic deformation, meaning that some deformation is permanent. If the load is even larger, the material will break. We do not want our material to be permanently deformed, so we assume that it only experiences elastic deformation. This is a reasonable assumption as most materials we use to carry loads, such as steel, have a high yield strength, that is, they can carry a large load without experiencing plastic deformation.

The mathematical model of how solid objects deform from infinitesimal elastic deformations is called linear elasticity. There are three important equations in linear elasticity. The first one governs how the displacements $\boldsymbol{u}$ resulting from $\boldsymbol{f}$ change over time:

$$\nabla \cdot \boldsymbol{\sigma} + \boldsymbol{f} = \bar{\rho}\ddot{\boldsymbol{u}},$$

where $\boldsymbol{\sigma}$ is the Cauchy stress tensor and $\bar{\rho}$ is the density of the material, not to be confused with the material distribution we are optimizing [39]. This equation is an

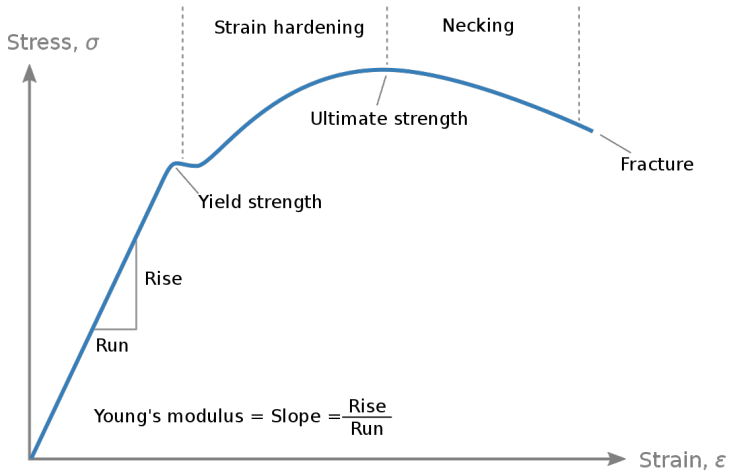


Figure 3.1: Typical stress vs. strain diagram for a ductile material (e.g. steel). Image is downloaded from commons.wikimedia.org/wiki/File: Stress_strain_ductile.svg.

expression of Newton's second law, where the internal forces are trying to counteract the external force. We are interested in the steady state equation where the material has displaced enough to fully counteract $\boldsymbol{f}$. In that case, $\ddot{\boldsymbol{u}} = \boldsymbol{0}$, so our state equation is

$$-\nabla \cdot \boldsymbol{\sigma} = \boldsymbol{f}. \tag{3.2}$$

The second governing equation relates the displacements and the strain:

$$\boldsymbol{\varepsilon} = \frac{1}{2}\Big(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T\Big), \tag{3.3}$$

where $\boldsymbol{\varepsilon}$ is the infinitesimal strain tensor [40]. The third equation relates the stress and the strain:

$$\boldsymbol{\sigma} = \mathsf{C} : \boldsymbol{\varepsilon},$$

where C is the fourth-order stiffness tensor and : represents the inner product between tensors [38]. This is the general equation for Hooke's law, and it comes from the linear relationship between stress and strain. We assume that our material is homogeneous and isotropic, meaning that its physical properties are the same throughout the entire material. In that case, we get:

$$\boldsymbol{\sigma} = \lambda \nabla \cdot \boldsymbol{u}\boldsymbol{I} + 2\mu\boldsymbol{\varepsilon}, \tag{3.4}$$

where $\lambda, \mu > 0$ are called Lamé parameters and $\boldsymbol{I}$ is the identity matrix [40]. The Lamé parameters relate to the material properties Young's modulus E and Poisson ratio ν as follows:

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)},$$
$$\mu = \frac{E}{2(1+\nu)}.$$

Equation (3.2) is typically subject to the Dirichlet boundary condition $\boldsymbol{u} = 0$ on $\Gamma_0 \subset \delta\Omega$, representing a region on the boundary where the material is fixed, and the Neumann boundary condition $\boldsymbol{\sigma n} = \boldsymbol{t}$ on $\Gamma_t \subset \delta\Omega, \Gamma_0 \cap \Gamma_t = \emptyset$, representing a region with an applied traction force. For the rest of the boundary, $\delta\Omega \setminus (\Gamma_0 \cup \Gamma_t)$, $\boldsymbol{\sigma n} = 0$ is used.

3.1.2 Linear Elasticity with Varying Density

$\boldsymbol{\sigma}$ is proportional to the Young's modulus of the material, a material property that tells us how easy the material is to deform. The Young's modulus is not defined for the empty regions, as it is only a property of solid materials, but we could claim that it is zero as deforming an empty region does not require any energy. This leads to the naive approach for introducing the material distribution ρ into the elastic compliance problem, which is to simply multiply $\boldsymbol{\sigma}$ with $\tilde{\rho}$. This has two problems; it does not penalize values of $\tilde{\rho}$ between 0 and 1, and it causes the PDE to be singular if $\tilde{\rho} = 0$. To solve these problems, the modified solid isotropic material penalization (SIMP) model is often used:

$$r(\tilde{\rho}) = r_{\min} + \tilde{\rho}^p(1 - r_{\min}),$$

where $0 < r_{\min} \ll 1$ is a small value that helps to avoid singularities, which can be interpreted as the fraction E_0/E, where E_0 is the Young's modulus of the empty regions. $p > 1$ is a penalization parameter that penalizes intermediate values, and it has been found that optimal value for p is 3 [37]. A potential reason for why this is the optimal number is that it is the lowest p where intermediate densities are physically realizable [6]. By physically realizable we mean that the intermediate densities act as a composite material made from void and the material we get when $\rho = 1$. The reason the SIMP model works is that it causes the stiffness of the material to be sublinearly proportional to the density ρ. The volume is linearly proportional to ρ, so intermediate densities are in a sense an inefficient use of volume. Combined with a volume constraint, this penalizes intermediate values. For the value $r_{\min}$, we used 10^{-6}.

We want to maximize the stiffness of the material, which is the same as minimizing its compliance. This gives us the objective function

$$\phi(\rho) = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{u}_\rho \, \mathrm{d}\boldsymbol{x} + \int_{\Gamma_t} \boldsymbol{t} \cdot \boldsymbol{u}_\rho \, \mathrm{d}s \,,$$

where $\boldsymbol{u}_\rho = S(F(\rho))$ is the solution to the elasticity equations.

To derive the weak form of the elasticity equations, we first combine (3.2), (3.4), and the modified Young's modulus to get the state equation

$$-\nabla \cdot (r(\tilde{\rho})(\lambda(\nabla \cdot \boldsymbol{u}\boldsymbol{I}) + 2\mu\boldsymbol{\varepsilon})) = \boldsymbol{f}.$$

Using Korn's inequality, we can find the weak form of this equation [13]: *Given* $\lambda, \mu \in \mathbb{R}_{\geq 0}$, $\tilde{\rho} \in H^1(\Omega)$ *and* $f \in L^2(\Omega)$, *find* $u \in H_0^1(\Omega, \Gamma_0)$ *such that*

$$\begin{aligned} &\lambda(r(\tilde{\rho})\nabla \cdot \boldsymbol{u}, \nabla \cdot \boldsymbol{v}) + 2\mu(r(\tilde{\rho})\boldsymbol{\varepsilon}(\boldsymbol{u}), \boldsymbol{\varepsilon}(\boldsymbol{v})) \\ &= (\boldsymbol{f}, \boldsymbol{v}) + (\boldsymbol{t}, \boldsymbol{v})_{\Gamma_t} \ \forall v \in H_0^1(\Omega, \Gamma_0). \end{aligned} \tag{3.5}$$

From this equation, a straight forward calculation gives the energy functional

$$\begin{aligned} \psi(\boldsymbol{u}; \rho) = \frac{1}{2} \int_\Omega r(\tilde{\rho}) \Big(\lambda ||\nabla \cdot \boldsymbol{u}||^2 + 2\mu ||\boldsymbol{\varepsilon}||^2 \Big) \, \mathrm{d}\boldsymbol{x} \\ - \int_\Omega \boldsymbol{f} \cdot \boldsymbol{u} \, \mathrm{d}\boldsymbol{x} - \int_{\Gamma_t} \boldsymbol{t} \cdot \boldsymbol{u} \, \mathrm{d}s \,. \end{aligned} \tag{3.6}$$

3.1.3 Elasticity Optimization Examples

For the three linear elasticity examples we are going to use, one is from *Proximal Galerkin: A structure-preserving finite element method for pointwise bound constraints*

[25] and two are from *Deep energy method in topology optimization applications* [24]. All of them use the domain $\Omega = [0, w] \times [0, h]$, for some width w and height h. The first one is a cantilever beam that is one unit tall and three units wide. We are going to use

$$\begin{aligned}
\epsilon &= 0.02, \\
\gamma &= 0.5, \\
\Gamma_0 &= \{(x, y) \in \Omega \mid x = 0\}, \ \Gamma_t = \emptyset, \\
\boldsymbol{f} &= \begin{cases} (0, -1) & \mid (x - 2.9)^2 + (y - 0.5)^2 \leq 0.05^2 \\ \mathbf{0} & \mid \text{Otherwise} \end{cases}, \\
E &= \frac{5}{2}, \ \nu = \frac{1}{4},
\end{aligned}$$

which means that the cantilever can fill half of the domain, is fixed on its left side, and a force equal to $(0, -1)$ is applied in a circular area with center $(2.9, 0.5)$ and radius 0.05. E and ν is chosen such that $\lambda = \mu = 1$. A sketch of the cantilever is shown in figure 3.2.

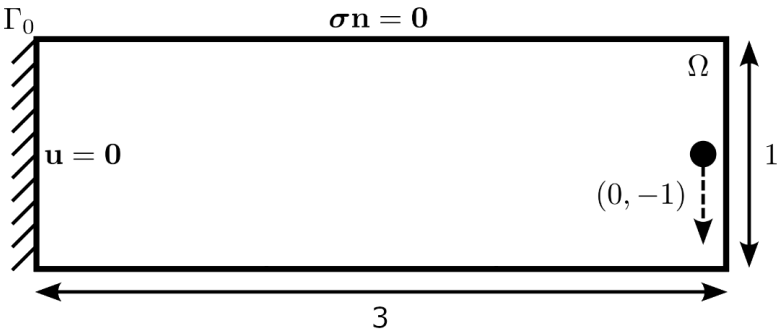


Figure 3.2: Figure of the cantilever beam we are going to optimize. Based on figure 6.2 from [25].

The second example is a short cantilever beam that is five units tall and 10 units wide. We are going to use

$$\begin{aligned}
\epsilon &= \frac{0.25}{2\sqrt{3}}, \\
\gamma &= 0.4, \\
\Gamma_0 &= \{(x, y) \in \Omega \mid x = 0\}, \\
\Gamma_t &= \left\{(x, y) \in \Omega \mid x = 10 \ \wedge \ y \in \left[\frac{5}{2} - \frac{1}{18}, \frac{5}{2} + \frac{1}{18}\right]\right\}, \\
\boldsymbol{t} &= (0, -2000), \ \boldsymbol{f} = 0, \\
E &= 2 \cdot 10^5, \ \nu = 0.3,
\end{aligned}$$

which means that the cantilever can fill 40% of the domain, is fixed on its left side, and a force equal to $(0, -2000)$ is applied in a small line with center $\frac{5}{2}$ and length $\frac{1}{9}$ on its right edge. A sketch of the short cantilever is shown in figure 3.3.

The third example is a bridge that is two units tall and 12 units wide. We are going

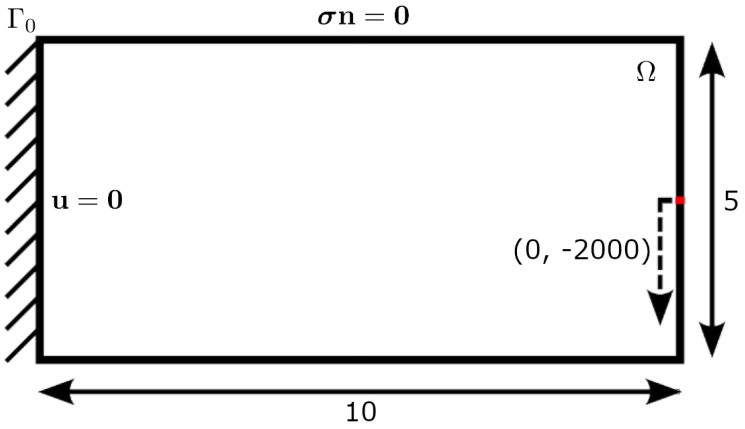


Figure 3.3: Figure of the short cantilever beam we are going to optimize. Based on an example from [24].

to use

$$
\begin{aligned}
\epsilon &= \frac{0.25}{2\sqrt{3}}, \\
\gamma &= 0.4, \\
\Gamma_0 &= \{(x, y) \in \Omega \mid x = 0 \ \vee \ x = 12\}, \\
\Gamma_t &= \left\{(x, y) \in \Omega \mid y = 2 \ \wedge \ x \in \left[6 - \frac{1}{4}, 6 + \frac{1}{4}\right]\right\}, \\
\boldsymbol{t} &= (0, -2000), \ \boldsymbol{f} = 0, \\
E &= 2 \cdot 10^5, \ \nu = 0.3,
\end{aligned}
$$

which means that the bridge can fill 40% of the domain, is fixed on both its left and right edge, and a force equal to $(0, -2000)$ is applied in a small line with center 6 and length $\frac{1}{2}$ on its top edge. A sketch of the bridge is shown in figure 3.4.

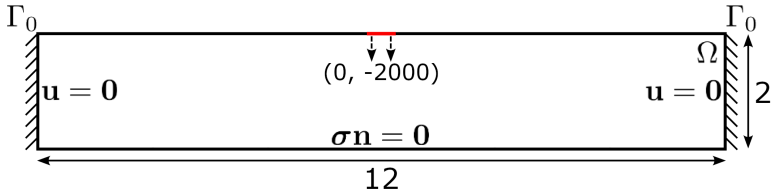


Figure 3.4: Figure of the bridge we are going to optimize. Based on an example from [24].

3.2 Topology Optimization of Fluids

3.2.1 Stokes Flow

In topology optimization of fluids, we typically want to find the shape of some kind of pipe, conforming to a volume constraint, in which a fluid dissipates the least amount of energy while traveling through it. We are going to assume that our fluid is Newtonian and incompressible. A fluid being incompressible means that its density does not change with pressure, which is typically approximately true for liquids and not true for gasses. A fluid being Newtonian means that the viscous stress from its flow is proportional to

the local strain rate. The definition of a Newtonian fluid is similar to the definition of an elastic deformation, and indeed, incompressible Newtonian fluids are subject to similar equations as those from linear elasticity. The only change is the equation of motion, which now describes how the fluid velocity $\boldsymbol{u}$ changes over time:

$$\nabla \cdot \boldsymbol{\sigma} + \boldsymbol{f} = \bar{\rho}\left(\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u}\right), \tag{3.7}$$

where the body force $\boldsymbol{f}$ can be interpreted as a forcing term that pushes the fluid in a certain direction [1]. The density $\bar{\rho}$ is the density of the fluid, and it is not related to the material distribution ρ.

The Cauchy stress tensor for an incompressible Newtonian fluid is

$$\boldsymbol{\sigma} = 2\mu\boldsymbol{\varepsilon} - p\boldsymbol{I}, \tag{3.8}$$

where μ is the viscosity of the fluid and p is the fluid pressure [1]. The incompressibility condition can be written in terms of $\boldsymbol{u}$ as $\nabla \cdot \boldsymbol{u} = 0$. Typically, equation (3.8) is combined with (3.3) and (3.7), which results in the Navier-Stokes equations

$$\begin{aligned} \bar{\rho}\left(\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u}\right) &= \mu\nabla^2\boldsymbol{u} - \nabla p + \boldsymbol{f}, \\ \nabla \cdot \boldsymbol{u} &= 0. \end{aligned}$$

We want to make two further simplifications. We assume that the fluid flow has reached a steady state, so the time derivative is zero. Further, we assume that the Reynolds number of the flow is small, so $\bar{\rho}(\boldsymbol{u} \cdot \nabla)\boldsymbol{u} = 0$. The flow having a low Reynolds number means that the fluid velocities are small, the viscosity is large, or the length-scale of the flow is small. With these simplifications, we get the Stokes equations

$$\begin{aligned} \mu\nabla^2\boldsymbol{u} - \nabla p + \boldsymbol{f} &= 0, \\ \nabla \cdot \boldsymbol{u} &= 0. \end{aligned}$$

We are not going to use a forcing term, so $\boldsymbol{f} = 0$. For the boundary conditions, we are going to use the Dirichlet boundary condition $\boldsymbol{u} = \boldsymbol{g}$ on $\delta\Omega$, where $\boldsymbol{g}$ describes boundary flows. We are going to be using parabolic boundary flows, which means they can be parameterized by

$$v(t;\ c, l, \hat{v}) = \hat{v} \max\left\{\left(1 - \frac{2(t-c)}{l}\right)^2,\ 0\right\},$$

where c is the center of the flow region, l is the length of the flow region, and $\hat{v}$ is the flow rate at c. We can then write $\boldsymbol{g}$ as

$$\boldsymbol{g}(x, y) = \sum_i \boldsymbol{v}_{\text{side}_i}(x, y;\ c_i, l_i, \hat{v}_i),$$

where $\boldsymbol{v}_{\text{side}} : \delta\Omega \to \mathbb{R}^2$ is defined as,

$$\boldsymbol{v}_{\text{side}}(x, y) = \begin{cases} v(y)\hat{\boldsymbol{x}} & \mid \text{side} = \text{left} \wedge x = 0 \\ -v(y)\hat{\boldsymbol{x}} & \mid \text{side} = \text{right} \wedge x = w \\ -v(x)\hat{\boldsymbol{y}} & \mid \text{side} = \text{top} \wedge y = h \\ v(x)\hat{\boldsymbol{y}} & \mid \text{side} = \text{bottom} \wedge y = 0 \\ 0 & \mid \text{otherwise} \end{cases},$$

where $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{y}}$ are the unit vectors in the x- and y-directions respectively. The minus signs in the right and top part of the definition are necessary so a positive $v_{\max}$ always represents an inflow, and a negative $v_{\max}$ always represents an outflow. It is important that the amount of fluid flowing into the domain is the same as the amount flowing out, which means that

$$\int_{\delta\Omega} \boldsymbol{g} \cdot \boldsymbol{n} \, \mathrm{d}s = 0,$$

which can be simplified to

$$\sum_i l_i \hat{v}_i = 0,$$

as long as the flow regions never extend past the domain.

3.2.2 Stokes Flow with Varying Permeability

Unlike elastic compliance, introducing our material distribution to Stokes flow is not trivial. As we are using a two-dimensional domain, we can use lubrication theory, a theory that describes how fluid flows in a domain in which one dimension is significantly smaller than the others. We can use this by assuming that our fluid flows in a domain $\Omega \times \left[-\frac{h}{2}, \frac{h}{2}\right]$, where h is a small value representing the height of the domain. From this, you get the modified state equations

$$\mu \nabla^2 \boldsymbol{u} - \nabla p - \kappa \boldsymbol{u} = 0, \tag{3.9}$$

$$\nabla \cdot \boldsymbol{u} = 0, \tag{3.10}$$

where $\kappa \propto \frac{1}{h^2}$ [22]. Fluid can easily flow in areas with a large h, and fluid cannot easily flow in areas with a small h, so if we assume that $h = h(x, y)$, we get an equation where $\kappa \ll 1$ represents presence of fluid, and $\kappa \gg 1$ represents absence of fluid. We therefore want $\kappa = r(\rho)$, $0 < r(1) = r_{\min} \ll 1$, $r(0) = r_{\max} \gg 1$. If r is a linear function, the optimal topology includes only discrete values of ρ [8], but in practice this makes it difficult for a solver to converge to the correct solution. It is therefore common to add a penalization parameter $q > 0$, giving us

$$r(\rho) = r_{\max} + (r_{\min} - r_{\max})\rho \frac{1+q}{\rho+q}.$$

As $q \to \infty$, this function approaches a linear function, so a higher q-value gives a more discrete solution. In practice, a value of 0.1 is often enough to get sharp boundaries between regions with and without fluid [8]. For the limits of r, we used the values $r_{\min} = 2.5/100^2$ and $r_{\max} = 2.5/0.01^2$. While the approach of using lubrication theory only makes physical sense with a 2D domain, equation (3.9) also works in 3D [22]. In that case, $r(\rho)$ describes the inverse permeability of some porous material the fluid flows through.

Unlike the case with elastic compliance, adding many small holes typically produces a worse result with fluids [8]. This means that we do not need to filter ρ. For our objective, we want to minimize the total potential power of a fluid, given by

$$\phi(\rho) = \frac{1}{2} \int_\Omega r(\rho) ||\boldsymbol{u}_\rho||^2 + \mu ||\nabla \boldsymbol{u}_\rho||^2 \, \mathrm{d}\boldsymbol{x} \,,$$

where $\boldsymbol{u}_\rho = S(\rho)$ is the solution to the modified state equations.

To get the weak form of the state equations for Stokes flow, we multiply (3.9) with a test velocity field $\boldsymbol{v}$ and (3.10) with a test pressure field q, and then apply partial integration. This gives

$$\begin{aligned}(r(\rho)\boldsymbol{u}, \boldsymbol{v}) + \mu(\nabla \boldsymbol{u}, \nabla \boldsymbol{v}) + (\nabla p, \boldsymbol{v}) = 0 \ \forall v \in H_0^1(\Omega, \delta\Omega),\\ (\nabla \cdot \boldsymbol{u}, q) = 0 \ \forall q \in L_0^2(\Omega).\end{aligned}$$

We need both of these equations to hold, so if we add them together, we get the weak form of the Stokes equations: *Given* $\mu \in \mathbb{R}_{\geq 0}$ *and* $\rho \in L^2(\Omega)$*, find* $(\boldsymbol{u}, p) \in U_{\boldsymbol{g}}(\Omega, \delta\Omega) \times L_0^2(\Omega)$ *such that*

$$\begin{aligned}&(r(\rho)\boldsymbol{u}, \boldsymbol{v}) + \mu(\nabla \boldsymbol{u}, \nabla \boldsymbol{v}) + (\nabla p, \boldsymbol{v}) + (\nabla \cdot \boldsymbol{u}, q) = 0\\ &\forall (\boldsymbol{v}, q) \in H_0^1(\Omega, \delta\Omega) \times L_0^2(\Omega).\end{aligned} \tag{3.11}$$

The energy functional is given by

$$\begin{aligned}\psi(\boldsymbol{u}, p; \rho) = \frac{1}{2} \int_\Omega r(\rho)||\boldsymbol{u}||^2 + \mu||\nabla \boldsymbol{u}||^2 \,\mathrm{d}x\\ + \int_\Omega \nabla p \cdot \boldsymbol{u} + (\nabla \cdot \boldsymbol{u}) p \,\mathrm{d}x\,.\end{aligned} \tag{3.12}$$

The fact that $(\nabla \cdot \boldsymbol{u}, q)$ becomes $(\nabla \cdot \boldsymbol{u}, p)$ instead of $(\nabla \cdot \boldsymbol{u}, \nabla \cdot \boldsymbol{u})$ might seem a bit weird, but is necessary, as we now get

$$\begin{aligned}\frac{\partial \psi}{\partial \boldsymbol{u}} &= \mu \nabla^2 \boldsymbol{u} - \nabla p - r(\rho)\boldsymbol{u},\\ \frac{\partial \psi}{\partial p} &= \nabla \cdot \boldsymbol{u},\end{aligned}$$

so finding $\nabla \psi = 0$ is equivalent to finding the solution to the Stokes equations.

There is a simpler way of defining the energy functional that does not include the pressure. If $\boldsymbol{u}$ is divergence free, we can write [8]

$$\psi(\boldsymbol{u}; \rho) = \frac{1}{2} \int_\Omega r(\rho)||\boldsymbol{u}||^2 + \mu||\nabla \boldsymbol{u}||^2 \,\mathrm{d}\boldsymbol{x}\,. \tag{3.13}$$

Clearly, this is similar to the objective value. In fact, the objective can be defined with respect to the functional as

$$\phi(\rho) = \min_{\boldsymbol{u} \in U_{\text{div}}} \psi(\boldsymbol{u}; \rho),$$

where $U_{\text{div}} = \{\boldsymbol{u} \in U_{\boldsymbol{g}}(\Omega, \delta\Omega) \mid \nabla \cdot \boldsymbol{u} = 0\}$.

3.2.3 Fluid Optimization Examples

We are going to use three of the examples from *Topology optimization of fluids in Stokes flow* [8]. The first one is a diffuser that is one unit wide and one unit tall. We are going to use

$$\begin{aligned}\gamma &= 0.5,\\ \boldsymbol{g}(x, y) &= \boldsymbol{v}_{\text{left}}\left(x, y;\ \frac{1}{2}, 1, 1\right) + \boldsymbol{v}_{\text{right}}\left(x, y;\ \frac{1}{2}, \frac{1}{3}, -3\right),\end{aligned}$$

that is, the diffuser can fill half of the domain and has two flows; one on the left and one on the right. The flow on the right is one third the length, and has three times the flow rate as the one on the right. A sketch of the diffuser is shown in figure 3.5.

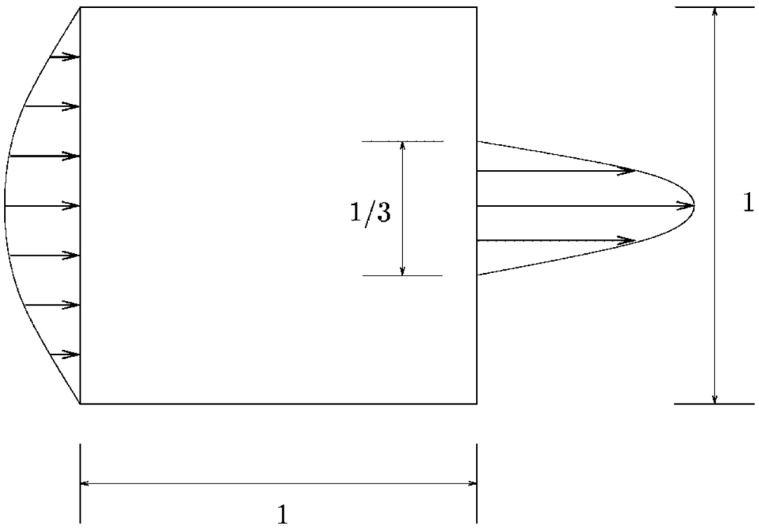


Figure 3.5: Figure of the diffuser we are going to optimize. From figure 4 in [8].

The second example is a pipe bend that is one unit wide and one unit tall. We are going to use

$$\begin{aligned}\gamma &= \frac{2\pi}{25},\\ \boldsymbol{g}(x,y) &= \boldsymbol{v}_{\text{left}}\left(x,y;\ \frac{4}{5},\frac{1}{5},1\right) + \boldsymbol{v}_{\text{right}}\left(x,y;\ \frac{4}{5},\frac{1}{5},-1\right),\end{aligned}$$

that is, the bend can fill about 25% of the domain and has two flows; one on the top of the left side and one on the right of the bottom side. A sketch of the pipe bend is shown in figure 3.6.

The third example is a twin pipe that is 1.5 units wide and one unit tall. We are going to use

$$\begin{aligned}\gamma &= \frac{1}{3},\\ \boldsymbol{g}(x,y) &= \boldsymbol{v}_{\text{left}}\left(x,y;\ \frac{1}{4},\frac{1}{6},1\right) + \boldsymbol{v}_{\text{left}}\left(x,y;\ \frac{3}{4},\frac{1}{6},1\right)\\ &\quad + \boldsymbol{v}_{\text{right}}\left(x,y;\ \frac{1}{4},\frac{1}{6},-1\right) + \boldsymbol{v}_{\text{right}}\left(x,y;\ \frac{3}{4},\frac{1}{6},-1\right),\end{aligned}$$

that is, the twin pipe can fill about 33% of the domain and has four flows; two on the left and two on the right. For this example, a penalization of $q = 0.1$ will not give the optimal solution, so we will use the approach described in [8], where we first solve the optimization problem for $q = 0.01$, giving a solution that is diffuse but has the correct shape, and then use that result as the initial value for a new run with $q = 0.1$. A sketch of the twin pipe is shown in figure 3.7.

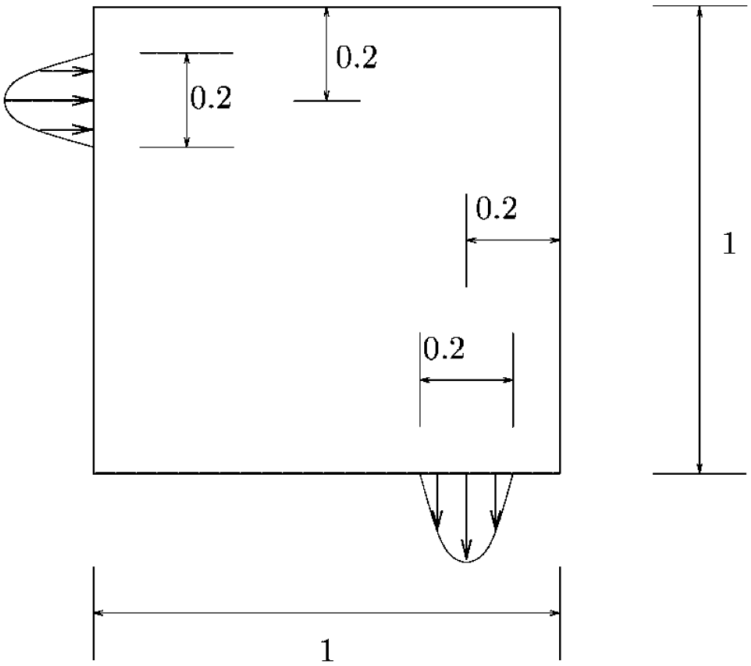


Figure 3.6: Figure of the pipe bend we are going to optimize. From figure 6 in [8].

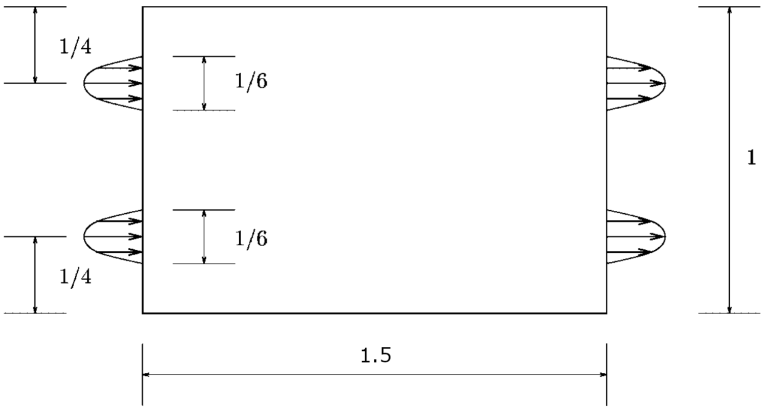


Figure 3.7: Figure of the twin pipe we are going to optimize. From figure 10 in [8].

Chapter 4

Methods

Topology optimization can broadly be divided into two parts; the first part solves the various PDEs required to calculate the objective function, and the second part iteratively optimizes the objective function. We want to compare two approaches for solving PDEs, so it is important that the optimization is performed using the same algorithm for both methods. This chapter will first explain the objective optimizer we are using, and then explain the two PDE solvers we are going to compare.

4.1 Entropic Mirror Descent

There exists many methods that can solve the type of constrained optimization problem we get with topology optimization. As the exact method is not the focus of this thesis, we are going to use a newly developed method called entropic mirror descent (EMD), which is an application of the proximal Galerkin method [25], described in algorithm 1. We chose this method as it is very simple to implement numerically. It is a latent space gradient descent algorithm, so instead of performing the descent on the density ρ directly, it instead operates on the transformed function ϑ using $\rho = \sigma(\vartheta)$, $\vartheta = \sigma^{-1}(\rho)$, where

$$\begin{aligned}\sigma(x) &= \frac{1}{1+e^{-x}},\\ \sigma^{-1}(x) &= -\log\Big(\frac{1}{x}-1\Big).\end{aligned}$$

The function σ is called the sigmoid function, and it is an increasing function with the domain $\mathbb{R}$ and the image $(0,1)$. This means that the transformation $\rho = \sigma(\vartheta)$ ensures that $0 \leq \rho \leq 1$. To obey the volume constraint, EMD corrects the gradient descent step

$$\vartheta^{k+1/2} = \vartheta^k - \alpha_k \nabla\phi\Big(\sigma\Big(\vartheta^k\Big)\Big),$$

where α_k is some step size, by adding a value $c \in \mathbb{R}$ that satisfies the equation

$$\int_\Omega \sigma\Big(\vartheta^{k+1/2} + c\Big)\,\mathrm{d}\boldsymbol{x} = \gamma|\Omega|.$$

This ensures that $\rho^{k+1} = \sigma\Big(\vartheta^{k+1/2} + c\Big)$ satisfies the volume constraint. Note that this volume correction ensures that $\int_\Omega \rho\,\mathrm{d}\boldsymbol{x} = \gamma|\Omega|$, not just $\int_\Omega \rho\,\mathrm{d}\boldsymbol{x} \leq \gamma|\Omega|$ which is how the constraint is typically formulated.

Algorithm 1 Entropic mirror descent

Input : Initial density distribution $\rho^0 \in L^2(\Omega)$, sequence of step sizes $\alpha_k > 0$, increment tolerance `itol` > 0, and normalized tolerance `ntol` > 0.
Output: Optimized material density $\rho = \sigma\left(\vartheta^k\right)$.

Initialize k = 0, $\vartheta^0 = \sigma^{-1}(\rho^0)$.
while $\left|\left|\sigma\left(\vartheta^k\right) - \sigma\left(\vartheta^{k-1}\right)\right|\right|_{L_2(\Omega)} > \min\{\alpha_k \texttt{ntol}, \texttt{itol}\}$ **do**
  `// Latent space gradient descent`
  Assign $\vartheta^{k+1/2} \leftarrow \vartheta^k - \alpha_k \nabla\phi\left(\sigma\left(\vartheta^k\right)\right)$
  `// Compute Lagrange multiplier`
  Solve for $c \in \mathbb{R}$ such that $\int_\Omega \sigma\left(\vartheta^{k+1/2} + c\right) \mathrm{d}\boldsymbol{x} = \gamma|\Omega|$.
  `// Latent space feasibility correction`
  Assign $\vartheta^{k+1} \leftarrow \vartheta^{k+1/2} + c$.
  Assign $k \leftarrow k + 1$.
end while

The equation defining c is a simple one dimensional root finding problem, which can easily be solved using Newton's method. This is done by iteratively calculating

$$c_{m+1} = c_m - \frac{\int_\Omega \sigma\left(\vartheta^{k+1/2} + c_m\right) \mathrm{d}\boldsymbol{x} - \gamma|\Omega|}{\int_\Omega \sigma'\left(\vartheta^{k+1/2} + c_m\right) \mathrm{d}\boldsymbol{x}}$$

until $|c_{m+1} - c_m| < \epsilon$, for some initial value c_0 and small value ϵ. Because $\sigma'(x) = \sigma(x)(1 - \sigma(x)) > 0 \ \forall x \in \mathbb{R}$, Newton's method should never hit a singularity. However, due to numerical inaccuracies, this is not true in practice, so a fallback algorithm is necessary in case Newton's method fails. For this, we used Brent's method, which is a combination of many methods, making it very reliable. Brent's method is more complicated than Newton's method, so a thorough explanation is outside the scope of this thesis; for more information see [10]. The important part is that Brent's method can find the root of a function f given two points a and b such that $f(a)$ and $f(b)$ have opposite signs. We do not know two such points a priori, so we developed a simple algorithm, shown in algorithm 2, that gives two values with opposite sign. The algorithm is not guaranteed to work for any function f, but for the volume correction we know that

$$\lim_{c\to\infty} \sigma\left(\vartheta^{k+1/2} + c\right) = 1 \text{ and } \lim_{c\to-\infty} \sigma\left(\vartheta^{k+1/2} + c\right) = 0,$$

which means the algorithm is guaranteed succeed in our case.

For the various parameters, we used $c_0 = 0$ and $\epsilon = 10^{-12}$ for Newton's method and $r_0 = 2$ for Brent's method.

The difficult part of EMD is calculating the gradient of our objective functions. From [25] we get that the gradient for linear elasticity is defined by the PDE

$$-\epsilon^2\nabla^2\tilde{w} + \tilde{w} = w \text{ on } \Omega, \ \nabla\tilde{w} \cdot \boldsymbol{n} = 0 \text{ on } \delta\Omega,$$

where

$$w = -r'(\tilde{\rho})\left(\lambda||\nabla \cdot \boldsymbol{u}||^2 + 2\mu||\boldsymbol{\varepsilon}||^2\right)$$

and $\tilde{w} = \nabla\phi(\rho)$. Note that this equation is just the Helmholtz filter applied to w. The gradient for Stokes flow is easier to calculate, as it does not require solving a PDE. It is

Algorithm 2 Brent bound finder

Input : Some function $f : \mathbb{R} \to \mathbb{R}$ and initial radius r_0.
Output: Two points a and b such that $a \leq b$ and $f(a)f(b) \leq 0$.

Initialize $r = r_0$.
Initialize $(a, b) = (-r, r)$
while $f(-r_0)f(r_0) > 0$ **do**
    Assign $r \leftarrow 2r$
    Assign $(a, b) \leftarrow (-r, r)$
end while

simply given by

$$\nabla\phi(\rho) = \frac{1}{2}r'(\rho)||\boldsymbol{u}||^2.$$

A sketch of the calculation for this result can be found in appendix B.

4.2 Finite Element Approach

4.2.1 The Finite Element Method

The FEM is a commonly used numerical method for solving PDEs that converts the original PDE into a system of algebraic equations. It approximates the true solution of the PDE by finding a solution in a simple finite dimensional subspace S_h of the full space of solutions $H^1(\Omega)$, where h is a discretization parameter. The discretization parameter is defined in such a way that the approximate solution converges to the true solution as $h \to 0$. The FEM consists of two main parts; constructing a partition of the domain Ω, called a mesh, and defining a basis $\{\varphi_1, \ldots, \varphi_k\}$ for the subspace S_h of functions defined on the mesh. Assuming that the approximation $\hat{\boldsymbol{u}}$ of the true solution $\boldsymbol{u}$ is an element of S_h, we can write

$$\hat{\boldsymbol{u}} = \sum_{i=1}^{k} u_i\varphi_i,$$

where $u_i \in \mathbb{R}$ are some unknown coefficients. The basis functions can be any set of linearly independent functions, such as piecewise continuous polynomials. A property we want the basis functions to have is that they have a small support, that is, they are 0 for most of Ω. This is useful as it means the value of $\boldsymbol{v} \in S_h$ at any point is determined by just a few coefficients, which makes the resulting linear system sparse. It is however important that the combined support of all the basis functions is the entire domain. If this is not the case for some point $\bar{\boldsymbol{x}} \in \Omega$, then $\boldsymbol{v}(\bar{\boldsymbol{x}}) = 0$ for all $\boldsymbol{v} \in S_h$.

The FEM in 1D

To explain how the FEM works, we will start with the simplest case. One of the simplest sets of basis functions are the piecewise linear polynomials defined on a one dimensional domain. For the domain $\Omega = [a, b]$, the only possible mesh is $\{[x_i, x_{i+1}]\}_{i=1}^{k-1}$ for some points $a = x_1 < \cdots < x_k = b$. We define $h_i = x_i - x_{i-1}$ and $h = \max\{h_i\}$. The basis functions are defined by the property $\varphi_i(x_j) = \delta_{ij}$ for $i, j \in \{1, \ldots, k\}$. This means that φ_i is zero in $[a, x_{i-1}]$, increases linearly to 1 in $[x_{i-1}, x_i]$, decreases linearly to 0 in $[x_i, x_{i+1}]$, and is zero in $[x_{i+1}, b]$. This gives a triangular shape that looks like a hat,

so these functions are commonly referred to as hat functions. It is easy to see that φ_i has the support $[x_{i-1}, x_{i+1}]$. The boundary functions φ_1 and φ_k are a bit different as x_0 and x_{k+1} are not defined, so they have a smaller support, being $[x_1, x_2]$ and $[x_{k-1}, x_k]$ respectively.

To explain how the FEM is used to transform the PDE into a linear equation, we will work through a simple example solving the boundary value problem

$$f''(x) = -8,\ f(0) = f(1) = 0,$$

which has the analytical solution $f(x) = 4x(1-x)$. Our domain is $\Omega = [0, 1]$, and we use piecewise linear polynomials. We define $\hat{f}$ to be the numerical approximation of the solution f, so we can write $\hat{f} = \sum_{i=2}^{k-1} \hat{f}_i \varphi_i$ for some unknown values $\hat{f}_2, \ldots, \hat{f}_{k-1}$. Note that $\hat{f}_1 = \hat{f}_k = 0$ due to the boundary conditions, so we can ignore them. We start by calculating the weak form of the differential equation. As we have Dirichlet boundary conditions, the test function v is zero at 0 and 1, giving us

$$\int_0^1 f'v' \,\mathrm{d}x = \int_0^1 -8v \,\mathrm{d}x\,.$$

We also assume that $v \in S_h$, meaning we can write $v = \sum_{i=2}^{k-1} v_i \varphi_i$ for some coefficients v_i. With this, we can rewrite the weak form:

$$\begin{aligned}
&\int_0^1 \left(\sum_{j=2}^{k-1} v_j \varphi_j\right)' \left(\sum_{i=2}^{k-1} \hat{f}_i \varphi_i\right)' \mathrm{d}x = \int_0^1 -8 \left(\sum_{j=2}^{k-1} v_j \varphi_j\right) \mathrm{d}x \\
\implies &\int_0^1 \sum_{j=2}^{k-1} v_j \varphi_j' \sum_{i=2}^{k-1} \hat{f}_i \varphi_i' \,\mathrm{d}x = \int_0^1 \sum_{j=2}^{k-1} v_j(-8\varphi_j) \,\mathrm{d}x \\
\implies &\sum_{j=2}^{k-1} v_j \sum_{i=2}^{k-1} \int_0^1 \varphi_j' \varphi_i' \,\mathrm{d}x\, \hat{f}_i = \sum_{j=2}^{k-1} v_j \int_0^1 -8\varphi_j \,\mathrm{d}x
\end{aligned}$$

If we define $\boldsymbol{v} = (v_2, \ldots, v_{k-1})$, $\hat{\boldsymbol{f}} = (\hat{f}_2, \ldots, \hat{f}_{k-1})$,

$$\begin{aligned}
\boldsymbol{b} = (b_2, \ldots, b_{k-1}),\ b_i &= \int_0^1 -8\varphi_i \,\mathrm{d}x\,, \\
A = \{a_{ij}\}_{i,j=2}^{k-1},\ a_{ij} &= \int_0^1 \varphi_i' \varphi_j' \,\mathrm{d}x\,,
\end{aligned}$$

this is equivalent to

$$\boldsymbol{v}^T A \hat{\boldsymbol{f}} = \boldsymbol{v}^T \boldsymbol{b}.$$

All the values in $\boldsymbol{v}$ are arbitrary, so for the above equation to hold, $A\hat{\boldsymbol{f}} = \boldsymbol{b}$ must hold. Solving this equation gives us the $\hat{f}_i$ values needed to compute $\hat{f}$. Figure 4.1 compares the FEM approximation to the analytical solution for $k = 5$, and shows the basis functions.

In this case the integrals needed to compute a_{ij} and b_i are solvable analytically, but in general a numerical integration method must be used. We can now see why we want the basis functions to have a small support; we only need to calculate a_{ii}, $a_{i+1,i}$ and $a_{i,i+1}$, as φ_i and φ_j lack a common support when $|i - j| > 1$. Calculating A therefore only requires $3k - 8$ integrals, instead of the full $(k-2)^2$. For large k, this results in a

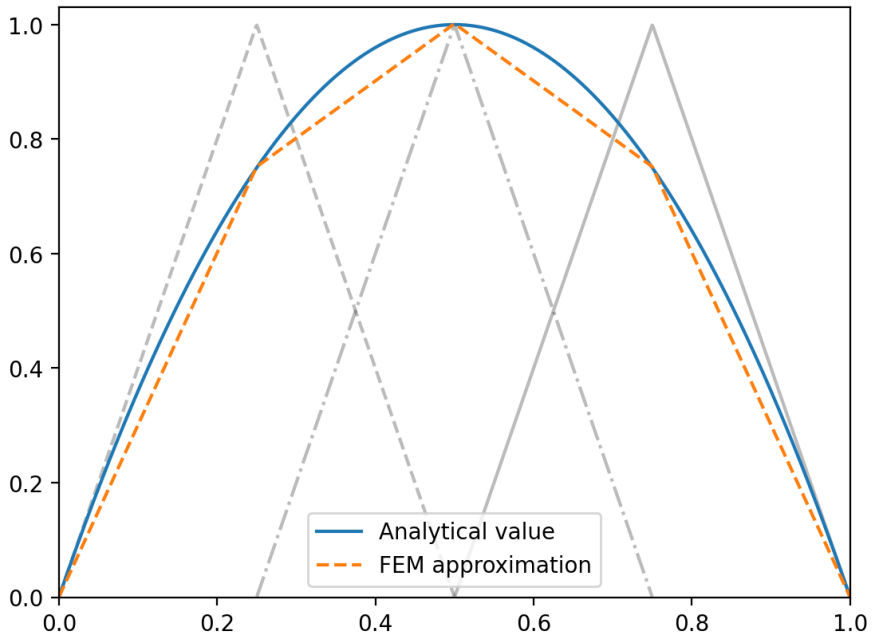


Figure 4.1: A figure comparing the analytical solution of the differential equation $f''(x) = -8,\ f(0) = f(1) = 0$ with a numerical approximation using the finite element method. The solid blue line is the analytical solution, and the striped orange line is the numerical approximation. The light gray triangles show the nodal basis functions, with various line styles to differentiate the different functions.

very sparse matrix, so both storing the matrix and solving the equation requires special consideration. In the general case of solving some PDE $\mathcal{D}\boldsymbol{u} = \boldsymbol{f}$, it is common to define the functions $a(\boldsymbol{u}, \boldsymbol{v})$ and $l(\boldsymbol{v})$ such that the weak form of the PDE is $a(\boldsymbol{u}, \boldsymbol{v}) = l(\boldsymbol{v})$. We still get the linear equation $A\hat{\boldsymbol{u}} = \boldsymbol{b}$, but now $b_i = l(\varphi_i)$, $a_{ij} = a(\varphi_i, \varphi_j)$. If we had used Neumann boundary conditions instead of Dirichlet ones, we would get an extra term in the weak form, and we would have to include the edge values at index 1 and k, but the computation would be otherwise unchanged.

The FEM in 2D

Unlike in one dimension, we have much more freedom in how we construct a 2D mesh. We will only focus on one of the most common types of 2D mesh, called a triangulation. A triangulation is a set of triangles $\{K_i\}$ that covers the domain, where the intersection of two triangles is either an edge, a corner, or the empty set. That is, the triangles do not cover each other. Another condition is that no corner can lie on the edge of another triangle, corners can only lie on corners. We define h_i to be the length of the longest edge in K_i, and $h = \max\{h_i\}$. For a rectangular mesh, constructing a triangulation is easy. We notice that if we use right-angled isosceles triangles, two of them combined will form a square of some size λ, and we get $h = h_i = \sqrt{2}\lambda$. For $\Omega = [0, w] \times [0, h]$, we define $l_{\min} = \min\{w, h\}$, $l_{\max} = \max\{w, h\}$ and $r = l_{\max}/l_{\min}$. We introduce the discretization parameter $N = l_{\min}/\lambda$, that is, N is the number of elements along the shortest side of the domain. We want N to be an integer, so it makes more sense to define h with respect to N, giving us $h = \sqrt{2}l_{\min}/N$. Note that this only works if $rN \in \mathbb{N}$, that is, if we can fit a whole number of elements along the longest side as well. The full triangulation will consist of $2rN^2$ triangles, and an example of such a triangulation with $r = 1.5$ and

$N = 4$ can be seen in figure 4.2. This is the kind of triangulation we will be using.

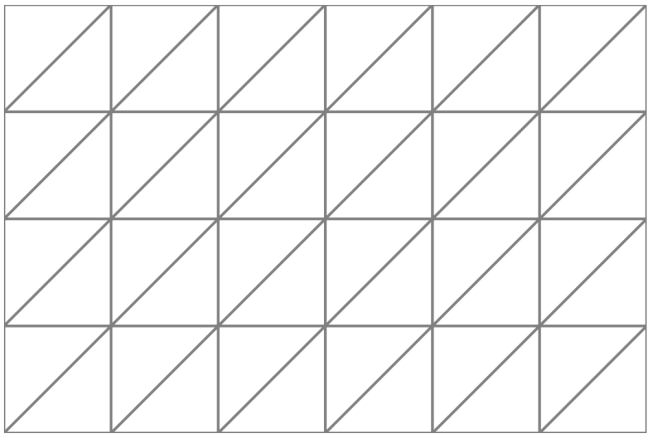


Figure 4.2: Figure depicting the type of triangulation of a 2D rectangular domain we will use when solving PDEs with the finite element method.

We must now discuss how to construct basis functions for a triangulation. Once again, the most common kind is piecewise continuous polynomials, so that is what we will focus on. A 2D linear polynomial is given by $p(x, y) = a + bx + cy$, that is, it has three degrees of freedom. A triangle has three corners, so it makes sense to define the polynomial within a triangle by its values at the corners. This results in a similar definition of the basis functions as in the 1D case; if we define $\boldsymbol{x_i}$ to be the coordinate of corner i, $i = 1, \ldots, (N + 1)(rN + 1)$, then φ_i is the piecewise linear polynomial such that $\varphi_i(\boldsymbol{x_j}) = \delta_{ij}$. This results in a two-dimensional hat function, depicted in figure 4.3. For higher dimensional polynomials, points along the edges of the triangle must also be used. With the basis functions defined, we have our subspace S_h, and we get that any function $v \in S_h$ can be written on the form

$$v = \sum_{i=1}^{(N+1)(rN+1)} v_i \varphi_i$$

for some coefficients v_i. With this, we can proceed with method explained in the previous section. The only difference appears when trying to calculate the integrals involved in a_{ij} and b_i. Now the support of the basis functions is several triangles instead of two line segments, so the computation is more complicated. The rough idea is that you turn the full integral into a sum of integrals over each triangle, compute the integral for one triangle in the general case, and then transform that result to give the value for each triangle integral. With this, the full integral becomes a sum over known values. For a more thorough example of how this is done see [27].

Implementing the FEM from scratch would require a significant effort. Luckily, there are many finite element frameworks that handle all the hard parts of for you, such as the popular python package FEniCS [33] which we are going to use. For the design function ρ and the latent variable ϑ, we will use piecewise linear polynomials. This means we cannot represent a sharp boundary exactly, but we can get arbitrarily close by using a finer mesh. For the linear elasticity examples, we will use piecewise quadratic polynomials for the displacements, and for Stokes flow, we will use piecewise linear polynomials for the pressure, and piecewise quadratic polynomials for the fluid velocities. This is referred to as Taylor-Hood finite elements, which must be used to guarantee that a solution exists [12].

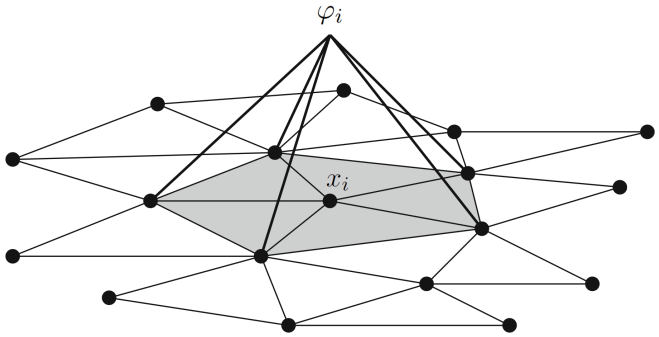


Figure 4.3: Figure depicting the two-dimensional hat function φ_i on a triangular mesh. From figure 3.4 in [27], with some small modifications.

4.2.2 Linear Algebra Solvers

The FEM turns solving a linear PDE into solving a system of linear equations, but how do you solve such a system? There are two main solver categories; direct methods and iterative methods. Iterative methods are in general used for very large systems, that is, when the matrix A is very large, but the downside is that they do not give the exact solution. The direct methods are better when A is smaller, and they do give an exact answer. As computers cannot represent numbers with infinite precision, numerical implementations of direct methods cannot give the exact answer, but they will in general be as close as possible. The matrices we are working with are not that large as we are using two-dimensional problems, so we decided to use a direct method. Most direct methods are based on Gaussian elimination, a procedure that works by manipulating the system in ways that do not change the solution. There are three important manipulations that preserve the solution, namely switching two rows or columns, multiplying any row by a constant value, and adding any row or linear multiple of a row to another row. A simple linear solver based on Gaussian elimination is described in algorithm 3, based on an algorithm from [20]. This algorithm uses the invariance of switching two rows to improve the stability of the algorithm using pivoting. As mentioned, numerical implementations of direct methods will often be close to the true solution, but there are cases where a rudimentary implementation will give a totally wrong answer. This happens when one of the diagonal entries is close to the limits of numerical accuracy. Pivoting is then a method to ensure large diagonal entries by either only swapping rows, called partial pivoting, or swapping both rows and columns, called full pivoting.

As stated previously, the matrix A from the FEM is very sparse, so storing and operating on its zeros would be inefficient. Therefore, special algorithms have been developed to solve linear equations involving sparse matrices. One such algorithm is called the multifrontal method, a method based on Gaussian elimination that organizes the factorization of A in such a way that the full factorization is performed through partial factorizations of a sequence of dense and small submatrices [3]. The full algorithm is complicated and describing it is outside the scope of this thesis; for more information see [16]. We used a popular implementation of the multifrontal method called MUMPS [43], which is one of the linear solvers accessible through FEniCS.

Algorithm 3 Gaussian elimination with partial pivoting

Input : Matrix $A \in \mathbb{R}^{n \times n}$ and vector $\boldsymbol{b} \in \mathbb{R}^n$.
Output: Solution vector $\boldsymbol{x} = (x_1, \ldots, x_n)$ to the linear equation $A\boldsymbol{x} = \boldsymbol{b}$.

Initialize $E = \begin{bmatrix} A & \boldsymbol{b} \end{bmatrix}$.
for $i = 1, \ldots, n$ **do**
  Find p, where p is the index of the largest absolute value in the i-th column at or below the i-th row.
  if $p \neq i$ **then**
    Exchange row i with row p in E.
  end if
  for $j = i + 1, \ldots, n$ **do**
    Assign $E_j \leftarrow E_j - \frac{a_{ji}}{a_{ii}} E_i$
  end for
end for
for $i = n, \ldots, 1$ **do**
  Assign $x_i \leftarrow \frac{1}{E_{ii}} \left(E_{i,n+1} - \sum_{j=i+1}^{n} E_{ij} x_j \right)$
end for

4.3 Neural Network Approach

4.3.1 Deep Neural Networks

A neural network (NN) is a function that takes in an input $\boldsymbol{x} \in \mathbb{R}^n$ and returns an output $\boldsymbol{y} \in \mathbb{R}^m$. The neural network repeatedly applies linear and non-linear transformations to the input. Each linear transformation has many parameters, which can be tuned such that the network approximates any function [15]. We are going to use a deep neural network, depicted in figure 4.4, which has many fully connected layers. We define the number of values in layer l to be n_l, where $l = 0, \ldots, L + 1$ for some number of hidden layers L. For the input and output layers we have $n_0 = n$ and $n_{L+1} = m$. The output of layer l is

$$\boldsymbol{y}^l = \sigma^l \left(\boldsymbol{W}^l \boldsymbol{y}^{l-1} + \boldsymbol{b}^l \right),$$

where $\boldsymbol{W}^l \in \mathbb{R}^{n_l \times n_{l-1}}$, $\boldsymbol{b}^l \in \mathbb{R}^{n_l}$, and σ is some non-linear function, called an activation function. Note here that $\boldsymbol{y}^0 = \boldsymbol{x}$ and $\boldsymbol{y}^{L+1} = \boldsymbol{y}$. The matrices $\boldsymbol{W}^l$, called the weights, and the vectors $\boldsymbol{b}^l$, called the biases, contain the optimization parameters. We define

$$\boldsymbol{\theta} = \left\{ \boldsymbol{W}^1, \ldots, \boldsymbol{W}^{L+1}, \boldsymbol{b}^1, \ldots, \boldsymbol{b}^{L+1} \right\}$$

and write the neural network as the function $\boldsymbol{u}_{NN}(\boldsymbol{x}; \boldsymbol{\theta})$.

To tune the parameters, called training the neural network, we typically use known correct outputs, called training data. If we have a set

$$\boldsymbol{X} = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), \ldots, (\boldsymbol{x}_k, \boldsymbol{y}_k)\}$$

of input output pairs, we can construct a cost function

$$C(\boldsymbol{\theta}) = \frac{1}{k} \sum_{i=1}^{k} ||\boldsymbol{u}_{NN}(\boldsymbol{x}_i; \boldsymbol{\theta}) - \boldsymbol{y}_i||^2$$

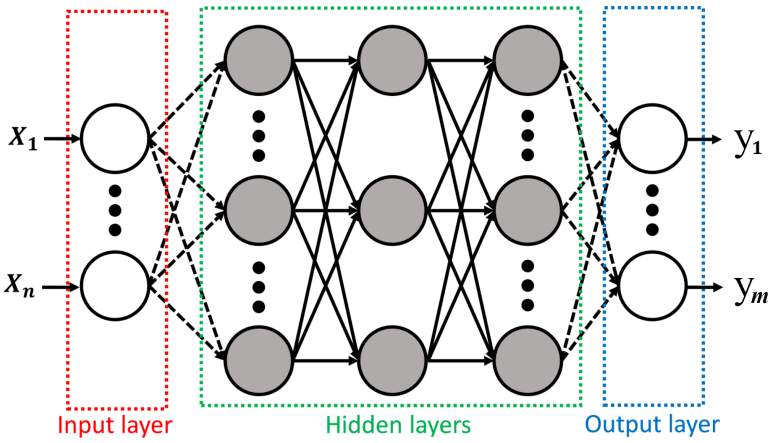


Figure 4.4: Figure depicting a fully connected deep neural network. From figure 1 in [24], with some small modifications.

which calculates the mean squared error. Training the network is then done by calculating the gradient $\nabla_{\theta} C$, and then using some form of gradient descent algorithm to minimize the cost. The gradient is typically calculated using backpropagation, a type of automatic differentiation that uses the chain rule to calculate the gradient one layer at a time. There exists tools made specifically for training neural networks, like the python package PyTorch [21] that we will use.

When training a neural network, the network might overfit the training data, which means it will give good results on the training data, but terrible results for other inputs. Figure 4.5 shows an example of overfitting when approximating a sinusoidal function with a high-degree polynomial. To detect overfitting, it is common to split the data into two sets, one containing training data and one containing test data. The network gets trained on the training data, and then tested on the test data to check if it performs well on unseen data [23].

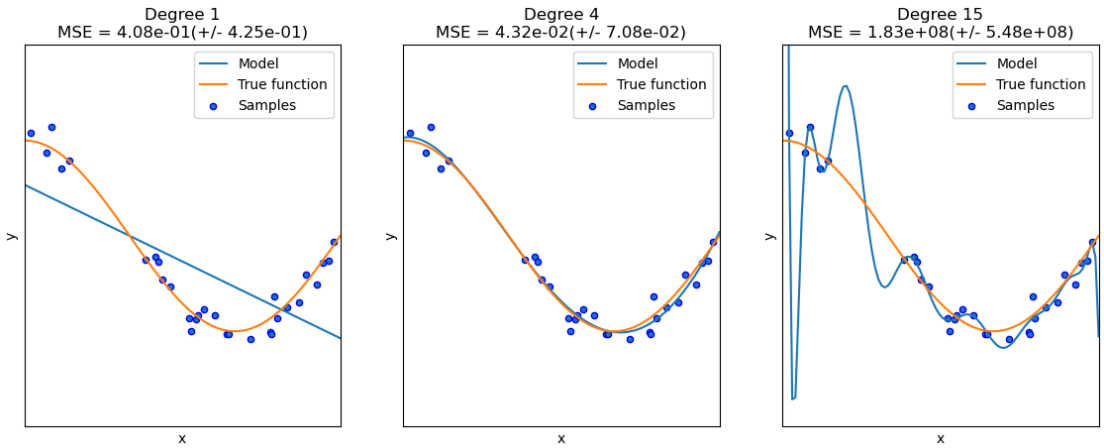


Figure 4.5: Figure depicting underfitting and overfitting of polynomial approximations, the blue curves, to a sinusoidal function, the orange curve. Downloaded from https://compphysics.github.io/MachineLearning/doc/LectureNotes/_build/html/_images/chapter3_64_1.png.

The architecture of a neural network is defined by a number of hyperparameters, such as the number of layers or number of nodes per layer. To get the best result, you

therefore need to do hyperparameter optimization. While it is possible to find decent hyperparameters manually, a better approach is to use Bayesian optimization, which only tests promising candidates [7]. When testing many combinations of hyperparameters, there is a chance the hyperparameters overfit the test data. To avoid this, some data can be put aside into a set of validation data, which is used while optimizing hyperparameters [23]. This ensures that the test data is used only once, and therefore gives a good estimate for the network's performance on real world data.

4.3.2 Physics-Informed Neural Networks

When solving a PDE using a neural network, we cannot use the standard training data approach, as that would require us to have already solved the PDE. We must instead use a form of the PDE as the cost function directly. One possibility is to use a physics-informed neural network (PINN) [34], where you use $\mathcal{D}\boldsymbol{u}_{NN} - \boldsymbol{f}$ to represent how close $\boldsymbol{u}_{NN}$ is to solving the PDE $\mathcal{D}\boldsymbol{u} = \boldsymbol{f}$. The cost function is then

$$C(\boldsymbol{\theta}) = \frac{1}{k}\sum_{i=1}^{k} ||\mathcal{D}\boldsymbol{u}_{NN}(\boldsymbol{x}_i;\boldsymbol{\theta}) - \boldsymbol{f}(\boldsymbol{x}_i)||^2,$$

where $x_i \in \Omega$ are some training points, typically evenly distributed in Ω.

There are two ways of dealing with boundary conditions. If you require that $\boldsymbol{u} = \boldsymbol{g}$ on $\Gamma_D \subset \delta\Omega$, then you can either penalize values not equal to $\boldsymbol{g}$, or enforce the boundary condition via additive decomposition. For the first approach, we divide the set of training points $\boldsymbol{X}$ into two sets, $\boldsymbol{X}^c$ and $\boldsymbol{X}^b$ containing points inside and on the boundary respectively. The modified cost function is then

$$C(\boldsymbol{\theta}) = \frac{1}{k^c}\sum_{i=1}^{k^c} ||\mathcal{D}\boldsymbol{u}_{NN}(\boldsymbol{x}_i^c;\boldsymbol{\theta}) - \boldsymbol{f}(\boldsymbol{x}_i^c)||^2 + \frac{\tau}{k^b}\sum_{i=1}^{k^b} ||\boldsymbol{u}_{NN}(\boldsymbol{x}_i^b;\boldsymbol{\theta}) - \boldsymbol{g}(\boldsymbol{x}_i^b)||^2,$$

where τ denotes a penalty parameter.

The other approach, additive decomposition, avoids defining an additional hyperparameter. Instead, we can label the neural network function $\tilde{\boldsymbol{u}}_{NN}$, and define

$$\boldsymbol{u}_{NN} = \boldsymbol{m} \odot \tilde{\boldsymbol{u}}_{NN} + \boldsymbol{g},$$

where $\boldsymbol{m}(\boldsymbol{x}) = 0\ \forall \boldsymbol{x} \in \Gamma_D$ and $\odot$ denotes element-wise multiplication. With this method, $\boldsymbol{u}_{NN}$ is guaranteed to satisfy the Dirichlet boundary conditions, so this is the approach we will use.

4.3.3 The Deep Energy Method

PINNs use the strong form of a PDE, which might have high order differential operators. If the PDE is a variational problem, we could instead use the energy functional as our cost function, as we know finding its minimum is the same as solving the PDE. This results in what is called a variational physics-informed neural network (VPINN) [26], and is called the deep energy method (DEM) [36]. Another benefit of this approach is that it makes implementing Neumann boundary conditions trivial, as those are simply added as an additional term in the energy functional.

As there are many ways of constructing neural networks, we will base our implementation on the one described by the paper *Deep energy method in topology optimization applications* [24], which used the DEM for linear elasticity topology

optimization. The NN architecture used in the paper is a simple fully connected feed-forward network, where the Fourier transform is applied to the input features to transform them to the frequency domain. The use of such Fourier features has been shown to improve the accuracy of PINNs [44]. The NN has the same number of neurons in each hidden layer, and always uses the same activation function. All the biases are initialized to zero, and the weights are initialized with a normal distribution with a constant standard deviation. The Fourier features are also initialized with a normal distribution, but with a separate standard deviation. This means the neural network has seven hyperparameters, namely the step size and maximum number of iterations for the parameter optimizer, the number of hidden layers, the number of neurons in a hidden layer, the activation function, and the two standard deviations. These hyperparameters were optimized using the python package `hyperopt` [7].

To evaluate the energy functional, Gauss quadrature integration was used, which means the training data formed quadrilateral isoparametric finite elements. The domain $\Omega = [0, w] \times [0, h]$ was divided into a mesh with rectangular faces, with N_x rectangles in the x-direction and N_y rectangles in the y-direction. The neural network was then evaluated at each node in the mesh, the density was defined at each face in the mesh, and the gradient of $\boldsymbol{u}$ was calculated using the gradients of the finite element shape functions. No testing or validation data was used, so there is no guarantee that the network will not overfit. This is fine, the results from the training data can safely be used, but it means the neural network cannot be evaluated on any other points. To see why this is true, we can look at the degree 15 case in figure 4.5. There, we see that evaluating an overfit model on the sampled points gives values close to the true function, while evaluating it on any other points can give a totally wrong result. The only consequence of the lack of testing and validation data is therefore a limited resolution. In our implementation, we only used square faces to match the FEM, which means the mesh for the DEM is also parameterized by the number of squares along the shortest side of the domain N. Note that the role N plays here is very different to the role it plays for the FEM. Unlike the FEM, there is no guarantee that the approximate solution approaches the true solution as $N \to \infty$. Instead, increasing N has two main benefits; it improves the accuracy of the energy functional evaluation, and it increases the resolution of the solution. Increasing N also has a potential drawback; as the neural network is evaluated at more points, it must give a good result for more input values, making training the network more difficult. A neural network can in theory approximate any function [15], but this requires that the number of parameters in the network goes to infinity, not just the amount of training data.

As the DEM is a finite dimensional method, it cannot use the Helmholtz filter to filter ρ. Instead, He et al. used a filter matrix F defined as follows

$$F_{ij} = \frac{q_{ij}}{\sum_{k=1}^{N} q_{ik}},$$
$$q_{ij} = \max(R - ||\boldsymbol{x}_i - \boldsymbol{x}_j||),$$

where R is some filter radius. They then filtered rho using $\tilde{\rho} = F\rho$, and filtered the objective gradient by multiplying it with F^T. This filter is the discretized form of a convolution that can also be defined implicitly as a solution of the Helmholtz PDE [28]. This means that He et al. used a discretized but otherwise equivalent filter to the one we have introduced. The only difference is that the filter radius is different, but from [28] we get the conversion factor $R = 2\sqrt{3}\epsilon$. This is the reason why the filter radius for

the bridge and short cantilever in section 3.1.3 is defined as such; He et al. used the filter radius $R = 0.25$, which is equivalent to $\epsilon = 0.25/2\sqrt{3}$.

To train the neural network, the L-BFGS optimization algorithm was used, which is a version of the BFGS algorithm that uses a limited amount of computer memory. The BFGS algorithm is a quasi-Newton method, that is, a method that uses Newton's method with some approximation of the true hessian. It can be written as

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha_k B_k^{-1} \nabla f(\boldsymbol{x}_k),$$

where α_k is some step size, f is the function you want to minimize, and B_k is an approximation of the hessian matrix of f at $\boldsymbol{x}_k$, defined by the equation

$$B_{k+1}(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) = \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k).$$

A more efficient formulation can be defined that directly finds B_{k+1}^{-1} using B_k^{-1} [19], described in algorithm 4.

Algorithm 4 Broyden-Fletcher-Goldfarb-Shanno algorithm

Input : Function $f : \mathbb{R}^n \to \mathbb{R}$, initial guess $\boldsymbol{x}_0 \in \mathbb{R}^n$, and approximate inverted Hessian matrix $B^{-1} \in \mathbb{R}^{n \times n}$.
Output: Approximate minimum $\boldsymbol{x}_k$ of f.

Initialize $k = 0$.
while stopping criterion not met **do**
  $\boldsymbol{p}_k \leftarrow -B_k^{-1} \nabla f(\boldsymbol{x}_k)$.
  Find a step size α_k that approximates $\underset{\alpha}{\operatorname{argmin}}\, f(\boldsymbol{x}_k + \alpha \boldsymbol{p}_k)$.
  $\boldsymbol{s}_k \leftarrow \alpha_k \boldsymbol{p}_k$.
  $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \boldsymbol{s}_k$.
  $\boldsymbol{y}_k \leftarrow \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k)$.
  $B_{k+1}^{-1} \leftarrow B_k^{-1} + \left(1 + \frac{\boldsymbol{y}_k^T B_k^{-1} \boldsymbol{y}_k}{\boldsymbol{s}_k^T \boldsymbol{y}_k}\right) \frac{\boldsymbol{s}_k \boldsymbol{s}_k^T}{\boldsymbol{s}_k^T \boldsymbol{y}_k} - \frac{\boldsymbol{s}_k \boldsymbol{y}_k^T B_k^{-1} + B_k^{-1} \boldsymbol{y}_k \boldsymbol{s}_k^T}{\boldsymbol{s}_k^T \boldsymbol{y}_k}$
  $k \leftarrow k + 1$.
end while

The stopping criterion for the iteration is complicated, so we must first introduce some notation. Let $\Psi = \{\psi_1, \ldots, \psi_n\}$ be a set of values from the energy functional evaluated at the output of the neural network such that ψ_i is the value for iteration i. Further, let M be some positive integer. As long as $n < 2M$, the iteration always continues. Otherwise, the quantities

$$\mu_1 = \frac{1}{M} \sum_{i=n+1-2M}^{n-M} \psi_i, \quad \mu_2 = \frac{1}{M} \sum_{i=n+1-M}^{n} \psi_i$$

are calculated. The iteration is then stopped if either $|\mu_2| < 10^{-6}$ or $\frac{|\mu_1 - \mu_2|}{|\mu_2|} < 5 \cdot 10^{-5}$. This means the iteration stops if the values are approaching zero, or if the difference between consecutive iterations is small, with the average being used to reduce noise. The first condition works for a PINN where the minimum cost is always zero, but the minimum of the energy functional can be a non-zero value, so the first condition serves no purpose. In the code, $M = 10$ was used.

In the paper, He et al. optimized the design function using the method of moving asymptotes (MMA), a well known method that was developed for structural optimization

[42]. This method also relies on the gradient of the design function, so replacing MMA with EMD in the source code provided is trivial.

The objective function used in the paper seems to be different compared to the one we introduced. They defined the objective as

$$\phi(\tilde{\rho}; \boldsymbol{u}) = \frac{1}{2} \int_\Omega r(\tilde{\rho}) \boldsymbol{\sigma} : \boldsymbol{\varepsilon} \, \mathrm{d}\boldsymbol{x}$$

instead of

$$\phi(\tilde{\rho}; \boldsymbol{u}) = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{u} \, \mathrm{d}\boldsymbol{x} + \int_{\delta\Omega} \boldsymbol{t} \cdot \boldsymbol{u} \, \mathrm{d}s \, .$$

While these expressions look very different, they are almost the same. From the weak form of the linear elasticity equation, (3.5), we get that if $\boldsymbol{u}$ solves the equation, then

$$\int_\Omega r(\tilde{\rho}) \Big(\lambda ||\nabla \cdot \boldsymbol{u}||^2 + 2\mu ||\boldsymbol{\varepsilon}||^2 \Big) \, \mathrm{d}\boldsymbol{x} = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{u} \, \mathrm{d}\boldsymbol{x} + \int_{\delta\Omega} \boldsymbol{t} \cdot \boldsymbol{u} \, \mathrm{d}s \, .$$

This is because the equation holds for all $\boldsymbol{v}$, including $\boldsymbol{v} = \boldsymbol{u}$. Note that this only works because $\boldsymbol{u}$ and $\boldsymbol{v}$ come from the same function space. Using the definitions of $\boldsymbol{\sigma}$ and $\boldsymbol{\varepsilon}$, (3.4) and (3.3), we get

$$\boldsymbol{\sigma} : \boldsymbol{\varepsilon} = \lambda ||\nabla \cdot \boldsymbol{u}||^2 + 2\mu ||\boldsymbol{\varepsilon}||^2 .$$

Combining these two equations, it is easy to see that the objective they used is exactly half the objective we introduced. When minimizing a function, any constant factors do not matter, so the two objectives give identical results.

4.3.4 The DEM and Stokes Flow

Instead of using the full Stokes flow energy functional (3.12), we instead used the simpler zero-divergence functional (3.13). This has the benefit that the neural network does not need to output a pressure as well as fluid velocities, simplifying the training process. If needed, the pressure can be recovered using equation (3.9), but as the Stokes flow objective does not depend on the pressure, we have not done so. A drawback of this approach is that there is, as far as we know, no way of guaranteeing that the output of a neural network is divergence-free. This means we had to add the zero-divergence condition as a penalty to the cost function, giving us the functional

$$\psi(\boldsymbol{u}; \rho) = \frac{1}{2} \int_\Omega r(\rho) ||\boldsymbol{u}||^2 + \mu ||\nabla \boldsymbol{u}||^2 + \tau ||\nabla \cdot \boldsymbol{u}||^2 \, \mathrm{d}x \, ,$$

where τ is a penalization parameter, which acts as an additional hyperparameter that must be tuned.

4.3.5 Numerical Integration

Using Gauss quadrature integration and the gradients of the finite element shape functions, we can calculate any integral on the form $\int_\Omega \mathcal{L}(\boldsymbol{x}, \boldsymbol{u}, \nabla \boldsymbol{u}) \, \mathrm{d}\boldsymbol{x}$, which lets us evaluate all energy functionals. There are however two integrals which require special consideration, namely

$$\int_\Omega \boldsymbol{f} \cdot \boldsymbol{u} \, \mathrm{d}\boldsymbol{x} \text{ and } \int_{\Gamma_t} \boldsymbol{t} \cdot \boldsymbol{u} \, \mathrm{d}s \, .$$

While both of these integrals can be calculated using the aforementioned algorithm, that would not be the most accurate way of computing them. Γ_t is only a small segment of the full boundary, and $\boldsymbol{f}$ is a step function which is zero everywhere except in a small circular region. As we are integrating over such small areas, we could get large discretization errors if we are not careful.

Traction Integral

We are going to focus on the traction integral first. As it is an integral over a subset of the boundary of a 2D rectangle, we only need to develop an algorithm that works in one dimension. We are therefore going to develop an algorithm for the integral

$$\int_{\mathcal{D}_t} g(x)\,\mathrm{d}x\,,$$

where $\mathcal{D}_t = \left[c - \frac{l}{2}, c + \frac{l}{2}\right] \subset [0, w]$ for some real numbers c, l, and w. This integral is equivalent to

$$\int_{c-\frac{l}{2}}^{c+\frac{l}{2}} g(x)\,\mathrm{d}x\,,$$

and is easy to calculate numerically by discretizing $\mathcal{D}_t$; if we discretize it into N intervals, we can use the trapezoidal rule as follows

$$\begin{aligned}\int_{b_l}^{b_r} g(x)\,\mathrm{d}x &\approx \text{trapezoidal}_g\Big(\{b_l + i\Delta x\}_{i=0}^{N}\Big)\\ &= \sum_{i=0}^{N-1} \frac{g(b_l + i\Delta x) + g(b_l + (i+1)\Delta x)}{2}\Delta x,\end{aligned}$$

where $b_l = c - \frac{l}{2},\ b_r = c + \frac{l}{2},\ \Delta x = \frac{l}{N}$.

The problem is that we have instead discretized the entire domain $[0, w]$ into N intervals, giving us the points $X = \{i\Delta x\}_{i=0}^{N}$, where $\Delta x = \frac{w}{N}$. Restricting the integral is the same as finding the points $X_t \subset X$ where $\inf X_t \geq b_l$ and $\sup X_t \leq b_r$. This gives us

$$X_t = \{M_L\Delta x, \ldots, M_R\Delta x\},$$

where

$$M_L = \left\lceil \frac{b_l}{\Delta x} \right\rceil,\ M_R = \left\lfloor \frac{b_r}{\Delta x} \right\rfloor.$$

We could use the trapezoidal rule on X_t, that is what He et al. used, but that gives inaccurate results as we would end up integrating over a smaller region due to the small differences between the bounds of X_t and the actual bounds of the integral. These differences are

$$\epsilon_L = M_L\Delta x - b_l,\ \epsilon_R = b_r - M_R\Delta x.$$

To integrate over these regions we chose to use linear interpolation, which we can use as we know the values of $g((M_L - 1)\Delta x)$ and $g((M_R + 1)\Delta x)$. If we assume that g is linear between $g((M_L - 1)\Delta x)$ and $g(M_L\Delta x)$, we get

$$\begin{aligned}g(b_l) &= g((M_L - 1)\Delta x)\frac{M_L\Delta x - b_l}{\Delta x} + g(M_L\Delta x)\frac{b_l - (M_L - 1)\Delta x}{\Delta x}\\ &= g((M_L - 1)\Delta x)\frac{\epsilon_L}{\Delta x} + g(M_L\Delta x)\frac{\Delta x - \epsilon_L}{\Delta x},\end{aligned}$$

and similarly for the right side

$$g(b_r) = g(M_R\Delta x)\frac{\Delta x - \epsilon_R}{\Delta x} + g((M_R + 1)\Delta x)\frac{\epsilon_R}{\Delta x}.$$

If we use the trapezoidal rule with a non-uniform step size, we get the integration algorithm we will use for the traction integral

$$\begin{aligned}
\int_{b_l}^{b_r} g(x)\,\mathrm{d}x &\approx \frac{g(b_l)+g(M_L\Delta x)}{2}\epsilon_L + \sum_{i=M_L}^{M_R-1}\frac{g(i\Delta x)+g((i+1)\Delta x)}{2}\Delta x + \frac{g(M_R\Delta x)+g(b_r)}{2}\epsilon_R \\
&\approx \frac{g((M_L-1)\Delta x)\frac{\epsilon_L}{\Delta x} + g(M_L\Delta x)\frac{2\Delta x-\epsilon_L}{\Delta x}}{2}\epsilon_L + \frac{g(M_R\Delta x)\frac{2\Delta x-\epsilon_R}{\Delta x} + g((M_R+1)\Delta x)\frac{\epsilon_R}{\Delta x}}{2}\epsilon_R \\
&\quad + \sum_{i=M_L}^{M_R-1}\frac{g(i\Delta x)+g((i+1)\Delta x)}{2}\Delta x \\
&= \mathrm{lerptrapz}_g(\mathcal{D}_t, X_t).
\end{aligned}$$

To relate this to the traction integral, we first define

$$\boldsymbol{u}_{\text{side}}(t) = \begin{cases} \boldsymbol{u}(0,t) & \mid \text{side} = \text{left} \\ \boldsymbol{u}(w,t) & \mid \text{side} = \text{right} \\ \boldsymbol{u}(t,h) & \mid \text{side} = \text{top} \\ \boldsymbol{u}(t,0) & \mid \text{side} = \text{bottom} \end{cases},$$

$$X_{t,\text{side}} = \begin{cases} \left\{\left\lceil\frac{b_l}{\Delta x}\right\rceil\Delta x, \ldots, \left\lfloor\frac{b_r}{\Delta x}\right\rfloor\Delta x\right\} & \mid \text{side} = \text{top or bottom} \\ \left\{\left\lceil\frac{b_l}{\Delta y}\right\rceil\Delta y, \ldots, \left\lfloor\frac{b_r}{\Delta y}\right\rfloor\Delta y\right\} & \mid \text{side} = \text{left or right} \end{cases}.$$

With this, we get

$$\int_{\Gamma_{t,\text{side}}} \boldsymbol{t}\cdot\boldsymbol{u}\,\mathrm{d}s \approx \boldsymbol{t}\cdot\mathrm{lerptrapz}_{\boldsymbol{u}_{\text{side}}}(\mathcal{D}_t, X_{t,\text{side}}).$$

Body Force Integral

The approach we used previously does not work with the body force integral $\int_\Omega \boldsymbol{f}\cdot\boldsymbol{u}\,\mathrm{d}\boldsymbol{x}$ as we use

$$\boldsymbol{f}(x,y;\boldsymbol{c},r) = \begin{cases} \hat{\boldsymbol{f}} & \mid (x-c_x)^2+(y-c_y)^2 \le r^2 \\ \boldsymbol{0} & \mid \text{Otherwise} \end{cases},$$

which means the integral cannot be reinterpreted as one dimensional. We must therefore work with the full discretization

$$X = \{i\Delta x\}_{i=1}^{N_x} \times \{j\Delta y\}_{j=1}^{N_y}.$$

Trapezoidal integration over this domain gives

$$\begin{aligned}
\int_\Omega g(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} &\approx \sum_{i=1}^{N_x-1}\sum_{j=1}^{N_y-1}\frac{g_{i,j}+g_{i+1,j}+g_{i,j+1}+g_{i+1,j+1}}{4}\Delta x\Delta y \\
&= \sum_{i=1}^{N_x-1}\sum_{j=1}^{N_y-1}\bar{g}_{i,j}\Delta x\Delta y,
\end{aligned}$$

where $g_{i,j} = g(i\Delta x, j\Delta y)$. This sum can be interpreted as calculating the weight of a surface, where $\bar{g}_{i,j}$ represents the area density of the surface section $S_{i,j}$ with area

$A(S_{i,j}) = \Delta x \Delta y$. The total weight is then the sum of the mass of each section. With this interpretation, we can imagine a function C which gives the area of $S_{i,j}$ that is covered by the body force circle. This means that, if the circle covers a fraction p of the surface $S_{i,j}$, $C(S_{i,j}) = p\Delta x \Delta y$. The body force integral can then be computed by replacing $A(S_{i,j})$ with $C(S_{i,j})$, giving us

$$\int_\Omega \boldsymbol{f} \cdot \boldsymbol{u} \, \mathrm{d}\boldsymbol{x} \approx \hat{\boldsymbol{f}} \cdot \sum_{i=1}^{N_x - 1} \sum_{j=1}^{N_y - 1} \bar{\boldsymbol{u}}_{i,j} C(S_{i,j}).$$

The difficulty lies in calculating $C(S_{i,j}) = C_{i,j}$. We opted for a simple approach, where we found the points where the circle boundary intersects the grid lines, and then approximate the circle area with triangles. To get the intersections, we used the formula for the circle and solved it for x to get the horizontal intersections, and for y to get the vertical intersections. We classified the triangular approximations into three types, depending on which side the circle enters and exits the surface section. Type 2 is when the circle enters and leaves the same side, type 1 is when the circle enters and leaves different sides and the point closest to the intersection points is inside the circle, and type 3 is when the circle enters and leaves different sides and the point closest to the intersection points is outside the circle. Figure 4.6 depicts the three types. We can then

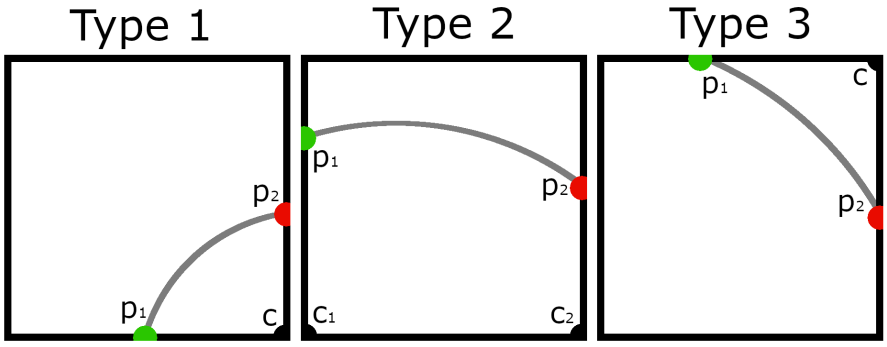


Figure 4.6: The three types of intersections you need to keep track of to approximate circle areas with triangles. The green point p_1 is where the circle enters the surface, and the red point p_2 is where the circle exits the surface. The c-points are the corner points needed to construct the triangles.

find the areas

$$\text{Type 1: } A = \frac{||\boldsymbol{p}_1 - \boldsymbol{c}||_2 ||\boldsymbol{p_2} - \boldsymbol{c}||_2}{2},$$
$$\text{Type 2: } A = ||\boldsymbol{c}_1 - \boldsymbol{c}_2||_2 \frac{||\boldsymbol{p}_1 - \boldsymbol{c}_1||_2 + ||\boldsymbol{p_2} - \boldsymbol{c}_2||_2}{2},$$
$$\text{Type 3: } A = \Delta x \Delta y - \frac{||\boldsymbol{p}_1 - \boldsymbol{c}||_2 ||\boldsymbol{p_2} - \boldsymbol{c}||_2}{2},$$

where $\boldsymbol{c}$ is the corner point closest to $\frac{\boldsymbol{p}_1 + \boldsymbol{p}_2}{2}$, $\boldsymbol{c}_1$ is the corner point inside the circle that is closest to $\boldsymbol{p}_1$, and $\boldsymbol{c}_2$ is the corner point inside the circle that is closest to $\boldsymbol{p}_2$. Figure 4.7 shows how the triangular approximations looks like.

This algorithm does not calculate areas that are fully inside or outside the circle, but that can easily be done using the corners of a surface section. If each corner is fully inside the circle, the entire surface section is covered, and if a surface section is neither inside nor on the boundary, it must be fully outside the circle. A nice property of

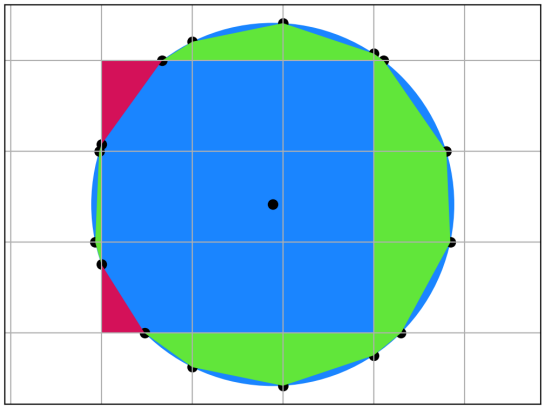


Figure 4.7: Figure depicting the triangular approximations of the area of a grid face a circle covers. Red triangles represent negative area. The black dots on the boundary show where the circle boundary intersects the grid, and the black dot inside the circle is located at its center.

this algorithm is that we can precompute $C_{i,j}$. We can also calculate the indices where $C_{i,j} \neq 0$, and use only those when calculating the integral. This avoids unnecessary computation when the circle is much smaller than the domain.

Chapter 5

Implementation

The code we wrote is divided into three folders, `src`, `FEM_src`, and `DEM_src`, containing the code for the EMD algorithm, the FEM-based solver, and the DEM-based solver respectively. The `src` folder also includes some utility functions for printing and timing, as well as the two penalizers.

5.1 `src`

As the DEM and the FEM are very different methods, the EMD implementation in `src` is an abstract base class, named `Solver`, which has many functions that are implemented separately in `DEM_src` and `FEM_src`, called `DEMSolver` and `FEMSolver` respectively. The Solver class takes in two arguments; the path to a design file and the discretization parameter N. A design file is a JSON file where details about the topology optimization problem is detailed, such as if the state equation is from linear elasticity or Stokes flow, what the domain size is, what the volume fraction is, and so on. It also details parameters specific to the state equation, such as what the traction is or where the boundary flows are. For a more detailed description of the design file, see the description in the GitHub repository linked in appendix A.

Once a solver is initialized, calling the `solve` function will solve the topology optimization problem and save the output of each iteration to an output folder. The solve function is implemented as

```
def solve(self):
    max_iterations = 1000
    objective_increasing_factor = 2
    max_iterations_without_improvement = 50

    objective_timer = Timer()

    psi = logit(self.to_array(self.rho))
    previous_psi = None

    for penalty in self.parameters.penalties:
        self.problem.set_penalization(penalty)

        objective_timer.restart()
        objectives = [self.problem.calculate_objective(self.rho)]
        times = [objective_timer.get_time_seconds()]

        k = 0
```

```
        exit_condition = ""
        for k in range(max_iterations):
            if k % self.skip_multiple == 0:
                self.save_iteration(self.rho, objectives[-1], k, penalty)

            objective_timer.restart()
            previous_psi = psi.copy()
            try:
                psi = self.step(previous_psi, self.step_size_at_iter(k))
            except ValueError as e:
                exit_condition = str(e)
                break

            self.set_from_array(self.rho, expit(psi))

            objectives.append(self.problem.calculate_objective(self.rho))
            times.append(objective_timer.get_time_seconds())

            if np.isnan(objectives[-1]):
                exit_condition = "Objective is NaN"
                break

            min_index = int(np.argmin(objectives))
            min_objective = objectives[min_index]

            if objectives[-1] > objective_increasing_factor * min_objective:
                exit_condition = "Objective is increasing"
                break

            if k >= min_index + max_iterations_without_improvement:
                exit_condition = "Objective is not decreasing"
                break

            difference = np.sqrt(
                self.integrate((self.to_array(self.rho) - expit(previous_psi)) ** 2)
            )

            if difference < self.tolerance(k):
                exit_condition = "Convergence treshold reached"
                break
        else:
            exit_condition = "Iteration did not converge"

        self.save_iteration(self.rho, objectives[-1], k + 1, penalty)
        self.save_result(objectives, times, penalty, exit_condition)
```

To save space, we have removed printing and comments. We will do this with all the code segments in this chapter. In the above code, `expit` implements $\sigma(x)$ and `logit` implements $\sigma^{-1}(x)$. The functions `to_array`, `set_from_array` and `integrate` are all abstract methods. The object `self.problem` is an instance of a `Problem` class, which is an abstract base class that encapsulates a state equation. The function `step` contains the core logic of the EDM:

```
def step(self, previous_psi: npt.NDArray, step_size: float):
    objective_gradient = self.to_array(self.problem.calculate_objective_gradient())
    half_step = previous_psi - step_size * objective_gradient
    return self.project(half_step, self.volume)

def project(self, half_step: npt.NDArray, volume: float):
```

```
    def error(c: float):
        return self.integrate(expit(half_step + c)) - volume

    def error_derivative(c: float):
        return self.integrate(expit_diff(half_step + c))

    try:
        c, result = optimize.newton(
            error,
            0,
            error_derivative,
            tol=1e-12,
            full_output=True,
        )
        if result.converged:
            return half_step + c
    except RuntimeError:
        pass

    c, result = smart_brentq(error, 2, 2000)
    if not result.converged:
        raise ValueError("Projection failed to converge")

    return half_step + c
```

Here, `expit_diff` implements $\sigma'(x)$, and `smart_brentq` uses the implementation of Brent's algorithm in `scipy.optimize.brentq` together with the boundary values from algorithm 2:

```
def smart_brentq(f: Callable[[float], float], initial_radius: float, max_radius: float):
    r = initial_radius
    while True:
        if r > max_radius:
            raise ValueError(
                "f(-max_radius) and f(max_radius) must have different signs!"
            )

        try:
            return optimize.brentq(f, -r, r, full_output=True)
        except ValueError:
            r *= 2
```

The `Problem` class just contains four abstract methods:

```
class Problem(ABC):
    @abstractmethod
    def set_penalization(self, penalization: float):
        ...

    @abstractmethod
    def calculate_objective_gradient(self) -> Any:
        ...

    @abstractmethod
    def calculate_objective(self, rho: Any) -> float:
        ...

    @abstractmethod
    def forward(self, rho: Any) -> Any:
        ...
```

The function `set_penalization` sets the penalization value, which is p for linear elasticity and q for Stokes flow. The function `forward` solves the state equation, `calculate_objective` calculates ϕ, and `calculate_objective_gradient` calculates $\nabla\phi$. Note that `calculate_objective_gradient` does not take in ρ. This is because calculating the gradient always happens after calculating the objective, and much of the computation to calculate the objective can be reused when calculating the gradient. As with the `Solver` class, these abstract methods are implemented separately in `DEM_src` and `FEM_src`.

5.2 `FEM_src`

To solve PDEs with FEniCS, we made the class `SmartMumpsSolver`. This class uses MUMPS instead of the default FEniCS solver, as that solver has a bug where it can only use up to about 5 GB of RAM. `SmartMumpsSolver` also precalculates A or b if $a(u, v)$ or $l(v)$ are iteration independent, which saves a bit of redundant computation. The class is defined as follows:

```
import dolfin as df
from ufl.form import Form
from ufl.argument import Argument

class SmartMumpsSolver:
    def __init__(
        self,
        a_func: Callable[[Argument, Argument, Any | None], Form],
        l_func: Callable[[Argument, Any | None], Form],
        function_space: df.FunctionSpace,
        boundary_conditions: list[df.DirichletBC] | None = None,
        a_has_no_args: bool = False,
        l_has_no_args: bool = False,
    ):
        self.a_func = a_func
        self.l_func = l_func
        self.function_space = function_space

        if boundary_conditions is None:
            self.boundary_conditions = []
        else:
            self.boundary_conditions = boundary_conditions

        self.A = None
        self.b = None

        if a_has_no_args or l_has_no_args:
            trial = df.TrialFunction(self.function_space)
            test = df.TestFunction(self.function_space)

            if a_has_no_args:
                a = self.a_func(trial, test, None)
                self.A = df.assemble(a)

            if l_has_no_args:
                l = self.l_func(test, None)
                self.b = df.assemble(l)

    def solve(self, *, a_arg: Any | None = None, l_arg: Any | None = None):
        A = self.A
```

```
        b = self.b

        if self.A is None or self.b is None:
            trial = df.TrialFunction(self.function_space)
            test = df.TestFunction(self.function_space)

            if self.A is None:
                a = self.a_func(trial, test, a_arg)
                A = df.assemble(a)

            if self.b is None:
                l = self.l_func(test, l_arg)
                b = df.assemble(l)

        _ = [bc.apply(A, b) for bc in self.boundary_conditions]

        solution = df.Function(self.function_space)
        solution_vector = solution.vector()

        solver = df.LUSolver("mumps")
        solver.solve(A, solution_vector, b)

        return solution
```

The Helmholtz filter is then simply implemented as

```
class HelmholtzFilter:
    def __init__(self, epsilon: float, function_space: df.FunctionSpace):
        def a_func(trial, test, _):
            return (
                epsilon**2 * df.inner(df.grad(trial), df.grad(test))
                + df.inner(trial, test)
            ) * df.dx

        def l_func(test, input_function):
            return df.inner(input_function, test) * df.dx

        self.solver = SmartMumpsSolver(
            a_func, l_func, function_space, a_has_no_args=True
        )

    def apply(self, input_function: df.Function):
        return self.solver.solve(l_arg=input_function)
```

The children of the `Problem` class for linear elasticity and Stokes flow are similar, so we will only show the one for Stokes flow, named `FluidProblem`. As with the `HelmholtzFilter`, we create a `SmartMumpsSolver`:

```
def create_solver(self):
    def a_func(trial, test, rho):
        (u, p) = df.split(trial)
        (v, q) = df.split(test)

        return (
            self.penalizer(rho) * df.inner(u, v)
            + df.inner(df.grad(u), df.grad(v))
            + df.inner(df.grad(p), v)
            + df.inner(df.div(u), q)
        ) * df.dx

```

```
    def l_func(test, _):
        return df.inner(test, df.Constant([0, 0, 0])) * df.dx

    return SmartMumpsSolver(
        a_func,
        l_func,
        self.boundary_conditions,
        self.solution_space,
        l_has_no_args=True,
    )
```

We use a body force of 0 in the Stokes examples, so $l(v) = 0$. Unfortunately, just returning 0 does not work, which is why `l_func` is defined so weirdly.

The functions responsible for calculating the objective and gradient are defined as follows

```
def calculate_objective_gradient(self):
    return df.project(
        0.5 * self.penalizer.derivative(self.rho) * self.u**2,
        self.rho.function_space(),
    )

def calculate_objective(self, rho):
    self.rho = rho
    (self.u, _) = df.split(self.forward(rho))

    t1 = self.penalizer(rho) * self.u**2
    t2 = self.viscosity * df.grad(self.u) ** 2
    objective = df.assemble(0.5 * (t1 + t2) * df.dx)

    return float(objective)

def forward(self, rho):
    return self.solver.solve(a_arg=rho)
```

Here, the reason for why `calculate_objective_gradient` does not take in rho is clear; the fluid velocities are saved after calculating the objective, and then reused when calculating the gradient.

The boundary conditions and function space are created by the functions

```
def create_boundary_conditions(self):
    flow_sides = [flow.side for flow in self.parameters.flows]
    self.marker.add(SidesDomain(self.domain_size, flow_sides), "flow")

    no_slip_sides = list(set(Side.get_all()).difference(flow_sides))
    self.marker.add(SidesDomain(self.domain_size, no_slip_sides), "no_slip")

    self.boundary_flows = BoundaryFlows(
        self.domain_size, self.parameters.flows, degree=2
    )

    boundary_conditions = [
        df.DirichletBC(
            self.solution_space.sub(0),
            self.boundary_flows,
            *self.marker.get("flow"),
        ),
        df.DirichletBC(
            self.solution_space.sub(0),
```

```
            df.Constant((0.0, 0.0)),
            *self.marker.get("no_slip"),
        ),
    ]

    return boundary_conditions

def create_solution_space(self):
    velocity_space = df.VectorElement("CG", self.mesh.ufl_cell(), 2)
    pressure_space = df.FiniteElement("CG", self.mesh.ufl_cell(), 1)
    return df.FunctionSpace(self.mesh, velocity_space * pressure_space)
```

Where `BoundaryFlows` implements the boundary flows as an expression:

```
class BoundaryFlows(df.UserExpression):
    def __init__(self, domain_size: tuple[float, float], flows: list[Flow], **kwargs):
        super().__init__(**kwargs)
        self.domain_size = domain_size
        self.flows = flows

    def get_flow(self, position: float, center: float, length: float, rate: float):
        t = position - center
        if -length / 2 < t < length / 2:
            return rate * (1 - (2 * t / length) ** 2)
        return 0

    def eval(self, values, pos):
        values[0], values[1] = 0.0, 0.0

        for side, center, length, rate in [f.to_tuple() for f in self.flows]:
            if side == Side.LEFT:
                if pos[0] == 0.0:
                    values[0] += self.get_flow(pos[1], center, length, rate)
            elif side == Side.RIGHT:
                if pos[0] == self.domain_size[0]:
                    values[0] -= self.get_flow(pos[1], center, length, rate)
            elif side == Side.TOP:
                if pos[1] == self.domain_size[1]:
                    values[1] -= self.get_flow(pos[0], center, length, rate)
            elif side == Side.BOTTOM:
                if pos[1] == 0:
                    values[1] += self.get_flow(pos[0], center, length, rate)
            else:
                raise ValueError(f"Malformed side: {side}")

    def value_shape(self):
        return (2,)
```

5.3 DEM_src

Our implementation of the DEM is based on the source code provided by [24]. We started by copying all the code into our codebase, and then removed all unnecessary code and cleaned it up to avoid sketchy global values. After that, we modified the program such that it used EMD instead of MMS, and made the code more generic so it could handle all the examples we needed. This means that the code in `DEM_src` is almost unrecognizable compared to what we started with, but you can see some of the same structures. While working with the DEM source code, we found and fixed two bugs, described in appendix

C. One of these bugs is in the calculation of the traction integral, and is the reason we developed a more robust integration algorithm for that integral.

Solving PDEs with a VPINN is done by the `DeepEnergyMethod` class, which has a `train_model` method which takes in ρ and returns the objective ϕ and the gradient $\nabla\phi$:

```
def train_model(self, rho: npt.NDArray[np.float64], mesh: Mesh):
    x = torch.from_numpy(flatten([mesh.x_grid, mesh.y_grid])).float()
    x = x.to(self.device)
    x.requires_grad_(True)

    density = torch.from_numpy(rho).float()
    density = torch.reshape(density, mesh.intervals).to(self.device)

    optimizer_LBFGS = torch.optim.LBFGS(
        self.model.parameters(),
        lr=self.nn_parameters.learning_rate,
        max_iter=20,
        line_search_fn="strong_wolfe",
    )

    def closure_generator(t: int):
        def closure():
            u_pred = self.get_u(x)
            u_pred.double()

            loss = self.objective_calculator.calculate_energy(
                u_pred, mesh.shape, density
            )
            optimizer_LBFGS.zero_grad()
            loss.backward()

            self.loss_array.append(loss.data.cpu())

            return float(loss)

        return closure

    for t in range(self.nn_parameters.iteration_count):
        optimizer_LBFGS.step(closure_generator(t))

        if self.convergence_check(
            self.loss_array,
            self.nn_parameters.convergence_tolerance,
        ):
            break

    return self.objective_calculator.calculate_objective_and_gradient(
        self.get_u(x), mesh.shape, density
    )
```

There are two important parts of this function. The first is `self.objective_calculator`, which is an instance of an `ObjectiveCalculator`, a class that contains logic for calculating the objective and objective gradient. It is quite complicated, so we will first explain the other important part, `self.get_u`, which basically just returns `self.dirichlet_enforcer(self.model(x))`. `self.dirichlet_enforcer` implements additive decomposition, and `self.model` is an instance of a `MultiLayerNet`, which is implemented as

```python
class MultiLayerNet(torch.nn.Module):
    def __init__(self, input_size, output_size, parameters: NNParameters):
        super().__init__()

        neuron_count = parameters.neuron_count
        weight_deviation = parameters.weight_deviation
        fourier_deviation = parameters.fourier_deviation

        self.layer_count = parameters.layer_count
        self.activation_function = getattr(torch, parameters.activation_function)
        self.encoding = rff.layers.GaussianEncoding(
            sigma=fourier_deviation, input_size=input_size, encoded_size=neuron_count//2
        )

        self.linears = torch.nn.ModuleList()
        for i in range(self.layer_count):
            linear_inputs = input_size if i == 0 else neuron_count
            linear_outputs = output_size if i == self.layer_count - 1 else neuron_count
            self.linears.append(torch.nn.Linear(linear_inputs, linear_outputs))

            torch.nn.init.constant_(self.linears[i].bias, 0.0)
            torch.nn.init.normal_(self.linears[i].weight, mean=0, std=weight_deviation)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        y = self.encoding(x)
        for i in range(1, self.layer_count - 1):
            y = self.activation_function(self.linears[i](y))
        y = self.linears[-1](y)

        return y
```

`ObjectiveCalculator` is an abstract base class, as the objective and gradient must be implemented separately for the linear elasticity and Stokes flow case. Its most important functions are

```python
def get_grad_u(
    self, gauss_point: int, point_lists: list[list[torch.Tensor]]
) -> tuple[torch.Tensor, torch.Tensor]:
    dxdy_list = np.array(
        [
            np.matmul(self.Jinv, dN_dsy[:, gauss_point].reshape((2, 1)))
            for dN_dsy in self.shape_derivatives
        ]
    ).reshape((len(self.shape_derivatives), 2))

    shape = point_lists[0][0].shape
    u = torch.zeros((len(point_lists), *shape))
    gradient = torch.zeros((len(point_lists), 2, *shape))

    for i in range(len(self.shape_derivatives)):
        dx, dy = dxdy_list[i, :]

        for j, Ux_list in enumerate(point_lists):
            Ux = Ux_list[i]

            u[j, :, :] += Ux

            gradient[j, 0, :, :] += Ux * dx
            gradient[j, 1, :, :] += Ux * dy

```

```
    u /= len(self.shape_derivatives)

    return u, gradient

def value_at_gauss_point(
    self,
    function: Callable[[torch.Tensor, torch.Tensor], list[torch.Tensor]],
    gauss_point: int,
    point_lists: list[list[torch.Tensor]],
):
    u, grad = self.get_grad_u(gauss_point, point_lists)
    return function(u, grad)

def evaluate(
    self,
    u: torch.Tensor,
    shape: tuple[int, int],
    function: Callable[[torch.Tensor, torch.Tensor], list[torch.Tensor]],
):
    _, dim = u.shape
    U = torch.transpose(u.reshape(shape[1], shape[0], dim), 0, 1)

    point_lists = self.get_gauss_points(U)

    values = self.value_at_gauss_point(function, 0, point_lists)
    for i in range(1, len(self.shape_derivatives)):
        for j, v in enumerate(self.value_at_gauss_point(function, i, point_lists)):
            values[j] += v

    return values
```

The `evaluate` function can then be used to calculate the integral of functions that depend on $\boldsymbol{u}$ and $\nabla\boldsymbol{u}$.

We will again only show the Stokes flow problem. It starts with creating an instance of a `DeepEnergyMethod`:

```
def create_dem_parameters(self):
    dirichlet_enforcer = FluidEnforcer(
        self.parameters, self.mesh, self.device
    )
    objective_calculator = FluidEnergy(self.mesh, self.parameters.viscosity, 500)

    return dirichlet_enforcer, objective_calculator

def create_dem(
    self,
    dirichlet_enforcer: FluidEnforcer,
    objective_calculator: FluidEnergy,
):
    nn_parameters = NNParameters(
        layer_count=5,
        neuron_count=66,
        learning_rate=1.3330789490587558,
        iteration_count=100,
        weight_deviation=0.36941508201470885,
        fourier_deviation=0.7239620095758805,
        activation_function="sigmoid",
        convergence_tolerance=5e-5,
    )

```

```
    dimension = 2
    return DeepEnergyMethod(
        self.device,
        self.verbose,
        dimension,
        nn_parameters,
        dirichlet_enforcer,
        objective_calculator,
    )
```

Where `FluidEnforcer` is a child of `DirichletEnforcer`, a class made to enforce the Dirichlet boundary conditions, and `FluidEnergy` is a child of `ObjectiveCalculator`, defined as

```
class FluidEnergy(ObjectiveCalculator):
    def __init__(self, mesh: Mesh, viscocity: float, tau: float):
        super().__init__(mesh, FluidPenalizer())
        self.viscocity = viscocity
        self.tau = tau

    def calculate_all_norms(self, u: torch.Tensor, grad_u: torch.Tensor):
        u_norm, grad_u_norm = self.calculate_some_norms(u, grad_u)
        div_u = grad_u[0][0] + grad_u[1][1]

        return [u_norm, grad_u_norm, div_u**2]

    def calculate_some_norms(self, u: torch.Tensor, grad_u: torch.Tensor):
        return [torch.sum(u**2, 0), torch.sum(grad_u**2, [0, 1])]

    def calculate_energy(
        self, u: torch.Tensor, shape: tuple[int, int], density: torch.Tensor
    ):
        u_norm, grad_norm, div_norm = self.evaluate(u, shape, self.calculate_all_norms)

        potential = (
            0.5 * (self.penalizer(density) * u_norm + self.viscocity * grad_norm)
            + self.tau * div_norm
        )

        return torch.sum(potential * self.detJ)

    def calculate_objective_and_gradient(
        self, u: torch.Tensor, shape: tuple[int, int], density: torch.Tensor
    ):
        u_norm, grad_norm = self.evaluate(u, shape, self.calculate_some_norms)
        potential = self.penalizer(density) * u_norm + self.viscocity * grad_norm

        objective = 0.5 * torch.sum(potential * self.detJ)
        gradient = 0.5 * self.penalizer.derivative(density) * u_norm

        return objective, gradient
```

The functions responsible for calculating the objective and gradient are defined as follows

```
def calculate_objective_gradient(self):
    return self.objective_gradient

def calculate_objective(self, rho: npt.NDArray[np.float64]):
```

```
    objective, objective_gradient = self.dem.train_model(rho, self.mesh)

    objective = objective.cpu().detach().numpy()
    self.objective_gradient = objective_gradient.cpu().detach().numpy()

    return float(objective)
```

As the `train_model` function returns both the objective and objective gradient, `calculate_objective_gradient` does no computation, it just returns a stored value.

5.4 Testing

While developing our program, we made many tests to ensure that we did not accidentally break one part while working on another. In this section, we will highlight two of those tests; one testing the Helmholtz filter and one testing the objective calculator.

For the Helmholtz filter, we first found an analytical solution to the PDE using the method of manufactured solutions [35]. To do this we chose the domain $\Omega = [0,1]^2$, and found some function that satisfies the boundary conditions, namely $\tilde{\rho}_{MMS}(x,y) = \cos(2\pi x)\cos(2\pi y)$. We then found the input ρ such that $\tilde{\rho}_{MMS}$ satisfied the PDE. Doing this gives $\rho_{MMS} = (8\epsilon^2\pi^2 + 1)\cos(2\pi x)\cos(2\pi y)$. This tells us that filtering ρ_{MMS} should give us $\tilde{\rho}_{MMS}$, so $\tilde{\rho}_{MMS}$ is an analytical solution to the Helmholtz equation. With this, we could make a test:

```
def test_HelmholtzFilter():
    random_epsilon = np.e / np.pi
    rho_expression = df.Expression(
        "(8*eps*eps*pi*pi + 1)*cos(2*pi*x[0])*cos(2*pi*x[1])",
        eps=random_epsilon,
        degree=2,
    )
    filtered_rho_expression = df.Expression("cos(2*pi*x[0])*cos(2*pi*x[1])", degree=2)

    def error_func(N):
        mesh, rho = initialize(N)

        design_filter = HelmholtzFilter(random_epsilon, rho.function_space())
        rho.interpolate(rho_expression)
        filtered_rho = design_filter.apply(rho)

        return df.errornorm(
            filtered_rho, filtered_rho_expression, "L2", degree_rise=2, mesh=mesh
        )

    Ns = list(range(10, 90 + 1, 10))
    assert get_convergance(Ns, error_func) <= -2
```

This test ensured that the error from the filter implementation decreases proportional to $1/N^2$. Figure 5.1 shows a log-log plot of the filter error as a function of N, together with the convergence rate.

For the objective calculator, we implemented a simple child class to calculate $\int_\Omega ||\boldsymbol{u}||^2 + ||\nabla \boldsymbol{u}||^2 \, \mathrm{d}\boldsymbol{x}$, and then tested it on $\Omega = [0,5] \times [0,2]$ and $\boldsymbol{u} = (\cos(2y)\sin(x), \cos(y)\sin(2x))$, which has an analytical value that is complicated, so we will not repeat it here. As in the previous test, we ensured that the error decreases

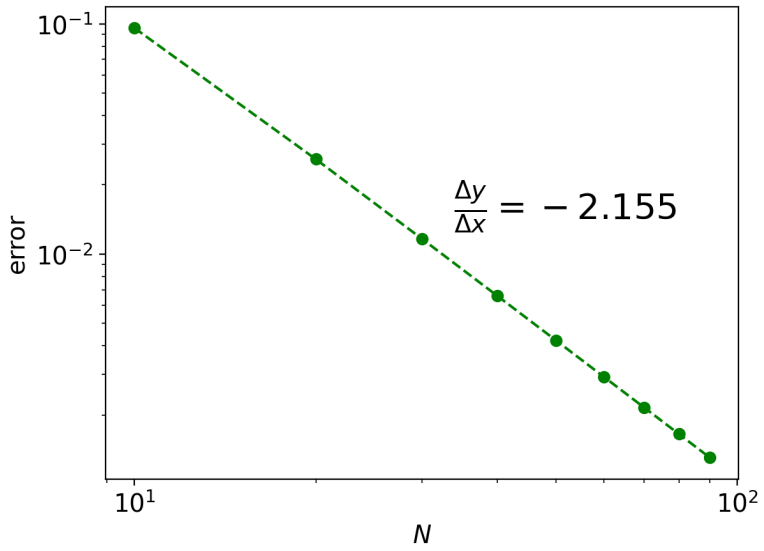


Figure 5.1: A figure depicting a log-log plot of the filter error as a function of N from the `HelmholtzFilter` test. The convergence rate of the filter error is also depicted.

proportional to $1/N^2$. Figure 5.2 shows a log-log plot of the error as a function of N, together with the convergence rate.

```
class DummyObjective(ObjectiveCalculator):
    def value(self, u, grad_u):
        return [torch.sum(u**2, 0) + torch.sum(grad_u**2, [0, 1])]

    def calculate_energy(
        self, u: torch.Tensor, shape: tuple[int, int], density: torch.Tensor
    ):
        (value,) = self.evaluate(u, shape, self.value)

        return torch.sum(value * self.detJ)

    def calculate_objective_and_gradient(
        self, u: torch.Tensor, shape: tuple[int, int], density: torch.Tensor
    ):
        pass


def trig(x_grid, y_grid):
    ux = np.cos(2 * y_grid) * np.sin(x_grid)
    uy = np.cos(y_grid) * np.sin(2 * x_grid)

    return [ux, uy]


def trig_analytic(mesh: Mesh):
    w, h = mesh.length, mesh.height

    t1 = 24 * w * h - 4 * h * np.sin(2 * w) + h * np.sin(4 * w) + 4 * w * np.sin(2 * h)
    t2 = (np.sin(2 * w) - w) * np.sin(4 * h) + np.sin(4 * w) * np.sin(2 * h)

    return (t1 + t2) / 8


```

```python
def test_evaluate():
    def trig_errfunc(N: int):
        mesh = Mesh(2 * N, 3 * N, 5, 2)
        objective = DummyObjective(mesh, ElasticPenalizer())
        u = torch.from_numpy(flatten(trig(mesh.x_grid, mesh.y_grid))).float()

        numeric = float(
            objective.calculate_energy(u, mesh.shape, torch.ones_like(u))
        )
        analytic = trig_analytic(mesh)

        return abs(analytic - numeric)

    Ns = list(range(5, 100, 5))
    assert get_convergance(Ns, trig_errfunc) < -2
```

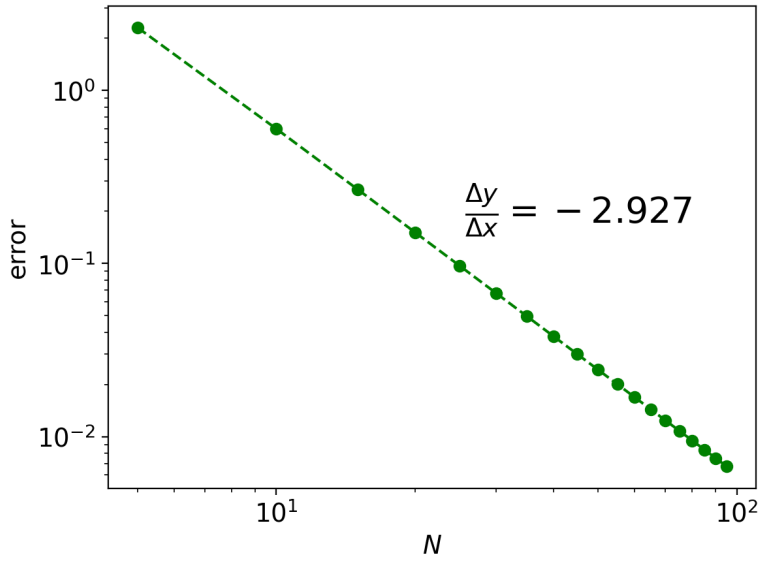


Figure 5.2: A figure depicting a log-log plot of the error as a function of N from the `ObjectiveCalculator` test. The convergence rate of the error is also depicted.

Chapter 6

Results and Comparisons

6.1 Hyperparameters

One hyperparameter both the FEM and the DEM share is the EMD step size. While a fixed or decreasing step size is typically used, the paper describing the EMD, [25], experimented with increasing step sizes. The cantilever example comes from [25], where a step size of $\alpha_k = 25(k+1)$ worked well. For the other examples, we used a step size of the form $\alpha_k = \alpha_0(k+1)$, and found α_0 by trying values until one worked. If α_0 is too large, the iteration diverges. We therefore tried finding the largest step size where the iteration still converges. This method of finding a step size is obviously not ideal, a better method might have been to use some sort of line search algorithm, but the best algorithm for finding the EMD step size is still an active area of research. We therefore decided to stick with the method used in the EMD paper, as we know that worked for the cantilever example. The step sizes we found are summarized in table 6.1. The only case where we used a different method was the twin pipe example, where we could not find a good step size. We therefore limited how big the step size could be by using $\alpha_k = \alpha_0 \min\{(k+1), 10\}$.

State equation	Example	FEM α_0	DEM α_0
Linear elasticity	Cantilever	25.0	37500.0
	Short cantilever	0.03	1.5
	Bridge	0.001	0.2
Stokes flow	Diffuser	0.0011	0.0002
	Pipe bend	0.0015	0.0001
	Twin pipe	0.0015	0.0004

Table 6.1: A table showing the EMD step sizes we found for each example, both for the FEM and the DEM. Step sizes were found by trying values until the iteration converged.

The DEM has many hyperparameters. For the linear elasticity examples, we just used the same hyperparameters they did in [24]. For Stokes flow, we first guessed some divergence penalty τ, and then ran `hyperopt` to find the optimal hyperparameters. The hyperparameters for both linear elasticity and Stokes flow are shown in table 6.2.

After finding the hyperparameters, we tuned τ by comparing the fluid velocities produced by the DEM on the diffuser example with $N = 40$ and $\rho = 0.5$ with those from the FEM. Figure 6.1 shows how τ affects the fluid velocities. We found that a too small τ value caused the magnitude of the fluid velocities to decrease as divergence was not

	NL	NN	σ	Max iters	η	σ_W	σ_F
Linear elasticity	5	68	RReLU	100	1.736	0.06226	0.1193
Stokes flow	5	66	Sigmoid	100	1.333	0.3694	0.7240

Table 6.2: Table of the hyperparameters used with the DEM for both linear elasticity and Stokes flow. NL is the number of layers, NN is the number of neurons in each layer, σ is the activation function, η is the learning rate, σ_W is the standard deviation of the random initialized weights, and σ_F is the standard deviation for the random Fourier features.

penalized enough, and a too large τ caused the fluid velocities to not spread out enough, causing most of the fluid to go straight forwards. We found that $\tau = 500$ has the best balance between those two trends, but it is not perfect, having an error in magnitude of about 20 % compared to the FEM solution.

6.2 Results

We ran all the examples for both methods with four discretization parameters; $N = 40$, $N = 80$, $N = 160$, and $N = 320$, and we will show the final designs for two of them, namely $N = 40$ and $N = 160$. We will showcase the smallest objective reached and the corresponding design, which is often, but not always, the same as the final objective. For the EMD convergence criterion, we used $\left|\left|\rho^{k+1} - \rho^k\right|\right|_{L_2} \leq \min\{25(k+1) \cdot 10^{-5}, 10^{-2}\}$. Note that we always used $25(k+1)$ instead of α_k. We did this as we would otherwise have to use a different `ntol` for each example.

The results for the three linear elasticity examples solved using the FEM are shown in figure 6.2 for the case with a discretization parameter of $N = 40$, and figure 6.3 for $N = 160$. Table 6.3 summarizes important values from the optimization, namely the best objective, the iteration where the best objective was reached, the total number of iterations, the time taken, and whether the iteration converged or not. For each of the tree examples, the values for all four discretization parameters are shown.

Example	N	Objective	Best iter	Total iters	Time	Converged?
Cantilever	40	0.003711	37	37	2 m 55 s	Yes
	80	0.003868	39	39	13 m 18 s	Yes
	160	0.003957	41	41	31 m 9 s	Yes
	320	0.003992	42	42	2 h 29 m	Yes
Short cantilever	40	2.910	149	149	3 m 56 s	Yes
	80	17.59	194	194	24 m 29 s	Yes
	160	21.14	232	232	48 m 56 s	Yes
	320	23.05	328	328	4 h 9 m	Yes
Bridge	40	325.7	188	188	5 m 45 s	Yes
	80	310.9	236	236	1 h 19 m	Yes
	160	304.5	311	311	3 h 32 m	Yes
	320	301.1	526	526	25 h 1 m	Yes

Table 6.3: A table showing minimum objective value, iteration where the minimum was reached, total iterations, time taken, and if the iteration converged or not. These values are shown for the three linear elasticity examples solved using the FEM. For each example, the results for four discretization parameters are shown; $N = 40$, $N = 80$, $N = 160$ and $N = 320$.

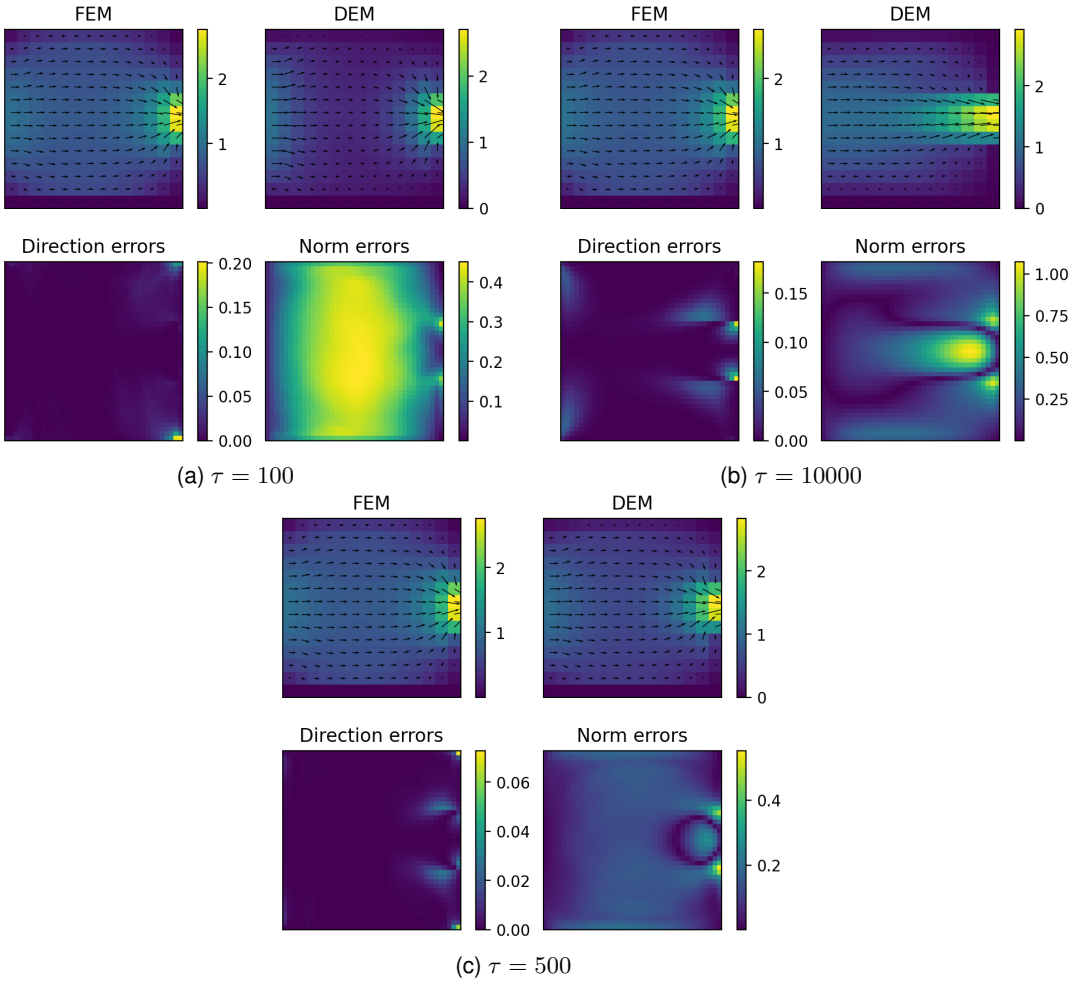


Figure 6.1: Figure showing the fluid velocities for the diffuser with $N = 40$ and $\rho = 0.5$ for different penalization parameters τ. Figure 6.1a shows $\tau = 100$, figure 6.1b shows $\tau = 10000$, and figure 6.1c shows $\tau = 500$. Fluid velocities are compared to those from the FEM, and both the error in the direction of the velocities, calculated with $\frac{1}{2} - \frac{\boldsymbol{u}_{FEM} \cdot \boldsymbol{u}_{DEM}}{2||\boldsymbol{u}_{FEM}|| \, ||\boldsymbol{u}_{DEM}||}$, and error in the magnitude of the velocities, calculated with $\big|\,||\boldsymbol{u}_{FEM}|| - ||\boldsymbol{u}_{DEM}||\,\big|$, are shown.

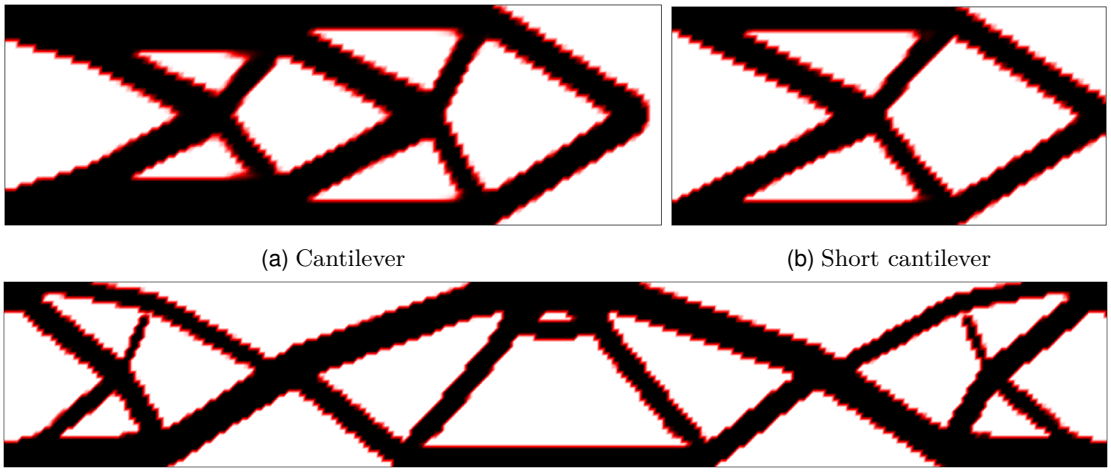
The results for the three Stokes flow examples solved using the FEM are shown in figure 6.4 for the case with $N = 40$, and figure 6.5 for $N = 160$. Table 6.4 again summarizes the most important values from the optimization for all four discretization parameters.

The results for the three linear elasticity examples solved using the DEM are shown in figure 6.6 for the case with $N = 40$, and figure 6.7 for $N = 160$. Table 6.5 summarizes important values from the optimization.

The results for the three Stokes flow examples solved using the DEM are shown in figure 6.8 for the case with $N = 40$, and figure 6.9 for $N = 160$. Table 6.6 summarizes important values from the optimization.

6.3 Comparison Indices

Before comparing the two methods, we must first decide what we want to compare. There are many aspects we could compare, with the most obvious one probably being how the



(a) Cantilever

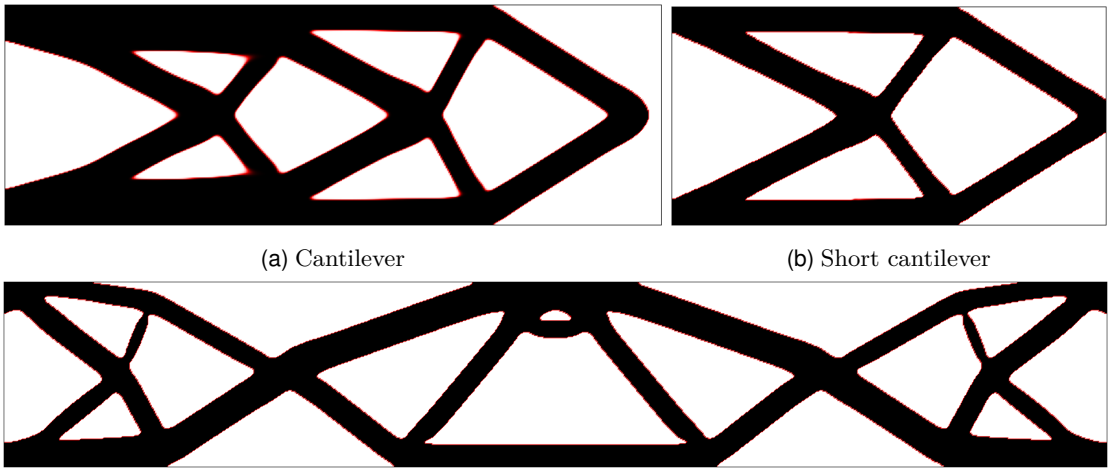
(b) Short cantilever

(c) Bridge

Figure 6.2: Figures showcasing the optimized topologies, using the FEM, for three linear elasticity examples; the cantilever shown in 6.2a, the short cantilever shown in 6.2b, and the bridge shown in 6.2c. In all three examples, the discretization parameter $N = 40$ is used. A density of 1, indicating presence of material, is shown in black, and a density of 0, indicating absence of material, is shown in white. Intermediate values are highlighted by coloring densities around 0.5 in red.

final design looks. The goal of topology optimization is to find the optimal design, so it seems reasonable to assert that the design is the most important result. Unfortunately, comparing figures is very imprecise. Even if two designs look the same, there might be differences that are not easily observable, but nevertheless have a meaningful impact on the objective. Despite this drawback, comparing the designs has some benefits. One of which is that it works as a sanity check. If something has gone wrong in the optimization, that should be obvious when looking at the finished design. Another benefit is that all the papers we have gotten our examples from include the optimal design they got, so we have a reference to compare our results to. The results from the papers which used linear elasticity, [25] and [24], are shown in figure 6.10. The results from the paper which used Stokes flow, [8], are shown in figure 6.11.

Another obvious comparison index is the objective value of the final design. This is indeed a good index, but with a few caveats. An optimization problem might have multiple solutions, meaning that the objective function has multiple local minima. If the optimization converges to a suboptimal local minimum, that is not necessarily the fault of the PDE solver. The other problem is that each method calculates its own objective, so they are not necessarily comparable. This is also true for different mesh sizes within a method. To make the objectives comparable, we made a simple program that can read in a finished design, interpolate it with the FEM with $N = 320$, and then recalculate the objective with the interpolated design. The results from this program are shown in table 6.7 for the FEM objectives and table 6.8 for the DEM objectives. With the objectives from multiple N-values, we can tell if the design seems to be converging or not. If the difference between consecutive objectives decreases as the mesh becomes finer, the design is probably converging. By calculating the rate at which the difference decreases, we can calculate an approximate convergence factor. We do not have any reference objectives for the linear elasticity examples, but we do have them for the Stokes flow examples, which we have complied in table 6.9. This table contains the objectives form the paper



(a) Cantilever

(b) Short cantilever

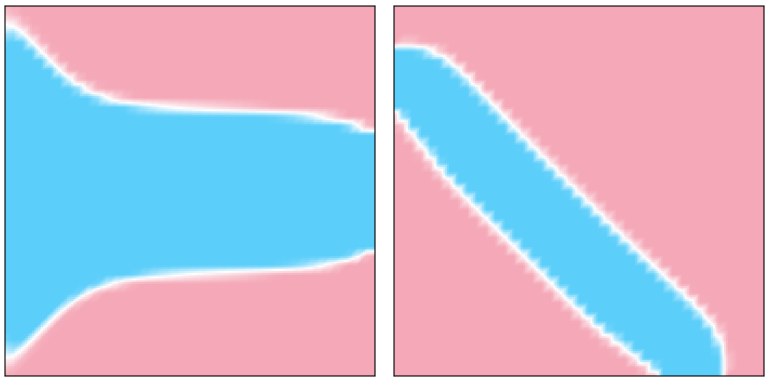
(c) Bridge

Figure 6.3: Figures showcasing the optimized topologies, using the FEM, for three linear elasticity examples; the cantilever shown in 6.3a, the short cantilever shown in 6.3b, and the bridge shown in 6.3c. In all three examples, the discretization parameter $N = 160$ is used. A density of 1, indicating presence of material, is shown in black, and a density of 0, indicating absence of material, is shown in white. Intermediate values are highlighted by coloring densities around 0.5 in red.

we got the examples from, [8], as well as the objectives for the twin pipe example from a paper that computed the objective for multiple minima, [32]. From that paper we see that the twin pipe has two solutions, one where the pipes are joined in the center, which is the global minimum, and one where the pipes do not join, which is a non-optimal local minimum.

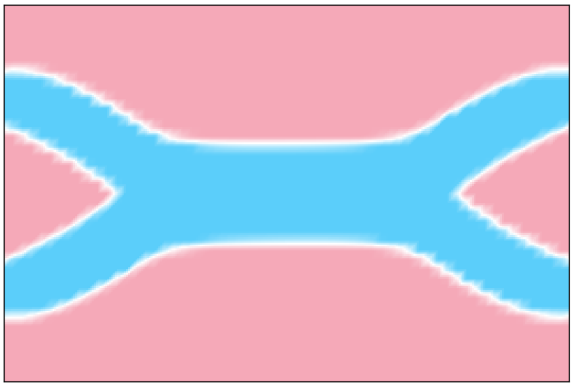
The speed of a method might seem like a good comparison index, and it is important for users of the program we developed, but it does not tell us much about the quality of the underlying methods. This is because the implementation of a method plays a large role in how fast it is, and we cannot guarantee that we have implemented the methods equally well. However, as the speed does influence the usability of our program, we will still compare how fast the methods are. From the time taken for several N-values, we can calculate a rough scaling factor for each method. This scaling factor is not dependent on the exact implementation, but it will still change based on which algorithms we have used.

Another index that is related to speed but not implementation dependent is the number of iterations before convergence. This is also a flawed index, as the number of iterations depends on the step size used in the descent, and as we found step sizes manually, we cannot guarantee that they are optimal. How the number of iterations changes as the mesh becomes finer is a useful index however, as it can be used to test mesh independence, which is a property we want the methods to have. For a method to be mesh independent, the number of iterations before convergence must stay constant as the mesh becomes finer.



(a) Diffuser

(b) Pipe bend



(c) Twin pipe

Figure 6.4: Figures showcasing the optimized topologies, using the FEM, for three Stokes flow examples; the diffuser shown in 6.4a, the pipe bend shown in 6.4b, and the twin pipe shown in 6.4c. In all three examples, the discretization parameter $N = 40$ is used. A density of 1, indicating presence of fluid, is shown in blue, and a density of 0, indicating absence of fluid, is shown in red. Intermediate values are highlighted by coloring densities around 0.5 in white.

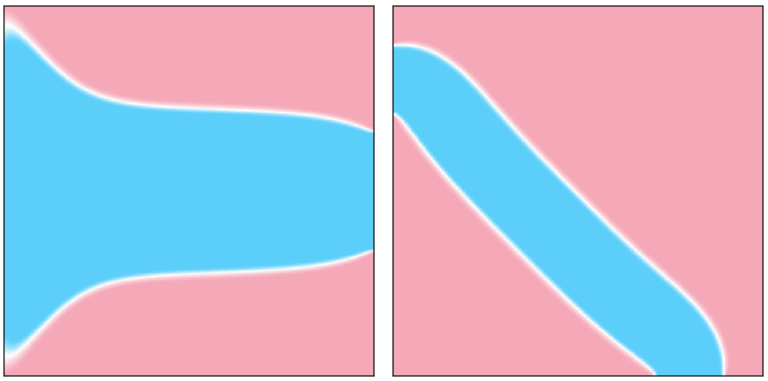
6.4 Comparison

6.4.1 Convergence

Before analyzing our results, we must first discuss the topic of convergence. We stopped the EMD iteration early if one of the following conditions were encountered:

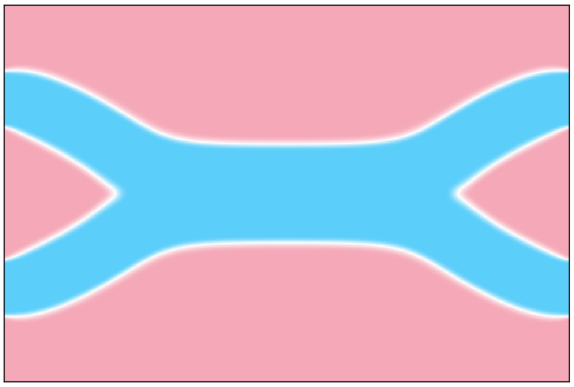
- If the current objective is twice as large as the smallest objective reached.
- If the smallest objective reached has not changed in the last 50 iterations.
- If we have done more than 1000 iterations.

All of these are counted as the iteration diverging. The third condition is only there to prevent the program from potentially running forever, we used a maximum large enough for the condition to never be encountered. The only time an iteration using the FEM did not converge was the twin pipe example with $N = 160$. In this case, the objective increased to a value greater than 10^{12} within one iteration, triggering condition one. We do not know why this happened, but we also observed similar behavior with the diffuser for certain step sizes. This does indicate that the FEM has some sort of



(a) Diffuser

(b) Pipe bend



(c) Twin pipe

Figure 6.5: Figures showcasing the optimized topologies, using the FEM, for three Stokes flow examples; the diffuser shown in 6.5a, the pipe bend shown in 6.5b, and the twin pipe shown in 6.5c. In all three examples, the discretization parameter $N = 160$ is used. A density of 1, indicating presence of fluid, is shown in blue, and a density of 0, indicating absence of fluid, is shown in red. Intermediate values are highlighted by coloring densities around 0.5 in white.

instability. For the DEM, condition one was encountered with the bridge example with $N = 80$. In this case the objective did not explode in a single iteration, but instead increased over multiple iterations. The cantilever example with $N = 40$ did not improve the objective in 50 iterations, and therefore encountered condition two. The diffuser struggled to converge, with the objective increasing for both $N = 160$ and $N = 320$. Aside from that, all other iterations converged. By comparing the iteration with the lowest objective and the total number of iterations of the DEM results in table 6.5 for linear elasticity and 6.6 for Stokes flow, we can notice a curious tendency. Even when the iteration converged, the final objective is often not the smallest. Most of the time, the difference is small, being just a few iterations, but there are some cases where the difference is more than ten iterations. We do not know why the DEM sometimes converges to a worse result.

6.4.2 Figures

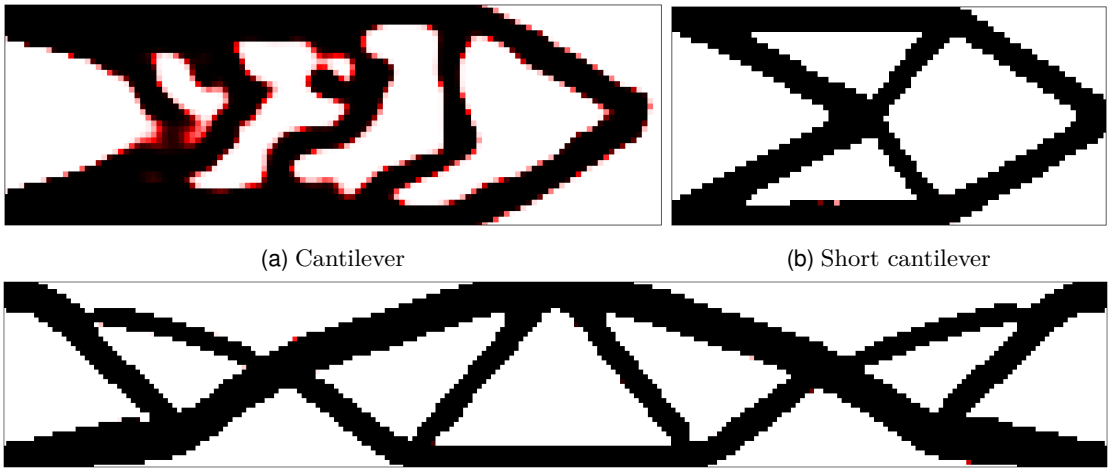
By comparing the results for linear elasticity with the FEM, figure 6.2 for $N = 40$ and figure 6.3 for $N = 160$, we can see that refining the mesh does not change the overall shape of the designs, which is good. As expected, a finer mesh results in smoother designs. It also results in smaller boundary regions where $\rho \notin \{0, 1\}$. This makes sense,

Example	N	Objective	Best iter	Total iters	Time	Converged?
Diffuser	40	31.03	13	13	2 s 917 ms	Yes
	80	30.54	12	12	10 s 560 ms	Yes
	160	30.46	12	12	46 s 736 ms	Yes
	320	30.45	13	13	5 m 13 s	Yes
Pipe bend	40	10.27	18	18	4 s 677 ms	Yes
	80	9.836	20	20	23 s 136 ms	Yes
	160	9.774	20	20	1 m 52 s	Yes
	320	9.767	20	20	8 m 40 s	Yes
twin pipe $q = 0.01$	40	15.82	47	47	17 s 465 ms	Yes
	80	15.67	50	50	1 m 26 s	Yes
	160	15.59	53	53	7 m 45 s	Yes
	320	15.70	49	49	38 m 43 s	Yes
twin pipe $q = 0.1$	40	25.80	23	23	8 s 241 ms	Yes
	80	24.01	422	422	12 m 35 s	Yes
	160	25.02	9	10	1 m 25 s	No
	320	23.93	389	389	5 h 9 m	Yes

Table 6.4: A table showing minimum objective value, iteration where the minimum was reached, total iterations, time taken, and if the iteration converged or not. These values are shown for the three Stokes flow examples solved using the FEM. For each example, the results for four discretization parameters are shown; $N = 40$, $N = 80$, $N = 160$ and $N = 320$.

we used linear elements, so changing the density from one to zero takes the full width of one element. A finer mesh means smaller elements, and thus a smaller diffuse boundary. The design for the bridge example has some problematic elements; for $N = 40$ there seems to be disconnected segments, and for $N = 160$ there are thin segments which could snap if built. This indicates that the filer radius is too small. As the results for linear elasticity with the FEM have the same shape, we will only compare the $N = 160$ case to the reference designs in figure 6.10. Here we can see that the cantilever we got seems to be identical to the reference cantilever. The bridge is also somewhat similar, but the short cantilever is missing the diagonal line. We cannot tell which design is superior from the figure alone.

Things are different with the DEM results. By comparing the results for linear elasticity, figure 6.6 for $N = 40$ and figure 6.7 for $N = 160$, we see that the rough mesh and fine mesh does not give the same shape. This is a problem as it means refining the mesh will not give a more refined version of the rough design, so we cannot tell what the design will look like in the limit as the mesh becomes infinitely fine. As mentioned in section 4.3.3, increasing the mesh size just increases the number of points where the DEM model is sampled, and should therefore just result in a smoother design. The fact that this is not the case indicates that the DEM struggles with the increased training complexity. Unlike with the FEM, a finer mesh does not seem to give smaller boundaries, instead, it seems the opposite is true. It is clear from looking at the cantilever design for both the rough and fine mesh that the final design is not optimal. The bridge with $N = 40$ looks good, but for $N = 160$ there is a thin connection on the right side that is entirely diffuse, so the result is not realizable. Compared to the reference designs in figure 6.10, it is clear that neither the rough nor the fine cantilever resembles the reference internally. For $N = 160$, the short cantilever is close to the reference figure,



(a) Cantilever

(b) Short cantilever

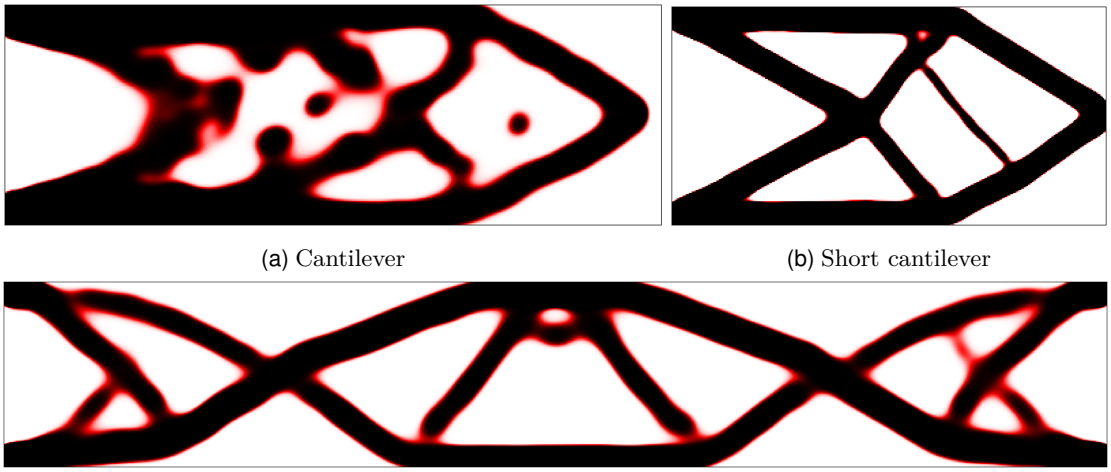
(c) Bridge

Figure 6.6: Figures showcasing the optimized topologies, using the DEM, for three linear elasticity examples; the cantilever shown in 6.6a, the short cantilever shown in 6.6b, and the bridge shown in 6.6c. In all three examples, the discretization parameter $N = 40$ is used. A density of 1, indicating presence of material, is shown in black, and a density of 0, indicating absence of material, is shown in white. Intermediate values are highlighted by coloring densities around 0.5 in red.

just flipped horizontally. For the bridge, the result with the fine mesh looks somewhat similar, but the one with the rough mesh looks very different. It might seem weird that our results for the short cantilever and bridge does not match the result from [24] given how we implemented the method described in that paper, but the differences are explainable. For the bridge, the cause is the discretization we chose combined with the fact that He et al. limited the number of iterations to 80. With $N = 30$ and a limit of 80 iterations the bridge design, shown in figure 6.12, looks almost identical to the reference. For the short cantilever, the fact that the diagonal line is missing for $N = 40$ is also caused by the iteration count. By looking at the intermediate designs during the optimization process, shown in figure 6.13, we can see that the short cantilever had a diagonal line at iteration 53, but it gradually disappeared. For both the $N = 40$ and the $N = 160$ design, the center line is flipped compared to the reference. This might actually be a problem with the reference picture, if we instead look at the cantilever we get when running the source code provided by He et al., shown in figure 6.14, we see a result very similar to the result we got with $N = 160$. The reason the code gave a different result when ran on our machine is probably caused by a difference in random number generation or floating point implementation in the machine we ran the code on.

Comparing the designs from the FEM with the ones with the DEM for the linear elasticity examples, we see that the only design that looks the same between the two is the short cantilever with $N = 40$. As the DEM result has no diffuse boundary, it is probably the superior result. Comparing the bridge with $N = 40$, it is safe to assume that the DEM gave a better result as it does not have disconnected segments. For $N = 160$, the two designs are similar, but as the DEM result has many diffuse parts, the FEM probably gave a better result. For the cantilever, the FEM obviously gave a better result for both $N = 40$ and $N = 160$.

When looking at the results from Stokes flow with the FEM, we see the same trends as in the linear elasticity case. Comparing figure 6.4 for $N = 40$ and figure 6.5 for



(a) Cantilever

(b) Short cantilever

(c) Bridge

Figure 6.7: Figures showcasing the optimized topologies, using the DEM, for three linear elasticity examples; the cantilever shown in 6.7a, the short cantilever shown in 6.7b, and the bridge shown in 6.7c. In all three examples, the discretization parameter $N = 160$ is used. A density of 1, indicating presence of material, is shown in black, and a density of 0, indicating absence of material, is shown in white. Intermediate values are highlighted by coloring densities around 0.5 in red.

$N = 160$, we see that the designs have the same shape, and the finer mesh results in smoother designs. Unlike linear elasticity however, the finer mesh does not seem to have a smaller diffuse boundary. Compared to the reference designs in figure 6.11, we see that all three designs we got look identical to the reference designs, except for a small difference for the twin pipe, where the combined section is longer in the reference than in our result. We cannot tell if our result is better or worse based on the figures alone.

The story is very different for the DEM. For $N = 40$, shown in figure 6.8, we see that the results look mostly right. The diffuser is close to the reference, figure 6.11, but the pipe bend is slightly off, and the twin pipe does not connect at the center. From [32] we get that two separate pipes is a non-optimal minimum, so the twin pipe result is not bad, but it means the inaccuracies in the DEM prevented the optimization from reaching the global minimum. Unlike the linear elasticity case, the Stokes flow designs have a diffuse boundary, but it does not seem to be any bigger than in the FEM case. The fact that the designs are slightly off is not surprising; as we saw when tuning the divergence penalty τ, our way of making the DEM solve the Stokes equations does not give the correct solution, so the design based on those solutions is obviously a bit wrong. Regarding the solutions for $N = 160$, figure 6.5, it is not necessary to study the designs closely to notice that something has gone horribly wrong. This is a more extreme version of what we saw with the linear elasticity examples, so it is clear that the DEM struggles with a finer mesh.

6.4.3 Objectives

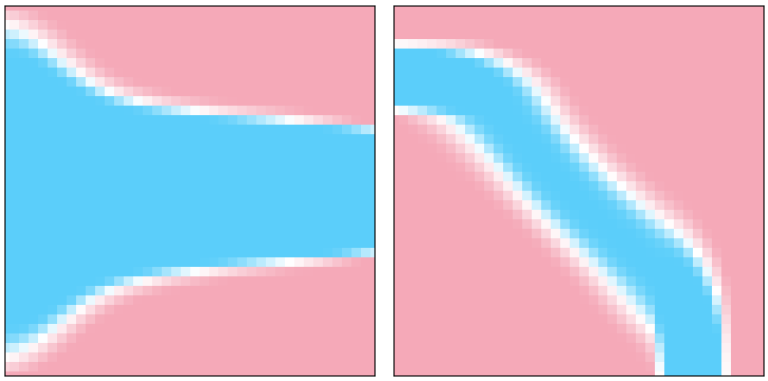
By looking at the table of computed and interpolated objective values for the FEM, table 6.7, we see that, for the cantilever and short cantilever, the computed objectives increase as the mesh is refined. This is probably due to the fact that those examples have a small region where force is applied, which causes it to be underestimated when the mesh is rough. The short cantilever with $N = 40$ is an outlier, it has an objective

Example	N	Objective	Best iter	Total iters	Time	Converged?
Cantilever	40	0.003541	23	74	18 m 33 s	No
	80	0.003775	62	71	33 m 33 s	Yes
	160	0.003979	65	77	12 m 42 s	Yes
	320	0.00423	91	91	13 m 33 s	Yes
Short cantilever	40	19.54	261	263	10 m 33 s	Yes
	80	19.75	200	211	11 m 25 s	Yes
	160	20.32	187	188	9 m 46 s	Yes
	320	20.78	212	213	12 m 34 s	Yes
Bridge	40	295.2	365	369	28 m 33 s	Yes
	80	275.0	629	630	49 m 59 s	Yes
	160	312.7	137	137	12 m 19 s	Yes
	320	338.5	163	163	56 m 27 s	Yes

Table 6.5: A table showing minimum objective value, iteration where the minimum was reached, total iterations, time taken, and if the iteration converged or not. These values are shown for the three linear elasticity examples solved using the DEM. For each example, the results for four discretization parameters are shown; $N = 40$, $N = 80$, $N = 160$ and $N = 320$.

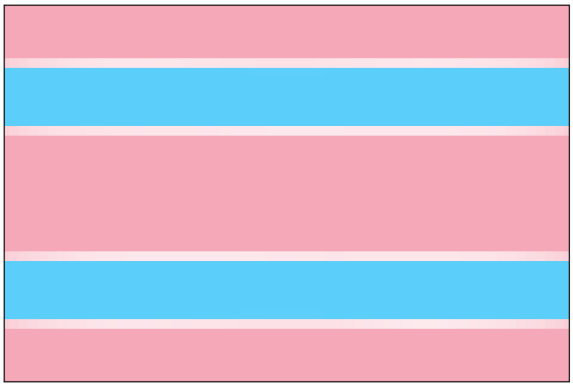
nearly one tenth of the versions with a finer mesh. The short cantilever does have a very small region of applied force, so the FEM probably drastically underestimates it when the mesh is rough. This hypothesis is supported by the Stokes flow examples, as there the computed objective decreases as the mesh is refined. This is what we expect to happen, the rough edges that results from a rough mesh are not optimal, you want smooth lines for both linear elasticity and Stokes flow. The exception here is the twin pipe, where the computed objective increases from $N = 80$ to $N = 160$, but this is due to the fact that the $N = 160$ case did not converge properly. The interpolated objectives decrease as the mesh is refined for all examples, which further supports our hypothesis, as the interpolated results are computed with a constant mesh size. This means the force is calculated equally for all cases, so the only difference between the results is how jagged they are. For Stokes flow, the interpolated objectives are very close to the computed objectives. Looking at all the interpolated objectives, they seem to be converging to some value. If we compute the convergence rate of the absolute differences between increasing N-values, we find that the bridge is converging the slowest with a rate of about $1/N^{0.95\pm0.41}$, and the pipe bend converges fastest with a rate of about $1/N^{3.0\pm0.1}$. The twin pipe example did not converge for $N = 160$, so we cannot really calculate a reliable convergence ratio in that case, but if we did so anyway it would be a barely statistically significant $1/N^{0.36\pm0.28}$. The average convergence rate is $1/N^{1.8\pm0.3}$.

The DEM objectives shown in table 6.8 do not follow the same nice pattern as the FEM objectives. Like the FEM, the computed objectives for the linear elasticity examples increase as the mesh becomes finer, but so does the objectives for the Stokes flow examples. While the interpolated objective for $N = 80$ is always better than the one for $N = 40$, both $N = 160$ and $N = 320$ give significantly worse results, which makes sense given how the final designs looked for $N = 160$. This indicates that $N = 80$ is the best balance between training complexity and smoothness. Unlike the FEM, the short cantilever with $N = 40$ does not have a significantly lower computed objective, which might be because we created an integration method specifically to handle the traction integral. As the interpolated objectives are computed using the FEM, comparing the computed and interpolated objectives is a way of comparing the two methods. From this



(a) Diffuser

(b) Pipe bend

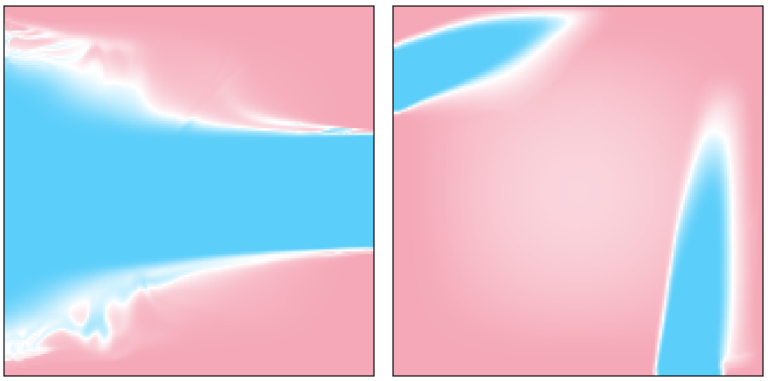


(c) Twin pipe

Figure 6.8: Figures showcasing the optimized topologies, using the DEM, for three Stokes flow examples; the diffuser shown in 6.8a, the pipe bend shown in 6.8b, and the twin pipe shown in 6.8c. In all three examples, the discretization parameter $N = 40$ is used. A density of 1, indicating presence of fluid, is shown in blue, and a density of 0, indicating absence of fluid, is shown in red. Intermediate values are highlighted by coloring densities around 0.5 in white.

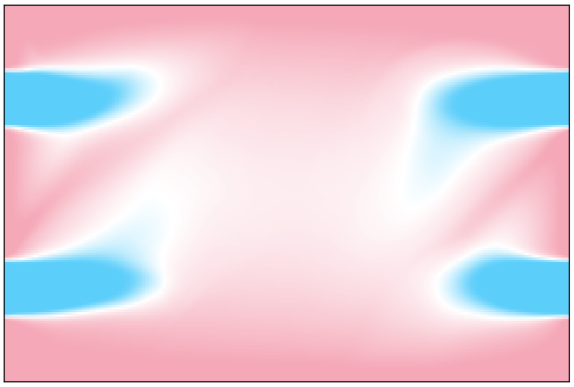
we can see that the objective value for the pipe bend example is very close to the one the FEM gives for both $N = 40$ and $N = 80$. If the DEM can accurately calculate the objective value it must accurately calculate the fluid velocities, but then the resulting design should be the same as with the FEM, which is not the case. We can see something similar for the diffuser, but there the difference is a bit bigger. We do not know why the designs are so different when the objectives are so similar. Looking at just the interpolated objectives, it does not seem like the objective of any example is converging. If we actually compute the convergence rate of the differences, we get that the cantilever is barely converging at a rate of $1/N^{0.68\pm0.37}$, but all the other examples are diverging, with the diffuser diverging at a rate of $N^{3.9\pm0.1}$.

Comparing the interpolated objectives for the FEM with those of the DEM, we find that there are four cases where the DEM gives better results than the FEM. Those are the short cantilever and bridge with $N = 40$ and $N = 80$. This makes sense, the diffuse region the FEM has when the mesh is rough is penalized, so when the two methods converge to the same shape, we would expect the sharp boundaries to give the DEM an advantage. The DEM short cantilever is better, but not by much. For $N = 40$, the difference is just 1.2 %, and for $N = 80$, the difference is an insignificant 0.08 %. The fact that the difference is only caused by the diffuse boundary is why the FEM result



(a) Diffuser

(b) Pipe bend



(c) Twin pipe

Figure 6.9: Figures showcasing the optimized topologies, using the DEM, for three Stokes flow examples; the diffuser shown in 6.9a, the pipe bend shown in 6.9b, and the twin pipe shown in 6.9c. In all three examples, the discretization parameter $N = 160$ is used. A density of 1, indicating presence of fluid, is shown in blue, and a density of 0, indicating absence of fluid, is shown in red. Intermediate values are highlighted by coloring densities around 0.5 in white.

with $N = 320$ is better than all the DEM short cantilever results. For the bridge with $N = 40$, the DEM is 1.65 % better than the DEM, and for $N = 80$, the DEM is 8.45 % better. This indicates that the DEM found a better solution than the DEM, either a better local minimum, or even the global minimum. This is further reinforced by the fact that the FEM result for $N = 320$ is still worse than the DEM result with both $N = 40$ and $N = 80$. The fact that DEM short cantilever with $N = 160$ is worse than the design the FEM got indicates that the diagonal line is suboptimal local minimum. For the other examples, the best DEM diffuser is with $N = 80$, and it is 8.9 % worse than the corresponding FEM diffuser. For the pipe bend, the closest result is with $N = 80$, where the error is 16.6 %, the closest twin pipe is with $N = 40$ and has an error of 34.7 %, and the closest cantilever is with $N = 160$ and has with an error of 34.9 %.

From the reference objectives in table 6.9, we see that the FEM results with $N = 320$ gave almost the same objectives as the reference. The diffuser gave a slightly worse result with an error of 0.03 %, but the pipe bend is 0.07 % better, and the twin pipe is 13 % better than the one from [8]. This indicates that a shorter combined section is better than a long one. If we instead compare with the value for the optimal solution found in [32], we see that our objective is 0.3 % higher than the one they got. The fact that our objectives so closely match the reference objectives indicates that our FEM implementation works

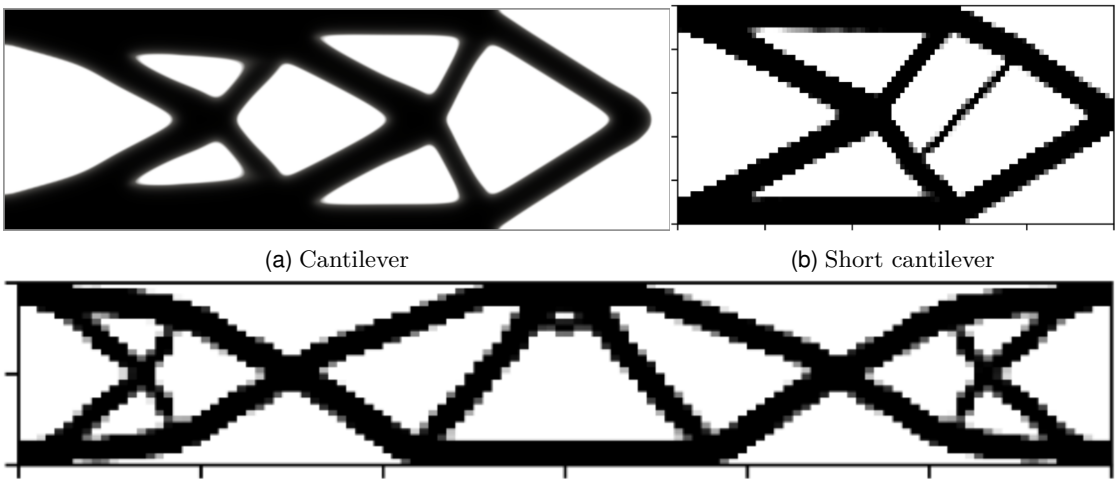
Example	N	Objective	Best iter	Total iters	Time	Converged?
Diffuser	40	35.37	11	11	5 m 8 s	Yes
	80	34.38	13	13	9 m 43 s	Yes
	160	61.57	8	14	13 m 30 s	No
	320	274.6	5	9	11 m 54 s	No
Pipe bend	40	14.91	21	21	10 m 19 s	Yes
	80	11.41	23	23	19 m 53 s	Yes
	160	27.48	15	15	11 m 17 s	Yes
	320	40.00	18	18	15 m 37 s	Yes
twin pipe $q = 0.01$	40	18.85	19	20	6 m 9 s	Yes
	80	18.86	12	12	6 m 6 s	Yes
	160	19.74	13	13	15 m 42 s	Yes
	320	21.37	12	12	7 m 59 s	Yes
twin pipe $q = 0.1$	40	28.73	11	11	1 m 46 s	Yes
	80	27.49	17	29	9 m 33 s	Yes
	160	41.70	10	10	1 m 37 s	Yes
	320	51.58	8	8	1 m 0 s	Yes

Table 6.6: A table showing minimum objective value, iteration where the minimum was reached, total iterations, time taken, and if the iteration converged or not. These values are shown for the three Stokes flow examples solved using the DEM. For each example, the results for four discretization parameters are shown; $N = 40$, $N = 80$, $N = 160$ and $N = 320$.

as it should, and that the EMD algorithm works well for solving fluid based topology optimization problems. For the DEM, using the interpolated objectives for the mesh size that gave the best results, $N = 80$, we get that the objective for the diffuser is 9.2% higher, for the pipe bend is 17% higher, and for the twin pipes is 21% higher than in [8]. This matches our observations of the designs; the diffuser looked quite close while the pipe bend and twin pipe looked quite different. Comparing the value for the twin pipe example with the one in [8] is not fair as we are comparing a suboptimal local minimum with the global minimum. If we instead compare the objective to the suboptimal local minimum in [32], we get an error of just 3%.

6.4.4 Speed

By looking at table 6.3 of the results for the FEM and linear elasticity, we can see that our program is quite fast for rough meshes, with the cantilever being the fastest at around three minutes for $N = 40$, and the bridge being the slowest at about six minutes for $N = 40$. The bridge being the slowest makes sense as it has a width that is significantly larger than its height. As we have seen from the interpolated objectives, the design the FEM gives is quite good even with a rough mesh, with the cantilever giving a result that is just 1.6 % larger, so a rough mesh is a good way of getting approximate results quickly. The speed depends on the size of the domain, which makes sense as doubling the length will also double the number of elements in the mesh. This means that the program is very slow with $N = 320$, with the cantilever example taking almost two and a half hours, and the bridge taking a bit more than a full day. It seems like the time the program takes increases proportional to about N^2, which is what we would expect. As the number of iterations taken also seem to increase with N, the exponent is a bit more than 2, being 2.5 ± 0.1 for the bridge and 3.0 ± 0.5 for the short cantilever.



(a) Cantilever

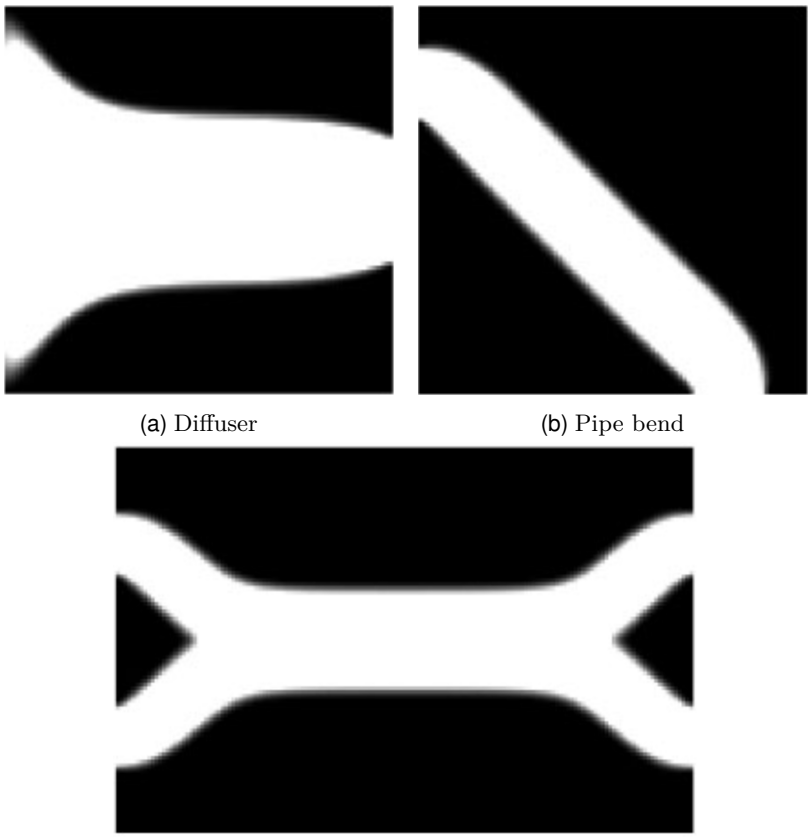
(b) Short cantilever

(c) Bridge

Figure 6.10: Figures showcasing the optimized topologies for three linear elasticity examples; the cantilever shown in 6.10a, the short cantilever shown in 6.10b, and the bridge shown in 6.10c. The cantilever comes from figure 6.4 from [25], where the FEM was used, and we have added a gray border to show Ω. The short cantilever and bridge comes from figure 2 and 3 in [24], where the DEM was used. The cantilever used the discretization parameter $h = 1/128$, corresponding to $N = 128$. The short cantilever used a 91-by-46 grid, corresponding to $N = 45$. The bridge used an 121-by-31 grid, which results in a rectangular grid, and therefore no singular N value. Instead, it has $N_x = 30$ and $N_y = 20$ in the x-direction and y-direction respectively. A density of 1, indicating presence of material, is shown in black, and a density of 0, indicating absence of material, is shown in white. Intermediate values are not highlighted, they are instead shown as shades of gray.

Interestingly enough, the exponent for the cantilever is less than 2, being 1.8 ± 0.1. This is probably a result of the fact that the time a program takes to run is affected by several external factors, so the uncertainty of the exponent is high. The table for Stokes flow, 6.4, tells us the same story, but the times are all much smaller. This is a combination of three factors. The first one is the domain size; both the diffuser and pipe bend are square, while the twin pipe has a width of 1.5. The second one is the iteration count; the diffuser used around 12 iterations and the pipe bend used around 20. The exception here is the twin pipe, which used many iterations for the refinement step. The third cause is the fact that you only need to solve one PDE per iteration, unlike with linear elasticity where you need to filter both the design and the gradient, resulting in two additional PDE solves. These effects combined means that the diffuser and pipe bend use around four seconds for $N = 40$. Even the $N = 320$ case is quite fast, with the diffuser taking around five minutes, and the pipe bend using about nine minutes. The twin pipe is still quite fast for $N = 40$, but for $N = 80$ and $N = 320$ it is slower than the cantilever. As the number of iterations is roughly constant for both the diffuser and pipe bend, the time they take increases at roughly the same rate, being proportional to $N^{2.0 \pm 0.2}$ for the pipe bend and $N^{1.9 \pm 0.2}$ for the diffuser. Here the exponent is again less than two, but this time it is not statistically significant. Given how much faster a rough mesh is, an idea for a faster implementation for fine meshes is to first quickly find the optimal rough design, and then use that as the initial design for a new run with a finer mesh.

Once again, the DEM results are very different. By looking at table 6.5 and 6.6, for the results for linear elasticity and Stokes flow respectively, we find that a finer mesh



(a) Diffuser (b) Pipe bend

(c) Twin pipe

Figure 6.11: Figures showcasing the optimized topologies for three Stokes flow examples; the diffuser shown in 6.11a, the pipe bend shown in 6.11b, and the twin pipe shown in 6.11c. All three figures are from [8], where the FEM was used. The diffuser comes from figure 5, the pipe bend from figure 7 and the twin pipe from figure 11. In all three examples, the discretization parameter $N = 100$ was used. A density of 1, indicating presence of fluid, is shown in white, and a density of 0, indicating absence of fluid, is shown in black. Intermediate values are not highlighted, they are instead shown as shades of gray.

does not result in a meaningfully slower time. The average exponent is 0.60 ± 0.28, so the time taken does in fact increase with N, but the exponent is small and the uncertainty is large. However, while the time taken might be almost constant, it is still quite slow. The DEM is the most competitive for linear elasticity, where it becomes faster than the bridge and short cantilever at $N = 80$, and faster than the cantilever at $N = 160$. For Stokes flow however, the FEM is still faster than the DEM with $N = 320$ for both the diffuser and pipe bend, and the DEM is only faster for the twin pipe with $N = 320$. This might indicate that, if you want to run a simulation with for instance $N = 1280$, the DEM is the clear choice. However, as we have seen previously, the DEM struggles with a fine mesh, so such a fine mesh might give unusable results.

6.4.5 Mesh Independence

Most of our results are not mesh independent. For linear elasticity, the FEM with the cantilever is the closest, where the number if iterations increases by just one or two as the mesh is refined. It is therefore weakly mesh dependent. For the other two examples, the number of iterations increases quickly. For Stokes flow, the FEM with the diffuser

State equation	Example	N	Computed ϕ	Interpolated ϕ
Linear elasticity	Cantilever	40	0.003711	0.004054
		80	0.003868	0.004008
		160	0.003957	0.003996
		320	0.003992	0.003992
	Short cantilever	40	2.910	23.83
		80	17.59	23.21
		160	21.14	23.09
		320	23.05	23.05
	Bridge	40	325.7	305.7
		80	310.9	302.8
		160	304.5	301.9
		320	301.1	301.1
Stokes flow	Diffuser	40	31.03	31.04
		80	30.54	30.55
		160	30.46	30.46
		320	30.45	30.45
	Pipe bend	40	10.27	10.26
		80	9.836	9.833
		160	9.774	9.774
		320	9.767	9.767
	Twin pipe	40	25.80	25.82
		80	24.01	24.02
		160	25.02	25.02
		320	23.93	23.93

Table 6.7: A table showing the best objective reached by the FEM for every example, labeled "Computed ϕ", and the objective calculated by interpolating the design and recalculating the objective using the FEM with $N = 320$, labeled "Interpolated ϕ".

and pipe bend seems to be almost completely mesh independent. Mesh dependence is a statement on asymptotic behavior, so, assuming that the number of iterations for the pipe bend will stay constant with further mesh refinements, it is fully mesh independent. For the DEM, the number of iterations jumps around for both linear elasticity and Stokes flow, so it is not mesh independent.

6.5 General Discussion

From our results, it seems clear that the FEM performed better than the DEM, but how much better depends on if you are using linear elasticity or Stokes flow. With linear elasticity, the DEM did outperform the FEM on two of the three examples when the mesh was rough, and was significantly faster than the FEM when the mesh was fine. We did however see that the DEM struggles with fine meshes, so it being faster is meaningless when the results it produces are significantly worse. The results for the rough mesh being better is rendered moot by the time the DEM takes to get those results; you would mostly use a rough mesh when testing or prototyping, which are both cases where the speed of the method is very important. This means you would rather have a fast algorithm that gives slightly worse results instead of a slow algorithm that

State equation	Example	N	Computed ϕ	Interpolated ϕ
Linear elasticity	Cantilever	40	0.003541	0.00973
		80	0.003775	0.007536
		160	0.003979	0.005392
		320	0.00423	0.006253
	Short cantilever	40	19.54	23.54
		80	19.75	23.19
		160	20.32	23.72
		320	20.78	24.29
	Bridge	40	295.2	300.7
		80	275.0	277.2
		160	312.7	316.7
		320	338.5	345.0
Stokes flow	Diffuser	40	35.37	34.12
		80	34.38	33.26
		160	61.57	47.93
		320	274.6	227.4
	Pipe bend	40	14.91	14.75
		80	11.41	11.46
		160	27.48	68.28
		320	40.00	77.51
	Twin pipe	40	28.73	34.78
		80	27.49	33.56
		160	41.70	111.8
		320	51.58	178.9

Table 6.8: A table showing the best objective reached by the DEM for every example, labeled "Computed ϕ", and the objective calculated by interpolating the design and recalculating the objective using the FEM with $N = 320$, labeled "Interpolated ϕ".

gives slightly better results, so in that case the FEM is still better. The exception to this is the bridge example, where the design the DEM got for a rough mesh was better than the one the FEM got with a fine mesh. However, this was caused by the FEM converging to a different design than the DEM, which, as mentioned previously, is not the fault of the FEM. If we had used a more advanced minimization algorithm that can find multiple minima, like the one described in [32], both methods might give both designs. In that case, the increased smoothness of the FEM design would make it superior to the DEM one. The fact that the DEM gave such bad results for the cantilever example is probably a problem with the hyperparameters; the hyperparameters we used were optimized for the two examples in [24], which use a larger domain, larger force and larger Young's modulus than the cantilever example, so the optimal hyperparameters for those two might not be optimal for the cantilever. If this is the case, it would be a large downside of the DEM. If the optimal hyperparameters are dependent on the specific problem you want to solve, the hyperparameter optimization should really be included in the time taken by the method, making it even slower than it already is.

For the results with Stokes flow, the DEM has no benefits compared to the FEM. The only case where it gave an acceptable result was the twin pipe with a rough mesh, in all other cases the results it converged to are far from the true minima. The bad

Example	Objective
Diffuser	30.46
Pipe bend	9.76
Twin pipe from [8]	27.64
Twin pipe from [32], combined	23.87
Twin pipe from [32], separate	32.58

Table 6.9: A table showing objective values from reference solutions for various examples. The objective for the diffuser and pipe bend come from table I and II in [8] respectively. Three separate objectives for the twin pipe example is shown. The first comes from table IV in [8], while the other two comes from the description of figure 6 in [32]. "Combined" means the pipes join in the center, which is the global minimum, and "separate" means that the pipes do not join, which is a local minimum.

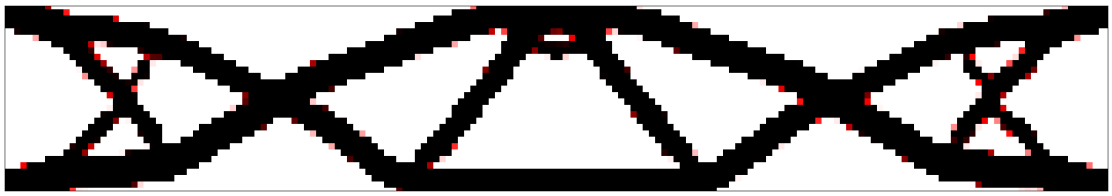


Figure 6.12: A figure of the optimized design of the bridge example using the DEM limited to 80 iterations and with $N = 30$.

performance of the DEM is most likely a result of the cost function we chose, but we do not know of a better way of defining it. We initially tried to use the full energy functional (3.12), but we noticed that the cost kept decreasing without ever converging. We hypothesized that this was due to the DEM struggling to enforce a relationship between the pressure and the fluid velocities, so if we used a pressure free approach, the results would improve, and this is indeed what we saw. The problem with the pressure free approach is that the divergence is never truly zero, as making the penalization parameter too big causes the DEM to ignore the first part of the energy functional. The ideal solution would be to use an approach similar to the additive decomposition used to enforce Dirichlet boundary conditions. If we could find a function $\boldsymbol{\xi}(\boldsymbol{u})$ such that $\nabla \cdot \boldsymbol{\xi}(\boldsymbol{u}) = 0$ and $\boldsymbol{\xi}(\boldsymbol{u}) = \boldsymbol{u}$ on $\delta\Omega$, we could just use $\boldsymbol{u}_{NN} = \boldsymbol{\xi}(\boldsymbol{m} \odot \tilde{\boldsymbol{u}}_{NN} + \boldsymbol{g})$ and safely use the pressure-free functional. Unfortunately, we do not know of such a function. However, even if such a function could be found, we are confident the FEM would still outperform the DEM. The FEM solved the Stokes flow examples incredibly quickly, so even if the DEM could get comparable objective values, it would probably still be significantly slower.

The fact that the DEM has an almost constant time complexity is interesting, and it means the DEM could be a useful method if its other flaws could be fixed. The way you typically improve neural network models is by tweaking the architecture, changing the amount of layers, the type of layers and how the layers are connected to each other. The architecture we used was very simple, so it would not be surprising if a different architecture could give better results. However, a more complicated architecture would increase the computational complexity of the model, making it even slower than it already is. The fact that we cannot know which architecture is the best is a flaw of neural networks in general; there is no rigor in how architectures are chosen, your only option is to try many architectures and choose the one that performs the best. The FEM on the other hand is a well studied method with much mathematical rigor. We know

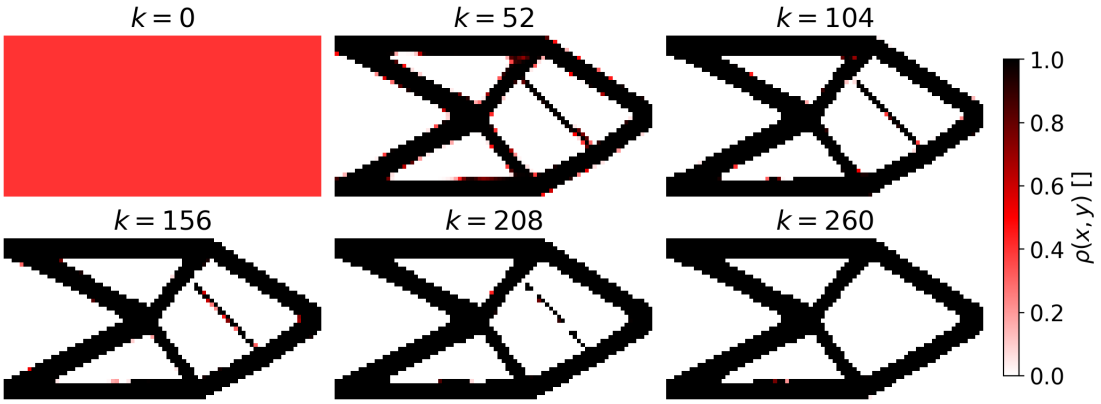


Figure 6.13: A figure showcasing the intermediate designs when optimizing the short cantilever with the DEM and $N = 40$.

Figure 6.14: A figure of the optimized design for the short cantilever that you get when running the source code provided by [24].

for instance that the diffuse boundary the FEM gave for the linear elasticity examples is due to the fact that we used piecewise linear polynomials, and that a different function space, such as discontinuous piecewise constants, would give sharper boundaries. Due to the mathematical rigor we know the benefits and drawbacks of the various function spaces, and can choose the one that works best for the problem we want to solve. For the DEM, you can never know if the architecture you have found is optimal. This is true for the hyperparameters in general. The lack of a guaranteed minimum is also a problem when training a neural network. The FEM turns the PDE into a linear equation, which is guaranteed to have a unique solution as long as the matrix A is not singular. The DEM on the other hand uses a gradient based minimization method to optimize the parameters, which is a very non-convex optimization problem. This means that the solution the DEM gives is very sensitive to random variations, and there is no guarantee that the training converges to the global minimum. This is also a problem when looking at the limiting case. The finite element subspace S_h is defined such that as $h \to 0$, the FEM solution converges to the true solution. We have no such guarantees with a neural network. One would assume that the solution the DEM gives converges to the true solution as the number of layers and amount of training data increases, but we cannot guarantee that it will.

A way of improving our DEM implementation would be to use both training and testing data, and training the network in a way that ensures good results on the testing data. If this is done, we could evaluate the neural network on any point in the domain, giving a high resolution design even with few training points. This would potentially be a big improvement, the higher resolution would make designs with a rough mesh smoother, making them comparable to the FEM results with a fine mesh. However, as we have seen,

the DEM struggles with a large amount of training data, so this approach might make the DEM struggle to train properly. A way of improving our FEM implementation would be to use the iterative refinement approach mentioned in section 6.4.4, which should make the FEM significantly faster for fine meshes, alleviating the main drawback of the method.

Chapter 7

Conclusion

In this thesis, we have developed a program that can solve topology optimization problems based on both the equations of linear elasticity and the Stokes equations. The program can use both the finite element method and the deep energy method to solve the underlying state equations. Using this program, we compared the performance of the two methods on a total of six examples; three for linear elasticity, and three for Stokes flow. As a product of this thesis, we have developed two novel methods for solving Stokes based topology optimization problems; one using entropic mirror descent combined with the FEM, and one using EMD combined with the DEM.

We have found that our FEM implementation is superior to our DEM implementation. The main benefit of the FEM is that it gives the same shape on both rough and fine meshes, which means refining the mesh improves the design, making it smoother and reducing the size of the diffuse boundary. For the DEM on the other hand, increasing the mesh size, that is, increasing the number of points where the model is sampled, results in a worse result. This means that, although the DEM does give nice sharp boundaries for a rough mesh, it cannot produce smooth versions of the same designs. The main benefit of the DEM is that its time complexity is almost constant, unlike the FEM, where the time taken increases proportionally to about $N^{2.2}$, which makes it slow for very fine meshes. The fact that the results from the DEM get worse as the mesh becomes finer does however mean this benefit is mostly meaningless. The FEM is significantly faster than the DEM for a rough mesh, meaning that even though the DEM sometimes gives slightly better results in that case, the FEM is more useful for prototyping and testing. Our novel Stokes based topology optimization solver using the FEM proved to be very fast, and gave results comparable to those from the existing literature. It does however have some sort of instability which caused the twin pipe example to diverge with $N = 160$, which needs to be fixed for the method to be truly useful. We have found that the instability can be avoided by changing the step size a bit, so there does exist a workaround which can be used. Our novel solver using the DEM did not give satisfactory results. For a rough mesh it gave results that were close but not quite right, and for a fine mesh it gave terrible results with both the pipe bend and twin pipes being entirely disconnected in the middle.

There were two cases where the methods gave designs from different minima. This makes comparing the two methods for those cases difficult. It would therefore be interesting to repeat this comparison using a more advanced minimizer that can converge to multiple minima, such as the one described in [32]. It would also be interesting to extend the algorithm to use a line search to find the optimal step size at each iteration, or even use the Hessian of the objective functionals combined with a trust region method

[14].

We have compared the two methods on simple two-dimensional examples. For future work, more complicated applications could be examined, such as three-dimensional topology optimization, the full Navier-Stokes equations, or optimizing a topology that includes multiple materials using multi-material topology optimization [48].

The fact that our DEM implementation has an almost constant time complexity is not a novel result. The DEM is known to be useful for high-dimensional problems such as high-dimensional optimal control problems, for instance multiagent path finding [31] or stochastic optimal control [29]. This does mean that the DEM is the obvious choice for topology optimization with thousands of dimensions, but creating real world shapes only requires three-dimensional topology optimization, so we doubt the DEM is ever going to outperform the FEM for structural design.

Bibliography

[1] David J. Acheson. "The Navier-Stokes equations." In: *Elementary Fluid Dynamics*. Clarendon Press, 1990, pp. 201–216. URL: https://global.oup.com/academic/product/elementary-fluid-dynamics-9780198596790.

[2] Grégoire Allaire, François Jouve, and Anca-Maria Toader. "A level-set method for shape optimization." In: *Comptes Rendus Mathematique* 334 (2002), pp. 1125–1130. URL: https://www.sciencedirect.com/science/article/pii/S1631073X02024123.

[3] Patrick Amestoy et al. "Multifrontal Method." In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1209–1216. URL: https://doi.org/10.1007/978-0-387-09766-4_86.

[4] Martin P. Bendsøe. "Optimal shape design as a material distribution problem." In: *Structural optimization* 1 (1989), pp. 193–202. URL: https://doi.org/10.1007/BF01650949.

[5] Martin P. Bendsøe and Noboru Kikuchi. "Generating optimal topologies in structural design using a homogenization method." In: *Computer Methods in Applied Mechanics and Engineering* 71 (1988), pp. 197–224. URL: https://www.sciencedirect.com/science/article/pii/0045782588900862.

[6] Martin P. Bendsøe and Ole Sigmund. "Material interpolation schemes in topology optimization." In: *Archive of Applied Mechanics* 69 (1999), pp. 635–654. URL: https://doi.org/10.1007/s004190050248.

[7] James Bergstra, Daniel Yamins, and David D. Cox. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures." In: *ICML* (2013). URL: http://proceedings.mlr.press/v28/bergstra13.pdf.

[8] Thomas Borrvall and Joakim Petersson. "Topology optimization of fluids in Stokes flow." In: *International Journal for Numerical Methods in Fluids* 41 (2003), pp. 77–107. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.426.

[9] Blaise Bourdin and Antonin Chambolle. "Design-dependent loads in topology optimization." In: *ESAIM: Control, Optimisation and Calculus of Variations* 9 (2003), pp. 19–48. URL: http://www.numdam.org/articles/10.1051/cocv:2002070/.

[10] Richard P. Brent. "An algorithm with guaranteed convergence for finding a zero of a function." In: *The Computer Journal* 14 (1971), pp. 422–425. URL: https://doi.org/10.1093/comjnl/14.4.422.

[11] Shengze Cai et al. *Physics-informed neural networks (PINNs) for fluid mechanics: A review*. 2021. arXiv: 2105.09506 [physics.flu-dyn].

[12] Long Chen. "A simple construction of a Fortin operator for the two dimensional Taylor-Hood element." In: *Computers & Mathematics with Applications* 68 (2014), pp. 1368–1373. URL: https://www.sciencedirect.com/science/article/pii/S0898122114004519.

[13] Philippe G. Ciarlet. "Existence Theory Based on the Implicit Function Theorem." In: *Mathematical Elasticity: Three-Dimensional Elasticity*. Elsevier, 1988, pp. 269–344. URL: https://sciencedirect.com/science/book/9780444702593.

[14] Andrew R. Conn, Nicholas I. M. Gould, and Philippe L. Toint. *Trust Region Methods*. MOS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics (SIAM), 2000. URL: https://epubs.siam.org/doi/book/10.1137/1.9780898719857.

[15] George Cybenko. "Approximation by superpositions of a sigmoidal function." In: *Mathematics of Control, Signals and Systems* 2 (1989), pp. 303–314. URL: https://doi.org/10.1007/BF02551274.

[16] Iain S. Duff and John K. Reid. "The Multifrontal Solution of Indefinite Sparse Symmetric Linear." In: *ACM Transactions on Mathematical Software* 9 (1983), pp. 302–325. URL: https://doi.org/10.1145/356044.356047.

[17] Lawrence C. Evans. "Sobolev Spaces." In: *Partial Differential Equations*. Vol. 19. American Mathematical Society, 1998, pp. 251–308.

[18] Lawrence C. Evans. "The Calculus of Variations." In: *Partial Differential Equations*. Vol. 19. American Mathematical Society, 1998, pp. 456–519.

[19] Roger Fletcher. "Newton-Like Methods." In: *Practical Methods of Optimization*. John Wiley & Sons, Ltd, 2000, pp. 44–79. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118723203.ch3.

[20] John T. Foster. *Direct Methods for Solving Linear Systems of Equations*. https://johnfoster.pge.utexas.edu/numerical-methods-book/LinearAlgebra_DirectSolvers.html. Accessed: 2024-03-07.

[21] The Linux Foundation. *PyTorch*. https://pytorch.org/. Accessed: 2023-12-23.

[22] Allan Gersborg-Hansen, Ole Sigmund, and Robert B. Haber. "Topology optimization of channel flow problems." In: *Structural and Multidisciplinary Optimization* 30 (2005), pp. 181–192. URL: https://doi.org/10.1007/s00158-004-0508-7.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. "Machine Learning Basics." In: *Deep Learning*. MIT Press, 2016, pp. 96–161. URL: http://www.deeplearningbook.org.

[24] Junyan He et al. "Deep energy method in topology optimization applications." In: *Acta Mechanica* 234 (2022), pp. 1365–1379. URL: http://dx.doi.org/10.1007/s00707-022-03449-3.

[25] Brendan Keith and Thomas M. Surowiec. *Proximal Galerkin: A structure-preserving finite element method for pointwise bound constraints*. 2023. arXiv: 2307.12444 [math.NA].

[26] Ehsan Kharazmi, Zhongqiang Zhang, and George E. Karniadakis. "Variational Physics-Informed Neural Networks For Solving Partial Differential Equations." In: *CoRR* abs/1912.00873 (2019). URL: http://arxiv.org/abs/1912.00873.

[27] Mats G. Larson and Fredrik Bengzon. "Piecewise Polynomial Approximation in 2D." In: *The Finite Element Method: Theory, Implementation, and Applications*. Springer-Verlag, 2014, pp. 45–69. URL: https://doi.org/10.1007/978-3-642-33287-6.

[28] Boyan S. Lazarov and Ole Sigmund. "Filters in topology optimization based on Helmholtz-type differential equations." In: *International Journal for Numerical Methods in Engineering* 86 (2011), pp. 765–781. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.3072.

[29] Xingjian Li, Deepanshu Verma, and Lars Ruthotto. *A Neural Network Approach for Stochastic Optimal Control*. 2023. arXiv: 2209.13104 [math.OC].

[30] Jikai Liu et al. "Current and future trends in topology optimization for additive manufacturing." In: *Structural and Multidisciplinary Optimization* 57 (2018), pp. 2457–2483. URL: https://doi.org/10.1007/s00158-018-1994-3.

[31] Derek Onken et al. "A Neural Network Approach for High-Dimensional Optimal Control Applied to Multiagent Path Finding." In: *IEEE Transactions on Control Systems Technology* 31 (2023), pp. 235–251. URL: http://dx.doi.org/10.1109/TCST.2022.3172872.

[32] Ioannis P. A. Papadopoulos, Patrick E. Farrell, and Thomas M. Surowiec. *Computing multiple solutions of topology optimization problems*. 2021. arXiv: 2004.11797 [math.NA].

[33] FEniCS Project. *The FEniCSx computing platform*. https://fenicsproject.org/. Accessed: 2023-09-22. 2021.

[34] Maziar Raissi, Paris Perdikaris, and George E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations." In: *Journal of Computational Physics* 378 (2019), pp. 686–707. URL: https://www.sciencedirect.com/science/article/pii/S0021999118307125.

[35] Patrick J. Roache. "The Method of Manufactured Solutions for Code Verification." In: *Computer Simulation Validation: Fundamental Concepts, Methodological Frameworks, and Philosophical Perspectives*. Ed. by Claus Beisbart and Nicole J. Saam. Cham: Springer International Publishing, 2019, pp. 295–318. URL: https://doi.org/10.1007/978-3-319-70766-2_12.

[36] Esteban Samaniego et al. "An energy approach to the solution of partial differential equations in computational mechanics via machine learning: Concepts, implementation and applications." In: *Computer Methods in Applied Mechanics and Engineering* 362 (2020), p. 112790. URL: http://dx.doi.org/10.1016/j.cma.2019.112790.

[37] Ole Sigmund and Kurt Maute. "Topology optimization approaches." In: *Structural and Multidisciplinary Optimization* 48 (2013), pp. 1031–1055. URL: https://doi.org/10.1007/s00158-013-0978-6.

[38] William S. Slaughter. "Constitutive Equations." In: *The Linearized Theory of Elasticity*. Birkhäuser, 2001, pp. 193–220. URL: https://doi.org/10.1007/978-1-4612-0093-2.

[39] William S. Slaughter. "Forces and Stress." In: *The Linearized Theory of Elasticity*. Birkhäuser, 2001, pp. 157–192. URL: https://doi.org/10.1007/978-1-4612-0093-2.

[40] William S. Slaughter. "Kinematics." In: *The Linearized Theory of Elasticity*. Birkhäuser, 2001, pp. 97–156. URL: https://doi.org/10.1007/978-1-4612-0093-2.

[41] Jan. Sokolowski and Antoni Zochowski. "On the Topological Derivative in Shape Optimization." In: *SIAM Journal on Control and Optimization* 37 (1999), pp. 1251–1272. URL: https://doi.org/10.1137/S0363012997323230.

[42] Krister Svanberg. "The method of moving asymptotes—a new method for structural optimization." In: *International Journal for Numerical Methods in Engineering* 24 (1987), pp. 359–373. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.1620240207.

[43] Mumps Technologies. *MUMPS: a parallel sparse direct solver*. https://mumps-solver.org/index.php. Accessed: 2024-02-24.

[44] Sifan Wang, Hanwen Wang, and Paris Perdikaris. "On the eigenvector bias of Fourier feature networks: From regression to solving multi-scale PDEs with physics-informed neural networks." In: *Computer Methods in Applied Mechanics and Engineering* 384 (2021), p. 113938. URL: https://www.sciencedirect.com/science/article/pii/S0045782521002759.

[45] Rebekka V. Woldseth et al. "On the use of artificial neural networks in topology optimisation." In: *Structural and Multidisciplinary Optimization* 65 (2022), p. 294. URL: https://doi.org/10.1007/s00158-022-03347-1.

[46] Cheng A. Yan, Riccardo Vescovini, and Lorenzo Dozio. "A framework based on physics-informed neural networks and extreme learning for the analysis of composite structures." In: *Computers & Structures* 265 (2022), p. 106761. URL: https://doi.org/10.1016/j.compstruc.2022.106761.

[47] Ji-Hong Zhu, Wei-Hong Zhang, and Liang Xia. "Topology Optimization in Aircraft and Aerospace Structures Design." In: *Archives of Computational Methods in Engineering* 23 (2016), pp. 595–622. URL: https://doi.org/10.1007/s11831-015-9151-2.

[48] Wenjie Zuo and Kazuhiro Saitou. "Multi-material topology optimization using ordered SIMP interpolation." In: *Structural and Multidisciplinary Optimization* 55 (2017), pp. 477–491. URL: https://doi.org/10.1007/s00158-016-1513-3.

Appendix A

Link to Our Source Code

The repository for the code developed for this thesis is available at https://github.com/Emilinya/topomax. At the time of writing, the latest commit is 6031bb7.

Appendix B

Gradient Calculation

We want to show that the gradient of

$$\phi(\rho) = \frac{1}{2}\int_\Omega r(\rho)||\boldsymbol{u}_\rho||^2 + \mu||\nabla \boldsymbol{u}_\rho||^2 \,\mathrm{d}\boldsymbol{x}\,,$$

where $\boldsymbol{u}_\rho$ is the solution to the Stokes equations with varying permeability, is

$$\nabla\phi(\rho) = \frac{1}{2}r'(\rho)||\boldsymbol{u}_\rho||^2.$$

As this appendix is intended as a sketch of the calculation, we will proceed semi-formally and comment on the potential functional analysis nuances at the end.

To calculate the gradient of ϕ, we use the directional derivative, defined as

$$\lim_{t\to 0}\frac{\phi(\rho+t\delta\rho)-\phi(\rho)}{t} = \lim_{t\to 0}\frac{1}{2t}\int_\Omega \Big(r(\rho+t\delta\rho)||\boldsymbol{u}_t||^2 - r(\rho)||\boldsymbol{u}_\rho||^2\Big)+\mu\Big(||\nabla\boldsymbol{u}_t||^2 - ||\nabla\boldsymbol{u}_\rho||^2\Big)\,\mathrm{d}\boldsymbol{x}\,,$$

where $\boldsymbol{u}_t = \boldsymbol{u}_{\rho+t\delta\rho}$ and $\delta\rho$ is some direction. Using the fact that

$$\phi(\rho) = \min_{\boldsymbol{u}\in U_{\mathrm{div}}}\psi(\boldsymbol{u};\rho),$$

we get that $\psi(\boldsymbol{u}_t;\rho)\geq\phi(\rho)$ and $\psi(\boldsymbol{u}_\rho;\rho+t\delta\rho)\geq\phi(\rho+t\delta\rho)$. We therefore get the two inequalities:

$$\begin{aligned}
\frac{\phi(\rho+t\delta\rho)-\phi(\rho)}{t} &\leq \frac{1}{2t}\int_\Omega \Big(r(\rho+t\delta\rho)||\boldsymbol{u}_\rho||^2 - r(\rho)||\boldsymbol{u}_\rho||^2\Big)+\mu\Big(||\nabla\boldsymbol{u}_\rho||^2 - ||\nabla\boldsymbol{u}_\rho||^2\Big)\,\mathrm{d}\boldsymbol{x}\\
&= \frac{1}{2}\int_\Omega \frac{r(\rho+t\delta\rho)-r(\rho)}{t}||\boldsymbol{u}_\rho||^2\,\mathrm{d}\boldsymbol{x}\,,\\
\frac{\phi(\rho+t\delta\rho)-\phi(\rho)}{t} &\geq \frac{1}{2t}\int_\Omega \Big(r(\rho+t\delta\rho)||\boldsymbol{u}_t||^2 - r(\rho)||\boldsymbol{u}_t||^2\Big)+\mu\Big(||\nabla\boldsymbol{u}_t||^2 - ||\nabla\boldsymbol{u}_t||^2\Big)\,\mathrm{d}\boldsymbol{x}\\
&= \frac{1}{2}\int_\Omega \frac{r(\rho+t\delta\rho)-r(\rho)}{t}||\boldsymbol{u}_t||^2\,\mathrm{d}\boldsymbol{x}\,,
\end{aligned}$$

To calculate the directional derivative of r, we use the Taylor expansion:

$$\begin{aligned}
\lim_{t\to 0}\frac{r(\rho+t\delta\rho)-r(\rho)}{t} &= \lim_{t\to 0}\frac{r(\rho)+r'(\rho)t\delta\rho+O(t^2)-r(\rho)}{t}\\
&= r'(\rho)\delta\rho+\lim_{t\to 0}O(t)\\
&= r'(\rho)\delta\rho.
\end{aligned}\tag{B.1}$$

Assuming that $\boldsymbol{u}_t$ converges to $\boldsymbol{u}_\rho$ in at least L_2, we get

$$\lim_{t\to 0} \frac{1}{2} \int_\Omega \frac{r(\rho + t\delta\rho) - r(\rho)}{t} ||\boldsymbol{u}_\rho||^2 \,\mathrm{d}\boldsymbol{x} = \frac{1}{2} \int_\Omega r'(\rho) ||\boldsymbol{u}_\rho||^2 \delta\rho \,\mathrm{d}\boldsymbol{x}$$
$$\lim_{t\to 0} \frac{1}{2} \int_\Omega \frac{r(\rho + t\delta\rho) - r(\rho)}{t} ||\boldsymbol{u}_t||^2 \,\mathrm{d}\boldsymbol{x} = \frac{1}{2} \int_\Omega r'(\rho) ||\boldsymbol{u}_\rho||^2 \delta\rho \,\mathrm{d}\boldsymbol{x}$$

The squeeze theorem then gives us the equality

$$\lim_{t\to 0} \frac{\phi(\rho + t\delta\rho) - \phi(\rho)}{t} = \frac{1}{2} \int_\Omega r'(\rho) ||\boldsymbol{u}_\rho||^2 \delta\rho \,\mathrm{d}\boldsymbol{x}\,.$$

To get the gradient of ρ, we can use the fact that it is the Riesz representation of the directional derivative, meaning it is the function $\nabla\phi$ such that

$$\int_\Omega \nabla\phi(\rho)\delta\rho \,\mathrm{d}x = \lim_{t\to 0} \frac{\phi(\rho + t\delta\rho) - \phi(\rho)}{t} \quad \forall \delta\rho \in L_\infty(\Omega).$$

Clearly, this means $\nabla\phi(\rho) = \frac{1}{2} r'(\rho) ||\boldsymbol{u}_\rho||^2$. □

Some comments on this calculation are as follows:

- Differentiating non-linear maps between L_p spaces as defined by the superposition of polynomials or rational functions, as done in (B.1), is non-trivial in general. However, our case provides enough structure that we can verify the sufficient properties directly (see Appell and Zabrejko 1990, Goldberg, Kampowsky, and Tröltzsch 1992).
- If $\boldsymbol{u}_t$ goes to $\boldsymbol{u}_\rho$ only in L_2, then it is essential that the directions $\delta\rho$ are taken in L_∞. Alternatively, if $\boldsymbol{u}_t$ converges to $\boldsymbol{u}_\rho$ in a space of higher regularity, we could take advantage of the Sobolev embedding theorem and use $\delta\rho$ in a less regular space.
- The convergence analysis for $\boldsymbol{u}_t$ to $\boldsymbol{u}_\rho$ is classical and can be done by appealing to standard energy estimates for linear elliptic equations of second order.

Appendix C

Bugs in The DEM Source Code

While working with the source code provided by [24], we found two bugs. The first one is a simple math error when calculating stress. The bug happens in the `Elastic2DGaussQuad` function of the `InternalEnergy` class in `Sub_Functions/InternalEnergy.py`. There, the stress is calculated as follows:

```
S_xx = self.E*(e_xx + self.nu*e_yy) / (1 - self.nu**2)
S_yy = self.E*(e_yy + self.nu*e_xx) / (1 - self.nu**2)
S_xy = self.E*e_xy / (1 + self.nu)
```

where we have made some small changes to improve readability. This formula comes from the equation for the Cauchy stress tensor:

$$\boldsymbol{\sigma} = \frac{E}{(1+\nu)(1-2\nu)}\mathrm{tr}(\boldsymbol{\varepsilon})\boldsymbol{I} + \frac{E}{1+\nu}\boldsymbol{\varepsilon}.$$

With this formula, you should get

```
S_xx = self.E*((1 - self.nu)*e_xx + self.nu*e_yy) / ((1 + self.nu)(1 - 2 * self.nu))
S_yy = self.E*((1 - self.nu)*e_yy + self.nu*e_xx) / ((1 + self.nu)(1 - 2 * self.nu))
S_xy = self.E*e_xy / (1 + self.nu)
```

The error that has occurred is a simple math error. The calculation in the source code is what you get from the equation

$$\boldsymbol{\sigma} = \frac{E}{(1+\nu)(1-\nu)}\mathrm{tr}(\boldsymbol{\varepsilon})\boldsymbol{I} + \frac{E}{1+\nu}\boldsymbol{\varepsilon},$$

that is, $1-\nu$ was used instead of $1-2\nu$.

The second bug is more subtle. While working on the code that calculated the traction integral $\int_{\delta\Omega} \boldsymbol{t} \cdot \boldsymbol{u}\,\mathrm{d}\boldsymbol{x}$, we noticed something weird. The integral is calculated by the function `lossFextEnergy` of the `IntegrationFext` class in `Sub_Functions/IntegrationFext.py`:

```
def lossFextEnergy(self, u,x, neuBC_coordinates, neuBC_values, neuBC_idx, dxdydz):
    dx=dxdydz[0]
    dy=dxdydz[1]
    dxds=dx/2
    dydt=dy/2

    # J= np.array([[dxds,0],[0,dydt]])
    # Jinv= np.linalg.inv(J)
```

```
    # detJ= np.linalg.det(J)

    traction_ID = 0
    neuPt_u = u[ neuBC_idx[traction_ID].cpu().numpy() ]
    W_ext = torch.einsum( 'ij,ij->i' , neuPt_u , neuBC_values[traction_ID] ) * dx

    W_ext[-1] = W_ext[-1] / 2
    W_ext[0] = W_ext[0] / 2

    FextEnergy = torch.sum( W_ext )
    return FextEnergy
```

This function is a bit messy, but the most important part is `W_ext`. This is equal to $\{\boldsymbol{t} \cdot \boldsymbol{u}_i \Delta x\}$, where i covers all spacial indices that lie in the region with non-zero traction. There is an obvious bug here; the values are always multiplied by Δx, even when integrating vertically. This does not actually change the results for the two examples we looked at as the short cantilever has $\Delta x = \Delta y$, so we do not count it as a bug. The actual bug is on the next few lines, where the traction integral is computed with the trapezoidal rule. What we noticed is that in the short cantilever example, `W_ext` only has one element. This makes the trapezoidal rule invalid, and existing implementations like `torch.trapezoidal` would return 0. The custom implementation does not return 0, it instead just divides the one value by 4 and returns it. This underestimates the applied traction, and thus causes the objective to be much smaller than it should be. Our implementation of the traction integral uses the points at the boundaries of the traction region, which means we always use at least 2 points, and thus avoid this bug.

The reason why `W_ext` only has one value in the case of the short cantilever is actually a result of floating point imprecision. The points that lie in the region of applied traction are found with the condition

```
np.where(
    (dom[:, 0] == Length)
    & (dom[:, 1] >= Height/ 2. - Height / ( Ny - 1 ) / 2.)
    & (dom[:, 1] <= Height/ 2. + Height / ( Ny - 1 ) / 2.)
)
```

This should give two points, $(w, h/2 - \Delta y/2)$ and $(w, h/2 + \Delta y/2)$, but numerically, $h/2 - (h/(N_y - 1))/2 = 2.4444444444444446$, and the bottom point is at $(10, 2.4444444444444442)$. This means that the condition does not hold for the bottom point, leaving just the top one. Fixing this is simple, you just need to subtract a small value from the top condition and add a small value to the bottom condition. A value of 10^{-10} is enough to actually get both points.