Institut für Strukturmechanik und Leichtbau

RWTH AACHEN UNIVERSITY

Masterarbeit

MA2023-13

Multi-Functional Topology Optimization of Lattice Structures using Automatic Differentiation

Diese Arbeit wurde vorgelegt am

Institut für Strukturmechanik und Leichtbau (SLA)
Univ.-Prof. Dr.-Ing. Kai-Uwe Schröder
Fakultät für Maschinenwesen
RWTH Aachen

von

Samuel Hayden

Betreuer:

M.Sc. Stefano Piacquadio
SLA, RWTH Aachen

Aachen, 6. November 2023

Institut für Strukturmechanik und Leichtbau
RWTH AACHEN UNIVERSITY

Masterarbeit

Multi-Functional Topology Optimization of Lattice Structures with Embedded Phase Change Material for use in Spacecraft Structures

Phase change materials (PCMs) are a promising family of materials that have great potential in thermal energy storage and absorption. PCMs on their own, however, generally have poor thermal conductivity which means the PCM only melts around the heat source. This also means there is a high temperature gradient throughout the domain which is undesirable when keeping electronic components cool. By embedding lattice structures with a PCM, the effective thermal conductivity is increased to levels high enough that the structure can be used as a simple thermal control unit that can damp transient effects of a pulsating heat source, or act alone and keep a heat source cool for a fixed duration without the need of an additional heat sink. To maximize the efficiency of a structure, a multi-functional approach to topology optimization can be used to find an ideal structure that fulfills both thermal and structural requirements.

This thesis aims to develop a methodology and basic tools to find an optimized structure that can minimize the temperature on a thermal boundary condition, and maximize the stiffness of the domain. The phase change problem will also be considered to account for the latent heat effects of how the domain melts. Previous works have been conducted that developed semi-analytical models for the material properties. These models will be used as homogenized material properties. It is hypothesized that automatic differentiation (AD) will also be useful in computing the many finite element derivatives that are required in topology optimization.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegene Diplom-/Projekt-/Bachelor-/Masterarbeit selbständig verfasst habe. Ich versichere, dass ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Aachen, den 6. November 2023 _____

(Unterschrift)

Kurzfassung

Die Optimierung der Strukturtopologie ist mittlerweile ein gut entwickeltes Werkzeug, mit dem viele Ingenieur:innen problemlos optimale Strukturen für minimales Gewicht ermitteln können. Die Optimierung der thermischen Topologie wird ebenfalls zu einem gut untersuchten Thema, und auch werden viele Methoden derzeit entwickelt. Durch die Integration dieser Tools in ein einziges Modul kann theoretisch eine Struktur gefunden werden, die beide Probleme optimiert. Normalerweise führen diese zu einem gewissen Kompromiss zwischen den einzelnen Problemen, die berücksichtigt werden müssen. Die Idee der Pareto-Optimalität entsteht, wenn eine Struktur gefunden werden kann, die den Fehler der Struktur in Bezug auf einige nicht erreichbare Entwurfskriterien minimiert.

Natürlich müssen die Materialeigenschaften bekannt sein, um den Bereich zu optimieren. Für viele Gitterelementarzellen wurden analytische und semianalytische Modelle entwickelt. Diese Modelle werden in dieser Arbeit verwendet.

In dieser Arbeit wurde eine Methodik zur multifunktionalen Topologieoptimierung entwickelt sowie eine grundlegende Implementierung in der Programmiersprache "Julia"geschrieben. Die Finite-Elemente-Methode wurde zur Lösung struktureller, thermischer und Phasenänderungsprobleme eingesetzt. Eine optimale Struktur kann anhand eines sogenannten Utopiepunkts ermittelt werden, der in der Regel nicht exakt gefunden werden kann. Es wurden auch Werkzeuge zur Lösung einfunktionaler Topologieoptimierungsprobleme implementiert, die die homogenisierten Eigenschaften ausgewählter Gitterelementarzellen berücksichtigen.

Schlagwörter: Gitterstrukturen, Phasenwechselmaterialien, Multifunktionalität, Topologieoptimierung, Pareto-Optimalität

Abstract

Structural topology optimization is now a well-developed tool that many engineers can easily use to determine optimal structures for minimal weight. Thermal topology optimization is also becoming a well-studied subject with many methodologies now being developed. By integrating these tools into a single module, in theory, a structure can be found that optimizes both problems. Usually, these induce some trade-off between the single problems that need to be considered. The idea of Pareto optimality arises when a structure can be found to minimize the error of the structure with respect to some unobtainable design criteria.

Of course, material properties need to be known to optimize the domain. Analytical and semi-analytical models have been developed for many lattice unit cells. These models will be utilized in this thesis.

In this thesis, a methodology for multi-functional topology optimization has been developed as well as a basic implementation written in the Julia programming language. The finite element method has been utilized for resolving structural, thermal and phase change problems. An optimal structure can be determined given a so-called Utopia point which generally can not be found exactly. Tools for solving single-functional topology optimization problems have also been implemented that consider the homogenized properties of selected lattice unit cells.

Keywords: lattice structures, phase change materials, multi-functional, topology optimization, Pareto optimality

Contents

1 Introduction

The demand for electricity continues to grow as the world develops. To fulfill the demand, the electrical grid must expand. The reduce the effects of climate change, green energies are being sought after more than ever. Green energy, however, is often generated with an offset to the demand [**?**]. Solar energy peaks in the day, whereas electricity demand peaks in the mornings and afternoons. Wind energy is also very inconsistent, having strong peaks and troughs as the day progresses [**?**]. It is often the job of coal or gas power plants to supplement the energy demands and keep the grid stable.

To negate the negative outcomes of green energy, the energy can be stored to dampen the transient effects and ensure power is available when it is required. Chemical batteries can be used for this, but are too expensive to be applied large scale, and the process of mining for the materials required also has negative effects on the climate. Phase change materials (PCMs) are a promising alternative that can be applied large scale to store energy as heat [**?**].

As the material melts, the temperature is kept at a near-constant temperature. This also makes them very good candidates for a thermal management system. In spacecraft structures, for example, many extreme temperature cycles can occur as a satellite orbits the Earth [**?**]. PCMs can help to keep electrical components' temperatures constant to improve their performance.

PCMs on their own, however, suffer many issues. The main problem is that they generally have very low thermal conductivity [**?**]. This means that the PCM only melts in the region near a heat source, which gives rise to a high temperature gradient throughout the domain. This means the latent heat effects of the PCM are not recognized. Work has been conducted [**?**] to improve the thermal conductivity of a PCM, but the most promising option is to use lattice structures.

Lattice structures themself are only a viable option thanks to advances in additive manufacturing technology. Additive manufacturing allows small-scale production of complex parts that were previously either impossible or too difficult to produce. Lattice structures can now be created without the need for complex single-use molds for casting. Additive manufacturing also allows for the production of complex topologically optimized thermal and structural parts.

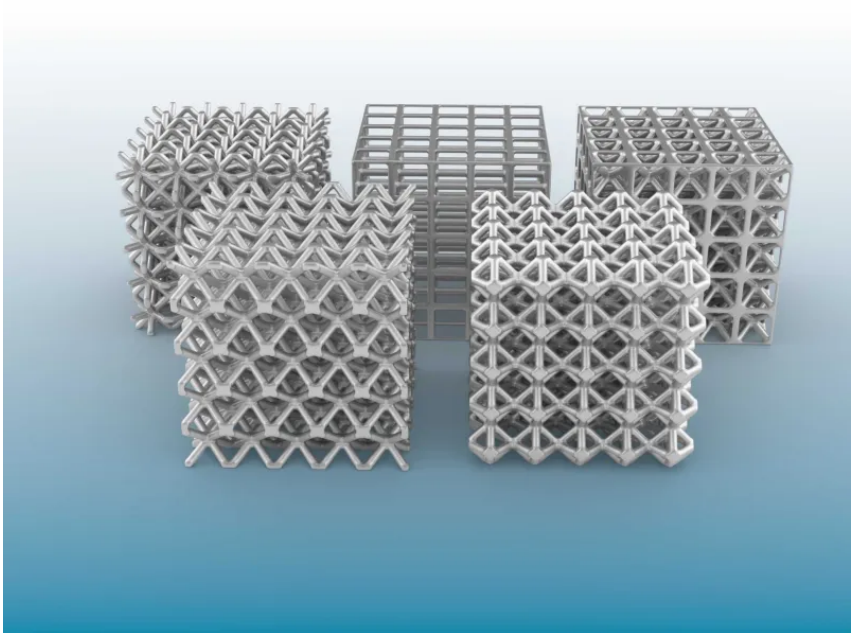


Figure 1.1: Example of lattice structures with various unit cells [**?**]

Schirp-Schoenen, [**?**] showed that the effective thermal conductivity of a PCM can be significantly increased using lattice structures, even when the volume fraction of the lattice is low. More unit cells were later investigated by Soika, et al., [**?**] and semi-analytical formula for the effective material properties were developed. Homogenized models for the effective structural properties of the unit cells have also been by [**?**].

This thesis begins by introducing the fundamental theoretical basis of all the topics involved. Those are the theory of heat transfer, the Stefan problem, structural mechanics and optimization methods. Following that, the state of the art is investigated to ensure a good practical basis of the problem is understood. Topology optimization and multi-functional topology optimization are the most important topics investigated in this thesis, and many methods have been developed for both. This thesis tries to utilize and adapt those methods to the current problem.

With the theory and state-of-the-art understanding, the single-functional topology optimization problems are implemented using the finite element method. These include structural, thermal and phase change topology optimization. The results are briefly discussed before moving on to the multi-functional topology optimization implementation. Finally, some test cases are produced to ensure the solver can find optimums given different boundary conditions.

2 Theory

2.1 Heat Transfer

Heat transfer is the phenomenon of thermal energy being transported as a result of spatial temperature differences [**?**]. In thermodynamics, heat can be transferred by three modes, conduction, convection and radiation. While it is important to describe how energy can be transferred, energy can also be stored in materials. Just by heating materials, the heat pumped in can be stored and transferred back later either by latent heat storage, or sensible heat storage.

Conduction

Conduction is the energy transfer from highly energetic particles to low energetic particles. The energy is transferred by the interactions between the particles. Essentially, when a high-energy particle collides with a low-energy particle, some of that energy is transferred to the lower-energy particle. However, the key difference separating it from convection, is there is no bulk motion [**?**]. In gasses, conduction occurs when particles, which move freely but are spaced far apart, collide. In liquids, the same phenomenon occurs, but the particles are much closer together so collisions are more frequent. And lastly, in solids, it relates to the oscillations of the lattice structure and the movement of free-flowing electrons. Figure 2.1 shows the thermal conductivities of various materials.

Conduction can be described by Fourier's law, shown in equation 2.1, where q represents a given heat flux, k is the thermal conductivity of the material and ∇T is the temperature gradient.

$$q = k\nabla T \tag{2.1}$$

Fourier's law of conduction describes a system that has reached a steady state. To describe a transient system, the heat equation, shown below in equation 2.2, can be used. This equation can describe the temperature evolution over time $\partial_t T$, given the density ρ, specific heat capacity c_p, thermal conductivity k as well as the heat generation q are known [**?**].

$$\rho c_p \partial_t T = \nabla \cdot \nabla(kT) + q \tag{2.2}$$

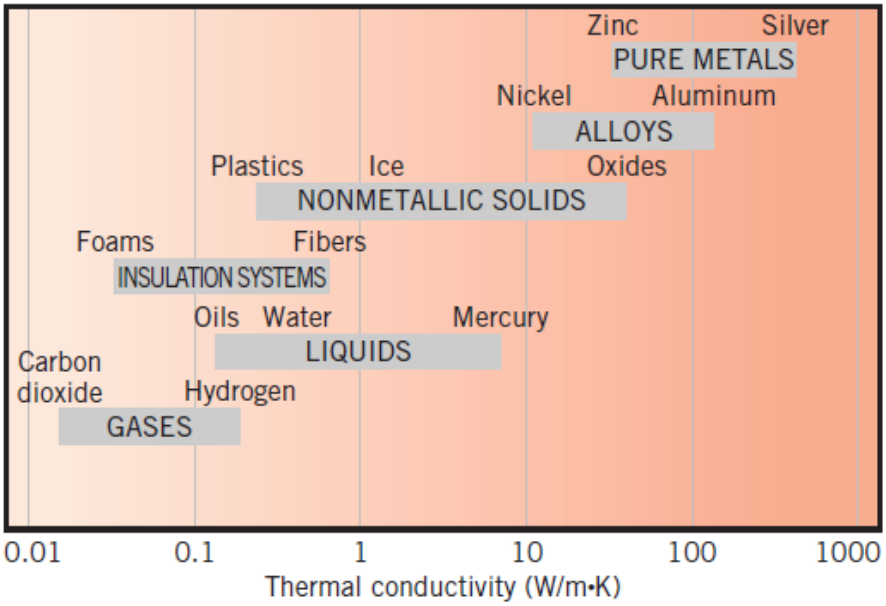


Figure 2.1: Range of thermal conductivities in different materials [**?**]

When the thermal conductivity remains constant or varies very little over the temperature range that is being investigated, the heat equation can be rewritten in the form shown in equation 2.3 where $\alpha = k/\rho c_p$ is a diffusion coefficient [**?**]. The diffusion coefficient relates the ability of a material to conduct heat relative to its ability to store heat [**?**]. For example, materials with a small diffusion coefficient can't conduct the heat away so they need to store it until they reach a new equilibrium.

$$\frac{1}{\alpha}\partial_t T = \nabla^2 T + \frac{q}{k} \tag{2.3}$$

The heat flux in the heat equation can come from multiple sources. For example, heat can be generated internally by chemical reactions or by electrical heating. However, the other heat transfer modes, convection and radiation, can add or subtract from the heat flux.

Convection

Convection is comprised of both conduction and heat transfer resulting from bulk fluid motion [**?**]. Because the thermal conductivity of low, it is the bulk fluid motion that dominates this form of heat transfer. Two forms of convection exist, either forced convection or natural convection. In both cases, equation 2.4 can be used to predict the temperature gradient between an interface, given some convection coefficient h [**?**], which is often determined empirically. Note that q is a heat flux.

$$q = h\Delta T \tag{2.4}$$

In forced convection, the fluid is driven to flow by an external device such as a fan. Many empirical relations exist which can be used to determine the convection coefficient. In the case of forced convection, they usually revolved around determining the Nusselt number Nu_x which is commonly a function of the Reynold's number Re and the Prandtl number Pr. Once the Nusselt number is known, the convection coefficient can be determined from equation 2.5.

$$h = \frac{kNu_x}{x} \tag{2.5}$$

In natural or free convection, fluid motion occurs as a result of slight density changes in the fluid [**?**]. The convection coefficient in this case is usually much lower, as heat is advected much slower. The Nusselt number in free convection usually depends on the Grashof number Gr_x.

$$Gr_x = \frac{g\beta\Delta T x^3}{\nu^2} \tag{2.6}$$

In reality, to capture the fluid motion and temperature distribution throughout a domain, the Navier-Stokes equations are needed. To solve these, numerical methods such as the finite-element method, or finite-volume method are used. This is computationally expensive, which is why a great deal of work has gone into determining convection coefficients for simple problems.

Radiation

Radiation differs from the previous two modes of heat transfer in that it does not require a transfer medium. In radiation, heat is transferred by photons which are emitted from any material of non-zero temperature [**?**]. Energy is emitted according to equation 2.7, where ε is the material emissivity which describes how close to an ideal radiator the material is, and $\sigma = 5.67 \times 10^{-8}$ is the Boltzmann constant [**?**].

$$E = \varepsilon\sigma T_s^4 \tag{2.7}$$

Because all materials emit radiation, some of the energy emitted by the surface will return. Therefore, heat transfer by radiation is described as a net transfer rate, meaning the temperature differences between mediums are still important. This is shown in equation 2.8 where T_s is the temperature of the surface of the material, T_{sur} is the temperature of the surroundings, and q is a heat flux.

$$q = \varepsilon\sigma(T_s^4 - T_{sur}^4) \tag{2.8}$$

2.2 The Stefan Problem

The Stefan problem is a particular kind of heat transfer problem that involves the melting or freezing of a material, giving rise to the movement of an interface within the domain. Josef Stefan developed the theory in the years surrounding 1890 and compared his findings to data that had been obtained during polar expeditions [?][?][?][?]. The classical Stefan problem assumes the fluid domain to be at rest.

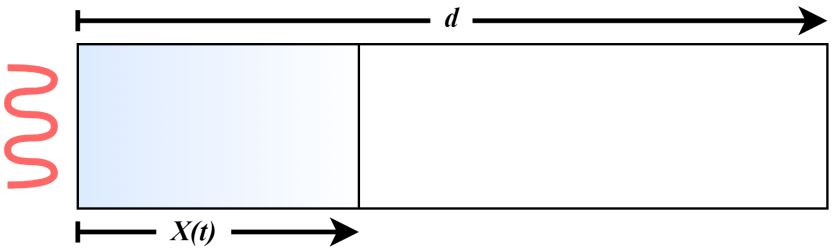


Figure 2.2: Illustration of the Stefan problem

For simplicity, this analysis, based on notes from [?], will consider an initial domain of ice that is all at the melting temperature $T(x,0) = T_m$. This is often referred to as the one-phase Stefan problem as the temperature distribution is only solved in the fluid domain. If a heat source is applied to the left side of the domain, the melt front position $X(t)$ will begin moving through the domain. In the fluid domain, the heat equation can be solved in the domain $0 \leq x \leq X(t)$. Equation 2.9 shows the interface boundary condition where $X^-(t)$ denotes the fluid side of the boundary, and $X^+(t)$ denotes the solid side of the boundary.

$$T(X^-(t),t) = T(X^+(t),t) = T_m \tag{2.9}$$

To close the equation, the position of the interface needs to be determined. One final boundary condition, called the Stefan condition is required. As shown in equation 2.10a it represents a local energy balance at the interface to account for the jump caused by the latent heat L. In the one-phase problem, the temperature gradient in the solid is zero, leaving only the liquid contribution as shown in equation 2.10b.

$$\rho_s L \partial_t X(t) = k_s \partial_x T(X^+(t),t) - k_l \partial_x T(X^-(t),t) \tag{2.10a}$$

$$\rho_s L \partial_t X(t) = -k_l \partial_x T(X^-(t),t) \tag{2.10b}$$

Now that the system is closed, a solution can be found by introducing a similarity variable $\zeta = x/\sqrt{t}$. After solving and back-substitution, the final temperature profile can be determined as shown in equation 2.11, where T_l is the temperature on the left boundary, and the error function $\text{erf}(x) := \frac{2}{\sqrt{x}} \int_0^x e^{-y^2} dy$.

$$T(x,t) = T_l - (T_l - T_m) \frac{\text{erf}(\frac{x}{2\sqrt{\alpha t}})}{\text{erf}(\frac{X(t)}{2\sqrt{\alpha t}})} \tag{2.11}$$

The final thing to determine is the position of the interface. This can be done by making use of the Ansatz function $X(t) = 2\sqrt{\alpha}\lambda\sqrt{t}$. This function introduces the unknown variable λ which can be found by substituting the Ansatz function into the Stefan condition, which results in equation 2.12a.

$$\lambda\sqrt{\pi} = \frac{c_p(T_l - T_m)}{L} \frac{1}{\text{erf}(\lambda)e^{\lambda^2}} = \text{Ste}^{-1} \frac{1}{\text{erf}(\lambda)e^{\lambda^2}} \tag{2.12a}$$

$$g(\lambda) = \text{Ste}^{-1} - \sqrt{\pi}\lambda e^{\lambda^2} \text{erf}(\lambda) = 0 \tag{2.12b}$$

Equation 2.12a cannot be directly solved for λ, so the equation is reformulated to be solved for a root instead as shown in equation 2.12b. This equation can be solved using a root finder. An example of how the front develops is shown in figure 2.3. In this example, the material properties of water were used.

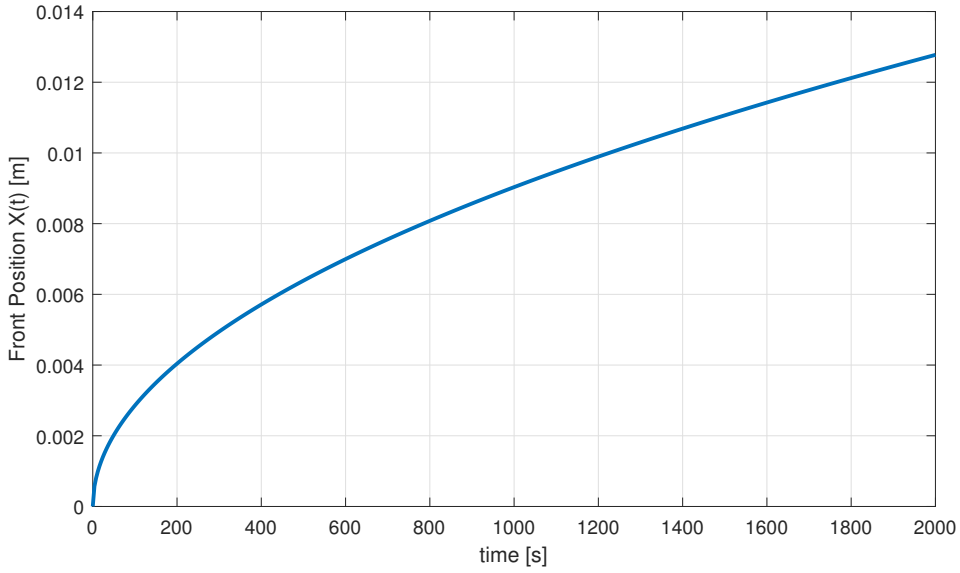


Figure 2.3: Stefan melting front development over time

2.3 Structural Mechanics

The main idea of structural mechanics is that a material deforms under a load. This gives rise to shear and normal stress in the material. It is usually enough to consider materials to behave linearly. This gives rise to the classical Hooke's law, which can be generalized for a multidimensional object. Continuum mechanics also needs to be considered for the derivation, but for simplicity, it is not discussed in this thesis.

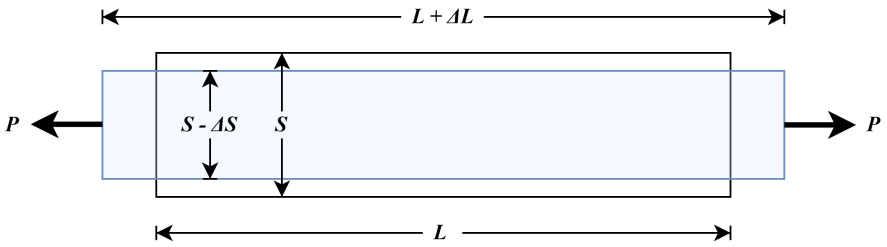


Figure 2.4: One-dimentional Hooke's law illustration

First considering the one-dimensional case as illustrated in figure 2.4. A simple bar of length L with cross-sectional area S is subjected to a load P. Under this load, the bar stretches in the longitudinal direction by ΔL and contracts by ΔS in the perpendicular direction. Classical Hooke's law states that this stretching can be determined, as shown in equation 2.13a [**?**]. The contraction can also be computed from equation 2.13b, where ν is Poisson's ratio.

$$\frac{\Delta L}{L} = \frac{1}{E}\frac{P}{S} \tag{2.13a}$$

$$\frac{\Delta S}{S} = \nu\frac{\Delta L}{L} = \nu\frac{1}{E}\frac{P}{S} \tag{2.13b}$$

Equation 2.13a is most commonly written as $\sigma = E\varepsilon$ where $\sigma = P/S$ and $\varepsilon = \Delta L/L$. In solid mechanics, the strain is usually $O(10^{-3})$ while Young's modulus E is usually $O(10^{6})$. Assuming the rod returns to its original form after the load is released, Hooke's law can be generalized and derived through continuum mechanics, and the linear, elastic material model can be determined. Equation 2.14 shows this.

$$\sigma = \lambda \mathrm{tr}(D)\mathbf{I} + 2\mu D \tag{2.14}$$

The linear elastic model of materials introduces so-called Lamé parameters λ and μ which can be calculated for each material, given its material properties are known, as shown in the equations 2.15b. It also introduces D, the small-strain tensor defined in equation 2.15a.

$$D = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{yx} & \varepsilon_{yy} & \varepsilon_{xz} \\ \varepsilon_{zx} & \varepsilon_{zy} & \varepsilon_{xz} \end{bmatrix} \tag{2.15a}$$

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)} \qquad \mu = \frac{E}{2(1+\nu)} \tag{2.15b}$$

Considering a three-dimensional case, stress occurs in six ways. Three in the normal directions, and three in the shear planes. A compliance matrix can be found for the system $\varepsilon = \mathbf{C}^{-1}\sigma$ can be determined for a general material as shown in equation 2.16, which includes the shear modulus G_{ij} [**?**].

$$\mathbf{C}^{-1} = \begin{bmatrix} \frac{1}{E_x} & \frac{\nu_{xy}}{E_x} & -\frac{\nu_{xz}}{E_x} & 0 & 0 & 0 \\ -\frac{\nu_{yx}}{E_y} & \frac{1}{E_y} & -\frac{\nu_{yz}}{E_y} & 0 & 0 & 0 \\ -\frac{\nu_{zx}}{E_z} & -\frac{\nu_{zy}}{E_z} & -\frac{1}{E_z} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{G_{yz}} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{G_{xz}} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{G_{xy}} \end{bmatrix} \tag{2.16}$$

Often, the materials are isotropic which gives the material uniform material properties in each direction. As in some cases, such as in 2D, the z component of stress and strain are equal to or negligible. This simplifies the matrix as shown in 2.17 for plane stress

$$\mathbf{C}^{-1} = \frac{1}{E}\begin{pmatrix} 1 & -\nu & 0 \\ -\nu & 1 & 0 \\ 0 & 0 & 2(1+\nu) \end{pmatrix} \tag{2.17}$$

or to the equation shown in 2.18 for plane strain [**?**].

$$\mathbf{C} = \frac{E}{1+\nu}\begin{pmatrix} \frac{1-\nu}{1-2\nu} & \frac{\nu}{1-2\nu} & 0 \\ \frac{\nu}{1-2\nu} & \frac{1-\nu}{1-2\nu} & 0 \\ 0 & 0 & \frac{1}{2} \end{pmatrix} \tag{2.18}$$

2.4 Methods for Optimization

In the context of engineering, the goal of optimization is to maximize or minimize one or more properties, possibly with a set of constraints. Typical examples include maximizing the stiffness of a frame while keeping the design below a weight requirement, or minimizing the mass of a bridge while ensuring the maximum stress is below some critical limit. In optimization, a maximum is equivalent to a minimum [**?**]. This allows the same optimization methods to be used for any function.

$$\begin{gathered} \max_x f(x) \\ \text{is equivalent to} \\ \min_x -f(x) \end{gathered} \tag{2.19}$$

In optimization, a constraint limits the design space of possible solutions [**?**]. Figure 2.5 shows a function with a global and local minimum. The χ variable denotes the constraint limiting the design space, and x^* is the optimum that fulfills the design criteria.

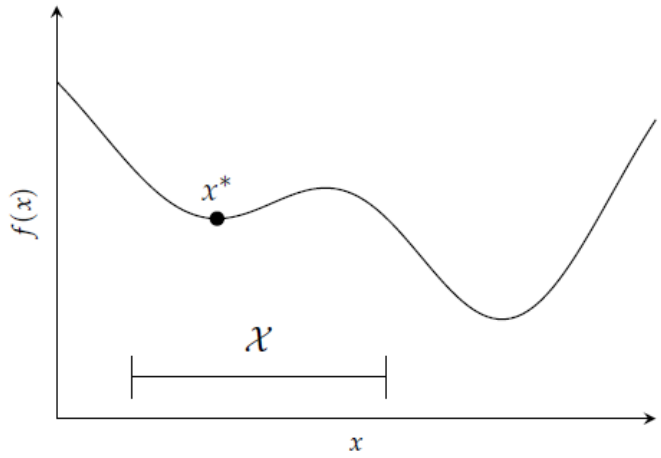


Figure 2.5: Example of a constraint limiting the design space [**?**]

Constrained Optimization

This section will consider the following system.

$$\begin{aligned} \min_x \quad & f(\mathbf{x}) \\ \text{subject to} \quad & h(\mathbf{x}) = \mathbf{0} \\ & g(\mathbf{x}) \leq \mathbf{0} \end{aligned} \tag{2.20}$$

As was briefly introduced, constrained optimization has a function that has had its design space limited by one or more constraints. In such a situation, Lagrange multipliers can be used. The idea behind Lagrange multipliers is to find the point where the gradient of the function and constraint are aligned [**?**]. Equation 2.21a shows the condition for aligned gradients where λ is the Lagrange multiplier. The purpose of the Lagrange multiplier is to scale the gradient, as while they may be aligned, they may not be the same size magnitude. Equation 2.21b is the Lagrangian which is a function of both the design variables and the constraint. By finding $\nabla\mathcal{L}(\mathbf{x},\lambda) = \mathbf{0}$, the gradient condition is met, and a critical point is found [**?**].

$$\nabla f(\mathbf{x}) = \lambda \nabla h(\mathbf{x}) \tag{2.21a}$$

$$\mathcal{L}(\mathbf{x},\lambda) = f(\mathbf{x}) - \lambda h(\mathbf{x}) \tag{2.21b}$$

The above formulation is for equality constraints. This means the critical point is assumed to be on the boundary of the constraint. However, if an inequality constraint is used instead, it may be that the critical point is simply where the gradient zero for the $f(\mathbf{x})$. In this case, the constraint is considered "inactive" [**?**]. Equation 2.22a and 2.22b show a slight change in the Lagrangian, however, the idea is the same. μ is a non-negative Lagrange multiplier to ensure the constraint holds.

$$\nabla f(\mathbf{x}) + \mu \nabla g(\mathbf{x}) = \mathbf{0} \tag{2.22a}$$

$$\mathcal{L}(\mathbf{x},\lambda) = f(\mathbf{x}) + \mu g(\mathbf{x}) \tag{2.22b}$$

To optimize the Lagrangian, a new solution strategy is required as shown in equation 2.23. This formulation is called the primal problem.

$$\min_{x} \max_{\mu \geq 0} \mathcal{L}(\mathbf{x},\mu) \tag{2.23}$$

To ensure a valid solution, the Kanush-Kuhn-Tucker (KKT) conditions [**?**] must be satisfied. These are 2.24a which states the point is feasible, meaning the constraint is satisfied. Equation 2.24b that states the penalty is in the right direction. Equation 2.24c states that either the point is on the boundary $g(\mathbf{x}) = 0$, or $\mu = 0$. Lastly, equation 2.24d states that if the constraint is active, the contours of f and g are aligned, or $\mu = 0$ which recovers $\nabla f(\mathbf{x}^*) = 0$ [**?**].

$$g(\mathbf{x}^*) \leq 0 \tag{2.24a}$$

$$\mu \geq 0 \tag{2.24b}$$

$$\mu g(\mathbf{x}^*) = 0 \tag{2.24c}$$

$$\nabla f(\mathbf{x}^*) + \mu \nabla g(\mathbf{x}^*) = \mathbf{0} \tag{2.24d}$$

Bisection Method

The bisection method is a bracketing method that closes in on a root y. While it is a simple method, it is guaranteed to converge if the starting region brackets the root, and the function is continuous. First, a region $[a,b]$ has to be found which contains at least one root. This can be done by finding a point above the root, and another below. Assuming the function is continuous, then according to the intermediate value theorem, there must be a value between the bracketing points $[a,b]$ that satisfies $f(x) = y$ [**?**]. As is illustrated in figure 2.6, the bisection method works by evaluating a middle point $c = (a+b)/2$ and selecting a new bracket that the root is still inside of. Note however that the function has five roots in the initial bracket, but it only converges to one.

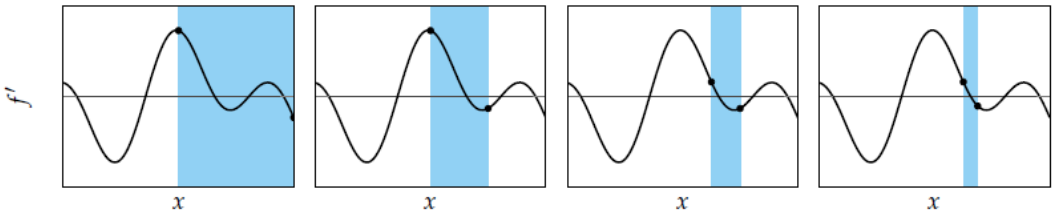


Figure 2.6: Illustration of the bisection method for finding the root of a function [**?**]

Newton-Raphson Method

The Newton-Raphson method, commonly known as the Newton method, is a second-order method that not only finds a decent direction but also predicts how far to step in that direction. Newton's method is an iterative scheme based on a second-order Taylor expansion shown in equation 2.25a and then taking its derivative 2.25b and then solving for x which is used as a new guess $x^{(k+1)}$ as shown in equation 2.25c [**?**]. Note that equation 2.25c divides by the second derivative, giving rise to the condition that the second derivative is non-zero.

$$q(x) = f(x^{(k)}) + (x - x^{(k)}) f'(x^{(k)}) + \frac{(x - x^{(k)})^2}{2} f''(x^{(k)}) \tag{2.25a}$$

$$\frac{\partial q}{\partial x} = f'(x^{(k)}) + (x - x^{(k)}) f''(x^{(k)}) = 0 \tag{2.25b}$$

$$x^{(k+1)} = x^{(k)} - \frac{f'(x^{(k)})}{f''(x^{(k)})} \tag{2.25c}$$

While Newton's method has quadratic convergence, it does have some cases where it will fail to converge, as illustrated in figure 2.7. The first is oscillations, where the new guesses continually overshoot the local minimum. The second is overshoot which occurs when the new guess goes beyond the local minimum. The last occurs when the second derivative is negative, causing the new guess to diverge. Therefore, in Newton's method, a good initial guess is often necessary [**?**].

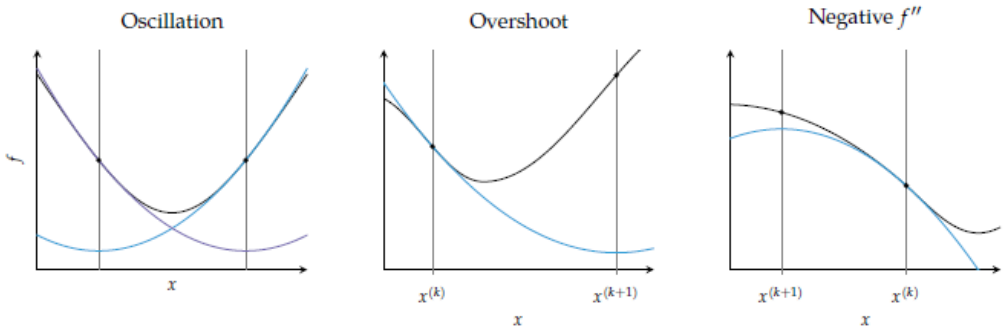


Figure 2.7: Some common failure cases for Newton's method [**?**]

Line Search

The line search method doesn't compute a descent direction, but rather it computes a step size α which decides how far to step in the decent direction, as shown in equation 2.26. The search direction $\mathbf{d}$ can be computed from Newton's method for example.

$$\min_{\alpha} f(\mathbf{x} + \alpha\mathbf{d}) \tag{2.26}$$

The step size can be computed exactly, but its computation is very expensive. Instead, an approximate α can be found quickly using a line search algorithm such as the backtracking algorithm [**?**], and then a new search direction can be computed.

One common line search algorithm is the back-tracking algorithm. This uses a condition called the Armijo condition, shown in equation 2.27, which ensures a sufficient decrease in the objective function [**?**]. Note that $\beta \in [0,1]$.

$$f(\mathbf{x}^{(k+1)}) \leq f(\mathbf{x}^{(}k)) + \beta\alpha\nabla_{\mathbf{d}^{(k)}} f(\mathbf{x}^{(k)}) \tag{2.27}$$

Multi-functional Optimization

Multi-functional optimization, also known as multi-objective optimization, is the optimization of two or more functions that may have conflicting optimal points. This requires the designer to find a trade-off between the objective functions. This gives rise to the idea of Pareto optimality.

A design is said to be Pareto optimal when it is no longer possible to improve one objective without worsening another [**?**]. With a set of Pareto optimal points, a Pareto frontier is constructed. Figure 2.8 shows a Pareto front that also has weakly Pareto-optimal points. Any point taken from the front is a valid Pareto-optimal design, but each will have a trade-off between each function.

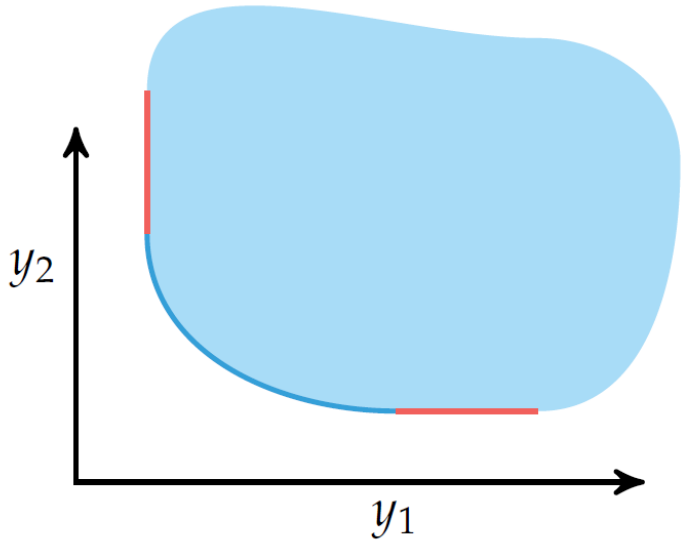


Figure 2.8: A Pareto front (blue) with weakly Pareto-optimal points (red) [**?**]

Many methods exist to obtain Pareto-optimal points, but only two methods will be discussed here. The first is the constraint method. In this method, all but one objective function is constrained [**?**]. This essentially converts the multiple functions into a single constraint function which can be optimized using constraint optimization techniques.

$$\begin{aligned} \min_{x} \quad & f_1(x) \\ \text{subject to} \quad & f_2(x) \leq c_2 \\ & \vdots \\ & f_n(x) \leq c_n \end{aligned} \tag{2.28}$$

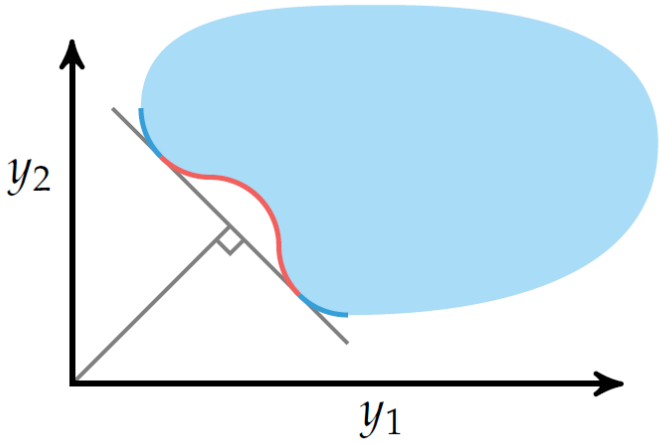


Figure 2.9: Weight method failure on convex regions

The drawback to the constrained method, and the motivation for using the next method, is that the constraints may not be known ahead of time. This could be resolved by minimizing both functions independently and finding the maximum of each function with respect to the other functions' minimums. However, this can be cumbersome. Instead, the weight method can be used. The weight method works but weighing each objective function by an importance factor as shown in equation 2.29. This method also converts the multiple objectives into a single function but doesn't require constraint optimization techniques, or having to determine the constraints. However, one drawback to this method is that this method cannot find Pareto optimal points that are in a convex region, as illustrated in figure 2.9.

$$f(x) = \mathbf{w}^T \mathbf{f}(x) \tag{2.29}$$

2.5 Finite Element Method

The finite element method (FEM) is a numerical method for solving boundary value problems (BVP). As a minor prelude, the Sobolev space $\mathcal{H}^k(\Omega)$ needs to be defined. The Sobolev space is defined as the set of functions that are square integrable in the domain Ω (meaning they are in the $\mathcal{L}_2(\Omega)$ space) and has k square integrable derivatives. The Sobolev space is therefore defined in 2.30a, where $\alpha = \alpha_1, \ldots, \alpha_d$ and $|\alpha| = \alpha_1 + \cdots + \alpha_d$. Also defined is $\mathcal{H}_0^k(\Omega)$ which simply has values that vanish on the boundaries [**?**].

$$\mathcal{H}^k(\Omega) = \left\{ u \in \mathcal{L}(\Omega) \quad | \quad \partial_x^\alpha u \in \mathcal{L}^2(\Omega) \quad \forall |\alpha| \leq k \right\} \tag{2.30a}$$

$$\mathcal{H}_0^k(\Omega) = \left\{ u \in \mathcal{H}^k(\Omega) \quad | \quad u = 0 \text{ on } \Gamma \right\} \tag{2.30b}$$

To define the variational form of a BVP, the notion of trial and weighting function spaces is needed. The weighting functions, denoted as $\mathcal{V}$ and shown in equation 2.31, is the set of functions in $\mathcal{H}^1(\Omega)$ with values that vanish on the Dirichlet boundaries Γ_D [**?**].

$$\mathcal{V} = \left\{ w \in \mathcal{H}^1(\Omega) \;\middle|\; w = 0 \text{ on } \Gamma_D \right\} \tag{2.31}$$

The trial function space, defined in equation 2.32, is the set of functions in $\mathcal{H}^1$ that satisfy the Dirichlet value u_D on the Dirichlet boundaries [**?**].

$$\mathcal{S} = \left\{ u \in \mathcal{H}^1(\Omega) \;\middle|\; u = u_D \text{ on } \Gamma_D \right\} \tag{2.32}$$

These function spaces contain an infinite number of functions, so in FEM, a finite-dimensional subset $\mathcal{V}^h$ and $\mathcal{S}^h$, that can approximate $\mathcal{V}$ and $\mathcal{S}$, is chosen [**?**]. Not only that but the domain is also discretized into element domains. Equation 2.33 shows convenient function spaces which make use of polynomial interpolation functions $\mathcal{P}_m(\Omega)$ of order m which are defined for different types of element e [**?**].

$$\mathcal{V}^h = \left\{ w \in \mathcal{H}^1(\Omega) \;\middle|\; w|_{\Omega^e} \in \mathcal{P}_m(\Omega)^e \;\forall e \text{ and } w = 0 \text{ on } \Gamma_D \right\} \tag{2.33a}$$

$$\mathcal{S}^h = \left\{ u \in \mathcal{H}^1(\Omega) \;\middle|\; u|_{\Omega^e} \in \mathcal{P}_m(\Omega)^e \;\forall e \text{ and } u = u_D \text{ on } \Gamma_D \right\} \tag{2.33b}$$

Dirichlet boundary conditions, which have already been mentioned, impose a value on the boundary $u = u_D$ on Γ_D. Neumann conditions impose a gradient or flux on the boundary $\frac{\partial u}{\partial n} = \mathbf{n} \cdot \nabla u = h$ on Γ_N.

With that, the weak form can now be derived. The weak form is found by multiplying the equation with a weighting function w and integrating it over the domain Ω. For the Poisson equation, equation 2.34 shows the derivation. The second step shows the application of integration by parts and the result after using Gauss' theorem. Note that a Neumann boundary condition naturally arises from this formulation.

$$\begin{aligned} -\int_\Omega w \nabla^2 u \, d\Omega &= \int_\Omega w f \, d\Omega \\ -\int_\Omega \left(\nabla \cdot (w \nabla u) - \nabla w \cdot \nabla u \right) d\Omega &= \int_\Omega w f \, d\Omega \\ \int_\Omega \nabla w \cdot \nabla u \, d\Omega &= \int_\Omega w f \, d\Omega + \int_{\Gamma_N} w h \, d\Gamma \end{aligned} \tag{2.34}$$

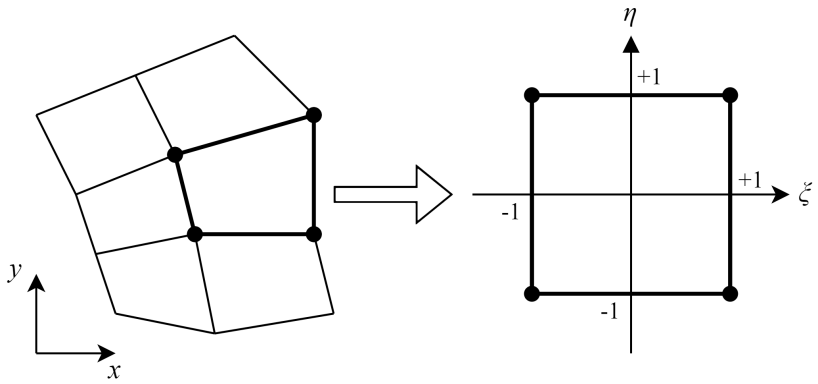


Figure 2.10: 1st order quadrilateral element mapping

Some shorthand notations can be used for simplification.

$$\begin{aligned} a(w,u) &= \int_\Omega \nabla w \cdot \nabla u \, d\Omega \\ (w,f) &= \int_\Omega wf \, d\Omega \\ (w,h)_{\Gamma_N} &= \int_{\Gamma_N} wh \, d\Gamma \end{aligned} \tag{2.35}$$

The Galerkin method can now be applied which discretizes the space into a finite subset of the full function spaces and reads as follows, where $u^h(\mathbf{x}) = \sum_{e=1}^{N_e} N_e(\mathbf{x}) u_e$.

Find $u^h \in \mathcal{S}^h$ such that $a(w^h,u^h) = (w^h,f) + (w^h,h)_{\Gamma_N} \; \forall w^h \in \mathcal{V}^h$

The shape functions N_e are defined for each element. For each element, there are n nodes and an equal number of shape functions. The shape functions are constructed in a way that they are equal to one on their node, and zero on each other node. For example, figure 2.10 shows a 1st-order quadrilateral element, which has the shape functions shown in equation 2.36.

$$\begin{aligned} N_1 &= \frac{1}{4}(1-\xi)(1-\eta) \\ N_1 &= \frac{1}{4}(1+\xi)(1-\eta) \\ N_1 &= \frac{1}{4}(1+\xi)(1+\eta) \\ N_1 &= \frac{1}{4}(1-\xi)(1+\eta) \end{aligned} \tag{2.36}$$

By defining the elements in a separate domain, these need to be mapped to the global domain. This leads to an isoparametric mapping for the quadrilateral element [**?**], as shown in equation 2.37.

$$\begin{Bmatrix} x \\ y \end{Bmatrix} = \sum_{i=1}^{4} N_i(\xi,\eta) \begin{Bmatrix} x_i \\ y_i \end{Bmatrix}, \quad \text{and} \quad u^h(x,y) = \sum_{i=1}^{4} N_i(\xi,\eta) u_i \tag{2.37}$$

By defining the reference elements, it is also possible to efficiently compute their integrals using Gaussian quadrature. Equation 2.38 shows the rule where w_i are weights for each Gauss point $[\xi_i, \eta_i]$ [**?**], given n_p Gauss points.

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n_p} w_i f(\xi_i, \eta_i) \tag{2.38}$$

All of the above is related to spatial discretization. For transient problems, however, time discretization is needed. The theta-family methods, shown in equation 2.39, are some of the most common first-order time discretization techniques [**?**]. Some common methods coming from this are the backward ($\theta = 1$) and forward ($\theta = 0$) Euler methods, as well as the Crank-Nicolson ($\theta = 1/2$) method.

$$\frac{\partial u}{\partial t} \approx \frac{u(t^{n+1}) - u(t^n)}{\Delta t} = \theta u_t(t^{n+1}) + (1-\theta) u_t(t^n) \tag{2.39}$$

Equation 2.39 can also be expressed in the incremental form, shown in equation 2.40, in which the incremental unknown $\Delta u = u^{n+1} - u^n$ is solved for. This form is often preferred [**?**]. Note that $u_t = \frac{\partial u}{\partial t}$

$$\frac{\Delta u}{\Delta t} - \theta \Delta u_t = u_t^n \tag{2.40}$$

The full discretization of the time-dependent poisson equation is given in equation 2.41. The discretization makes use of the backward Euler method ($\theta = 1$). Note that after spatial discretization, the derivation makes use of the bilinear forms defined in equation 2.35.

$$\begin{aligned} \frac{u^{n+1} - u^n}{\Delta t} &= f^{n+1} + \nabla^2 u^{n+1} \\ u^{n+1} - \Delta t \nabla^2 u^{n+1} &= \Delta t f^{n+1} + u^n \\ (w, u^{n+1}) - \Delta t \; a(w, u^{n+1}) &= \Delta t (w, f) + \Delta t (w, h)_{\Gamma_N} + (w, u^n) \end{aligned} \tag{2.41}$$

2.6 Automatic Differentiation

Automatic differentiation (AD) is a powerful numerical tool that is capable of computing exact derivatives (to machine precision) of computer programs. This is possible based on the idea that computer programs are made up of elementary operations such as multiplication, division, addition and subtraction [**?**]. The key to AD is the application of the chain rule [**?**], shown in equation 2.42.

$$\frac{d}{dx}f(g(x)) = \frac{df}{dg}\frac{dg}{dx} \tag{2.42}$$

Automatic differentiation can be implemented through operator overloading and the use of dual numbers [**?**]. Dual numbers are similar to complex numbers, in that they are expressed as $a+b\varepsilon$. The variable ε^2 is defined to be 0 and the values a and b are both real numbers. The operations performed on dual numbers act as normal, as shown below in equation 2.43 [**?**].

$$(a+b\varepsilon)+(c+d\varepsilon) = (a+c)+(b+d)\varepsilon \tag{2.43a}$$

$$(a+b\varepsilon)\times(c+d\varepsilon) = (ac)+(ad+bc)\varepsilon \tag{2.43b}$$

Dual numbers are powerful tools when passed into any smooth function f. By Taylor expansion, it can be shown that the function evaluation, as well as its derivative, can be found [**?**].

$$\begin{aligned} f(x) &= \sum_{k=0}^{\infty}\frac{f^{(k)}(a)}{k!}(x-a)^k \\ f(a+b\varepsilon) &= \sum_{k=0}^{\infty}\frac{f^{(k)}(a)}{k!}(a+b\varepsilon-a)^k \\ &= \sum_{k=0}^{\infty}\frac{f^{(k)}(a)b^k\varepsilon^k}{k!} \\ &= f(a)+bf'(a)\varepsilon+\varepsilon^2\sum_{k=2}^{\infty}\frac{f^{(k)}(a)b^k}{k!}\varepsilon^{(k-2)} \\ &= f(a)+bf'(a)\varepsilon \end{aligned} \tag{2.44}$$

Furthermore, AD can be used in either forward or reverse mode. Consider a generic problem where the Jacobian $\frac{df_i}{dx_j}$ is to be computed, where $i \in [1,2,\ldots,n]$ and $j \in [1,2,\ldots,m]$. Forward mode AD is capable of computing the Jacobian column-by-column $\frac{d\mathbf{f}}{dx_j}$. In reverse mode, the Jacobian is computed row-by-row $\frac{df_i}{d\mathbf{x}}$. How this is achieved will be discussed in the following sections. For the following examples, the derivative of the function in equation 2.45 will be evaluated.

$$f(x_1,x_2) = x_1x_2 + x_1\sin(x_2) \tag{2.45}$$

Forward Mode

As was mentioned, forward mode AD computes a Jacobian column-by-column. This means it is more efficient to use forward mode when $n < m$. The evaluation of equation 2.45 can be thought of visually as a directed acyclic graph (DAG), where each sub-evaluation is a vertex. The lines connecting each vertex can be thought of as where the derivatives are evaluated. The DAG for equation 2.45 is shown in figure 2.11.

With forward mode AD, a function can be differentiated with respect to a single variable in each forward pass. A forward pass is simply computing the DAG from left to right. To achieve this using dual numbers, the dual number for only one variable is '*seeded*.' In this example $f(a,b)$, equation 2.46. In this example, v_4 and v_5 using equation 2.43b and v_6 used equation 2.43a. To compute the full derivative, two forward passes are required.

$$\begin{aligned} v_1 &= a + 1\varepsilon & v_4 &= v_1 v_2 = ab + b\varepsilon \\ v_2 &= b + 0\varepsilon & v_5 &= v_1 v_3 = a\sin(b) + \sin(b)\varepsilon \\ v_3 &= \sin(b) + 0\varepsilon & v_6 &= v_4 + v_5 = ab + a\sin(b) + (b + \sin(b))\varepsilon \end{aligned} \tag{2.46}$$

Reverse Mode

Reverse mode AD computes a Jacobian row-by-row. This makes it more efficient when $m < n$. Reverse mode AD consists of a forward pass where the vertices and lines are computed, and a reverse accumulation pass. In this case, the lines are the derivatives of v_j w.r.t v_i rather than w.r.t the variable.

The reverse pass requires the evaluation of equation 2.47. The parents (p) of vertex i are all the forward vertices connected to v_i. For this example, v_2 has v_3 and v_4 as parents. From equation 2.47, it is clear that the entire DAG needs to be stored in memory to be able to compute the derivative. However, it is now also possible to compute the derivative w.r.t all the inputs.

$$\frac{\partial y}{\partial v_i} = \sum_{p \in \text{parent}(i)} \frac{\partial y}{\partial v_p} \frac{\partial v_p}{\partial v_i} \tag{2.47}$$

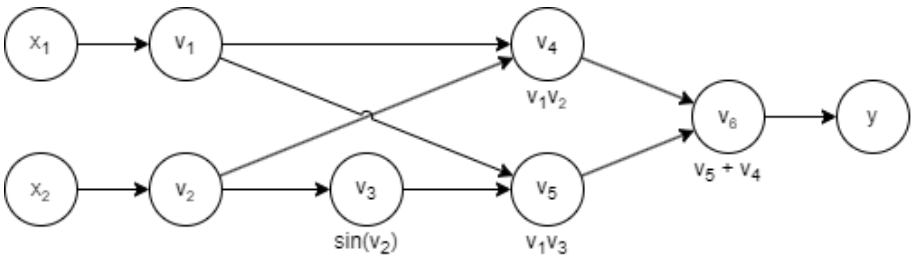


Figure 2.11: Graph of equation 2.45

3 State of the Art

3.1 Cellular Structures

Cellular structures are a relatively new kind of structure with promising properties that can be utilized in engineering, as they combine high stiffness with low mass. They are common in nature, such as in honeycombs, bone and wood. While they have been manufactured in the past using conventional approaches, such as using complex molds for casting [**?**], their use in recent times has been accelerated due to the rise of additive manufacturing [**?**]. Cellular structures can be broken into two categories. These are stochastic structures (foams) that have random non-repeating structures, and lattice structures which have repeating unit cells [**?**]. Lattice structures have been found to have superior properties when compared to foams, making them a promising solution to applications such as in lightweight structure design [**?**].

There is a multitude of different types of cells that can be used. Some common unit cells are shown in figure 3.1. These cells are open cells with a height h, that are built up by struts with radius r.

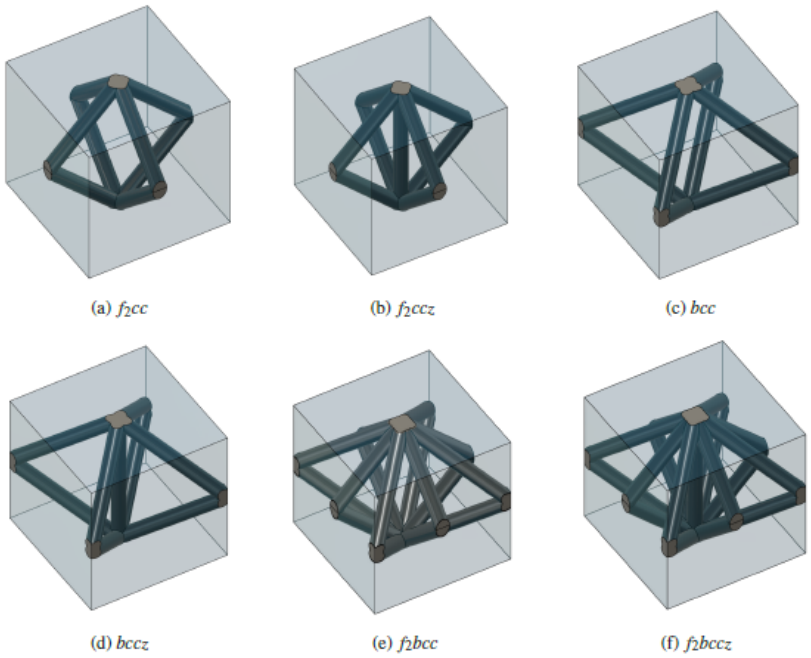


Figure 3.1: A selection of unit cells [**?**]

To simulate cellular structures, without having to mesh each cell, they can be treated as materials. For this, researchers have worked on homogenizing each type of cell to give effective material properties [**?**][**?**]. In structural mechanics, this involves determining the elastic modulus in each direction, whereas, in thermal problems, it involves determining the conductivity in each direction with an embedded PCM. The cells are considered porous media with porosity ε. The porosity can be defined as the proportion of void volume to the total volume, or one minus the solid volume fraction $\mathcal{X}$ [**?**], as shown in equation 3.1.

$$\varepsilon = 1 - \mathcal{X} = 1 - \frac{V_{\text{solid}}}{V_{\text{total}}} \tag{3.1}$$

Structural Properties

For the structural properties, only the matrix (the cell itself) is considered. This is because the selected phase change material, *paraffin*, has a very low elastic modulus, relative to the material of the cell. Not only that but when the phase change material is in its liquid phase, then it provides no stiffness to the cells. For this thesis, the *bcc* and f_2cc,z cells have been considered, but any other homogenized unit cell could be used instead. The properties were obtained from [**?**].

For the selected cells, only the Young's moduli in each direction are required. Young's modulus is non-linear for the unit cells. Figure 3.2 shows the non-linear nature of Young's modulus. This will later have an impact on the topology optimization because the intermediate densities will have a different impact on how the structure grows.

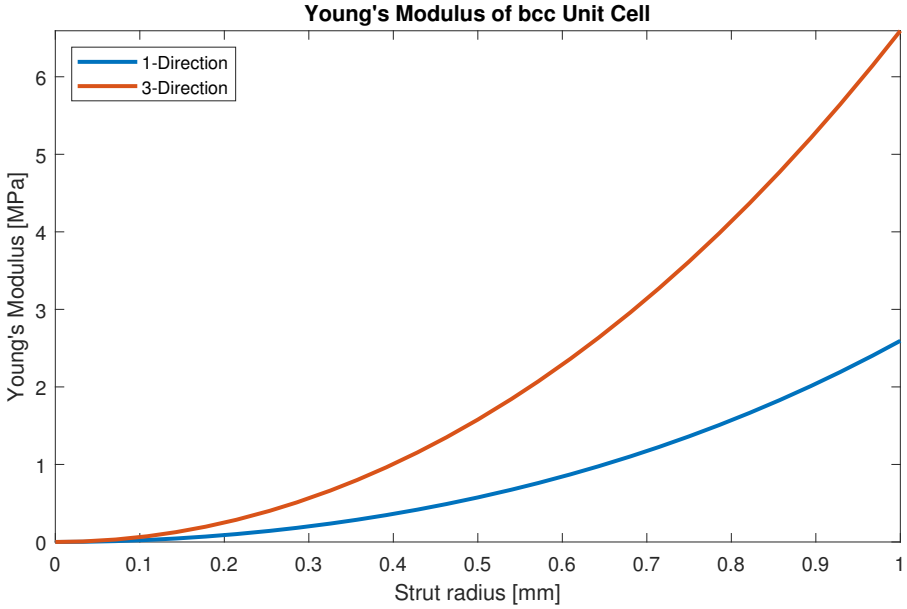


Figure 3.2: Young's Modulus of the BCC unit cell in each direction

Thermal Properties

The thermal properties of the unit cells are made up of both a highly conductive matrix (the cells themself), and a low-conductivity filler (phase change material). The cell properties are obtained from the master's thesis [**?**]. Again, this thesis only focuses on the *bcc* and f_2cc,z cells, but any other homogenized unit cell could be used. The filler material is considered in these equations as it provides a non-negligible contribution to the thermal properties.

The required thermal properties that are needed are the effective thermal conductivity k_{eff}, the effective specific heat $c_{p,eff}$, and the effective latent heat L_{eff}. The specific heat and latent heat properties can be computed using the equations shown in equation 3.2 which include the effective density ρ_{eff} [**?**]. Note that these equations all consider the porosity ε.

$$\rho_{eff} = \varepsilon\rho_{\text{PCM}} + (1-\varepsilon)\rho_{\text{solid}} \tag{3.2a}$$

$$c_{p,eff} = \varepsilon\frac{\rho_{\text{PCM}}}{\rho_{eff}}c_{p,\text{PCM}} + (1-\varepsilon)\frac{\rho_{\text{solid}}}{\rho_{eff}}c_{p,\text{solid}} \tag{3.2b}$$

$$L_{eff} = \varepsilon\frac{\rho_{\text{PCM}}}{\rho_{eff}}L_{\text{PCM}} \tag{3.2c}$$

Figure 3.3 shows the properties of the *bcc* unit cell as an example. Each of the properties is non-linear. Note that the effective latent heat is in terms of volume fraction, as this is important during the topology optimization. The specific heat curve is essentially a scaled version of the latent heat curve, so is not shown in the plots.

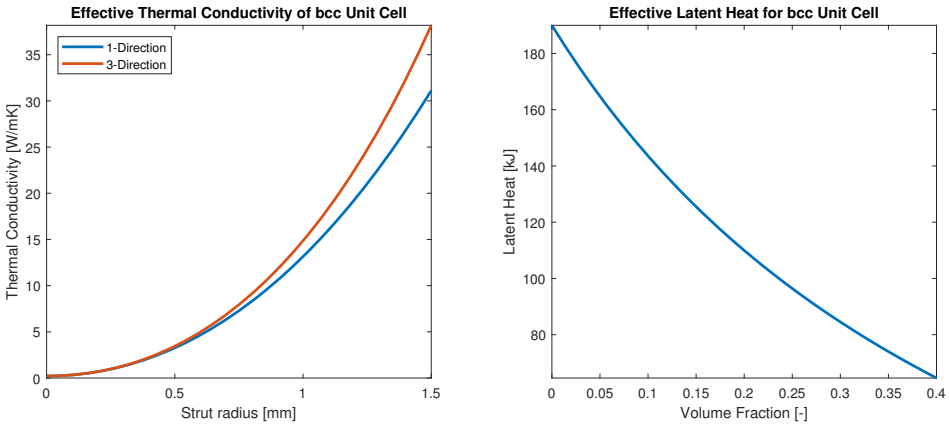


Figure 3.3: Thermal properties of the *bcc* unit cell

3.2 Latent Heat Energy Storage

Thermal energy storage is a method of storing heat in materials, either actively or passively, that can then be dissipated at a later time [**?**]. One of the main uses currently is to assist the temporal and location mismatches between energy generation, and energy use. Thermal energy storage can also be used as a thermal management solution, where large amounts of heat may be generated irregularly, and can be dissipated passively over time.

A common thermal energy storage method is sensible heat energy storage. This method stores energy in a material without any phase change. The amount of sensible energy stored, Q, is expressed in equation 3.3 [**?**]. Because this method involves no phase change, the best materials to select are those with a high specific heat c_p and low cost. Materials such as water, oil and molten salts have been commonly used due to their availability and low cost. Note that the variable m denotes the mass. Note that here $\Delta T = T - T_{\text{ref}}$, where T_{ref} is the initial or ambient temperature.

$$Q = mc_p\Delta T \tag{3.3}$$

In contrast to sensible heat energy storage, latent heat energy storage relies on the phase transition enthalpy Δh, or latent heat $L = \Delta h$, of a material to store vast amounts of energy for virtually no change in temperature [**?**]. The total absorbed energy of the material can be expressed as shown in equation 3.4, which is the sum of sensible energy storage resulting in the solid and liquid phase, and the latent heat of melting. Latent heat energy storage materials are referred to as phase change materials. Note that the subscripts denote the solid phase s or liquid phase l. Also note that $\Delta T_s = T_{\text{melt}} - T_{\text{ref}}$ and $\Delta T_f = T - T_{\text{melt}}$.

$$Q = mc_{p,s}\Delta T_s + mL + mc_{p,l}\Delta T_l \tag{3.4}$$

Comparing the two methods, as shown in figure 3.4, the choice of energy storage method should depend on the application. When the temperature range is large, sensible energy storage can store much more energy, while in a narrow temperature range, latent heat energy storage is far superior.

Phase Change Materials

The selection of phase change materials can be quite complex. The most important criteria are shown in table 3.1 [**?**]. The most obvious selection criterion is the melting temperature of the material. This needs to be within the operating temperature, otherwise, it will simply act as a sensible heat storage medium. The next major selection criterion is the latent heat of the material. The higher, the more energy can be stored, and the longer the material remains around the melting temperature. Another very important property is the thermal conductivity. Having

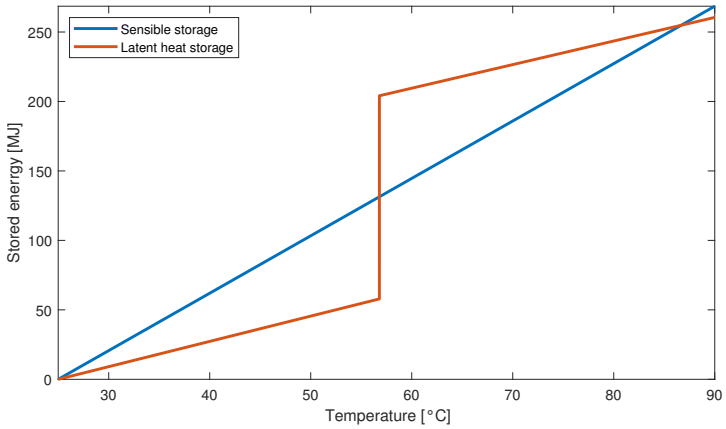


Figure 3.4: Sensible energy storage in water vs. the latent heat energy storage in paraffin

a low thermal conductivity would mean the temperature gradient in the domain could be quite high. This is not desirable when used in a cooling situation.

The physical properties are also quite important to consider. If the phase change material is in a fixed volume, then the density change of the material could lead to very high pressures being sustained. It is for this reason that phase change materials are usually selected for solid-to-liquid phase change.

Table 3.1: Phase change material selection [**?**]

Thermal	Melting temperature
	Latent heat
	Thermal conductivity
	Specific heat
Physical	High density
	Volume change
	Low vapour pressure
Chemical	Chemical stability
	Non-corrosiveness
	Non-flammability
Economical	Availability
	Cost
	Commercial viability

Some common phase change materials are shown in figure 3.5. A particularly good phase change material is water/ice, due to it having a high melting enthalpy and high availability. However, at ambient temperatures, the ice would have already melted, and therefore, only act as a sensible heat energy storage medium. Instead, paraffins have become a promising phase change material. Paraffins are organic materials that are a mixture of straight-chain n-alkanes [**?**]. By increasing the chain length, the melting temperature, as well as the latent heat increases [**?**]. This allows the material to be fine-tuned, to a degree, to the specific low-temperature use case. Of course, the other selection criteria also need to be considered.

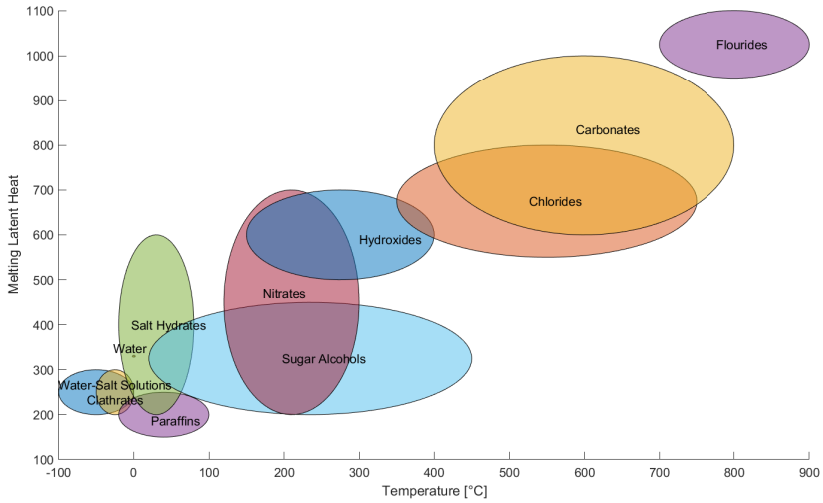
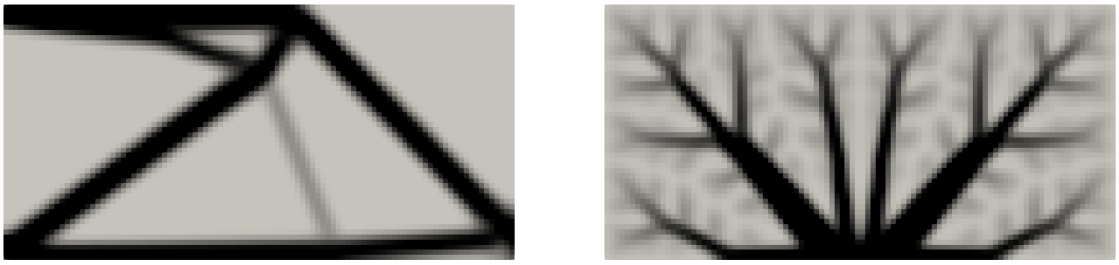


Figure 3.5: Phase change material selection chart [**?**]

3.3 Topology Optimization

Topology optimization is a shape optimization technique that is used to automatically design an ideal structure based on a set of boundary conditions. It works by distributing material throughout a domain only into the regions where it is required. In structural mechanics, this results in truss-like structures whereas in thermal problems, it results in organic tree-like structures.

To perform a topology optimization, the boundary conditions must be specified. The final volume V of the structure should also be defined which is used to define how much material can be distributed. Lastly, some solid or void regions can also be defined which enforces full or empty regions in the domain. These may be desired for screw locations for example. Other than these quantities, the shape and connectivity of the structure are unknown [**?**].



(a) Solid isotropic material optimization (b) Thermal isotropic material optimization

Figure 3.6: Examples of optimized domains for structural and thermal problems

Compliance Minimization using SIMP

To optimize the domain, a function needs to be minimized (or maximized). The most common approach is to iteratively minimize the compliance, c, of the design, which in the context of a structural problem, maximizes the stiffness. Using the finite element method, the discrete form of this problem is given in equation 3.5 [**?**] where $\mathbf{f}$ is the forcing array or right-hand side coming from the discretized system of equations, $\mathbf{u}$ is the vector of unknowns, $\mathbf{K}$ is the stiffness matrix coming from the system of equations and D_e is the element material tensor that could be a conductivity or strain matrix. The compliance is essentially a measure of the objective because it relates the displacements or temperature at a boundary condition to the force or flux on that boundary. By minimizing compliance, the displacement or temperature is minimized on that boundary.

$$\begin{aligned} \min \quad & c = \mathbf{f}^T \mathbf{u} \\ \text{s.t:} \quad & \mathbf{K}(D_e)\mathbf{u} = \mathbf{f} \end{aligned} \tag{3.5}$$

One problem with this formulation is that it is non-differentiable due to the binary nature of materials. This restriction can be relaxed by introducing an interpolation function that smooths this transition. The most popular of which is the *'Solid isotropic material with penalization (SIMP)'* [**?**], shown in equation 3.6. In this equation, D_0 is the real material tensor, and D_e is the penalized material tensor.

$$D_e(x) = \rho(x)^p D_0, \quad p > 1 \tag{3.6}$$

Equation 3.6 introduces the element relative density $\rho(x) \in [0,1]$ which interpolates between the material being present ($\rho = 1$), or void ($\rho = 0$). The equation also introduces the penalization parameter p which is used to penalize intermediate densities. By doing so, it ensures that the intermediate densities are undesirable, and forces them to become either solid or void [**?**]. Usually, $p \geq 3$ to ensure the intermediate densities are penalized enough. With that, equation 3.5 can be slightly reformulated as given in equation 3.7. The second constraint is the volume

constraint which ensures the final volume is less than or equal to the user-defined maximum volume.

$$\begin{aligned} \min \quad & c(\rho(\mathbf{x})) = \mathbf{f}^T \mathbf{u} \\ s.t \quad & \mathbf{K}(D_e(\mathbf{x}))\mathbf{u} = \mathbf{f} \\ & \int_\Omega \rho(x)\, d\Omega \leq V \end{aligned} \tag{3.7}$$

Complications

While this formulation can obtain a structure, many complications must be dealt with. The main complication is the so-called *'checkerboard'* problem [**?**]. These are a result of poor numerical modeling that overestimates the stiffness, similar to problems that occur in the pressure field in Stokes flow problems using FEM [**?**]. As a result of the checkerboard pattern, the optimized structure is non-physical. To resolve this problem, filtering can be used. Many filters have been proposed, such as those investigated by Sigmund [**?**]. Of those proposed, the so-called density or sensitivity filters are the most popular for their ease of implementation and efficiency [**?**]. These act like low-pass filters which essentially blurs the sensitivities in a local region. Equation 3.8b shows the form of a sensitivity filter [**?**] and equation 3.8c shows the form of a density filter [**?**]. Each uses a filter matrix, such as the one shown in equation 3.8a with filter radius r. Using the suggested filter matrices also has the effect of making the results mesh-independent.

$$H_{ij} = \max(r - \|x_i - x_j\|, 0) \tag{3.8a}$$

$$\widetilde{\frac{\partial c_i}{\partial \rho_e}} = \frac{1}{\rho_i \sum_{j=1}^{N} H_{ij}} \sum_{j=1}^{N} H_{ij} \rho_j \frac{\partial c_i}{\partial \rho_j} \tag{3.8b}$$

$$\widetilde{\rho}_i = \frac{\sum_{j=1}^{N} H_{ij} v_j \rho_j}{\sum_{j=1}^{N} H_{ij} v_j} \tag{3.8c}$$

Figure 3.7 shows an optimized structure with no filtering. It shows checkerboarding in the top middle area, as well as struts with elements that only join at one node. Such a structure could not be manufactured due to these voids.

Another issue with topology optimization is that many local minima can be obtained for the same given set of boundary conditions [**?**]. These can be found by changing the initial structure, or by using different penalization parameters. Some methods have been used to overcome this problem, but one simple approach is to gradually increase the penalization parameter from 1 to the desired value. Another method is to start with a large filter radius, and gradually decrease it [**?**].

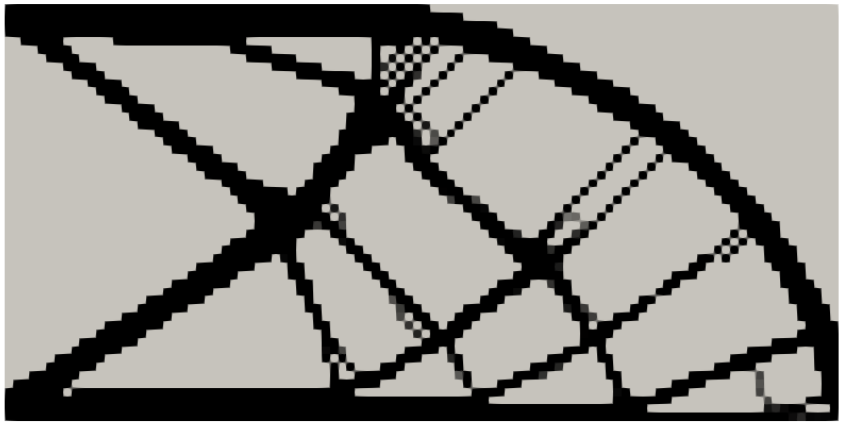


Figure 3.7: Checkerboard pattern and non-physical regions resulting from no filter

Update Criterion

The final ingredient required to perform a topology optimization is to incrementally update the element's relative densities until a converged structure is obtained. The method of moving asymptotes has been used for updating the design variables [**?**] as well as the so-called *'optimality criteria method'* [**?**]. For this thesis, the optimality criteria method as well as the generalized optimality criteria method, proposed by Kim, et al., 2021 [**?**] has been adopted.

Recall that constrained optimization has been discussed in chapter 2.4 where the idea of Lagrange multipliers was introduced. A Lagrangian can be constructed for a topology optimization problem as shown in equation 3.9 [**?**]. Note that f is the volume fraction and V_0 is the total volume of the domain. This is equal to the user-defined final volume. For simplicity, $\mathbf{x}$ will be used in place of $\rho(\mathbf{x})$.

$$\begin{aligned} \mathcal{L}(\mathbf{x},\lambda) &= c(\mathbf{x}) + \lambda(V(\mathbf{x}) - fV_0) \\ \text{where: } V(\mathbf{x}) &= \int_\Omega \rho(\mathbf{x})\, d\Omega \end{aligned} \tag{3.9}$$

A critical point is found when the following conditions in equation 3.10 are met.

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{x}} &= \frac{\partial c(\mathbf{x})}{\partial \mathbf{x}} + \lambda \frac{\partial V(\mathbf{x})}{\partial \mathbf{x}} = 0 \\ \frac{\partial \mathcal{L}}{\partial \lambda} &= V(\mathbf{x}) - fV_0 = 0 \end{aligned} \tag{3.10}$$

To determine the Lagrange multiplier λ, the first constraint can be rearranged as shown in equation 3.11 for each element. This variable is called the scale factor, and an optimum is reached when it is equal to one. The optimal Lagrange multiplier can be found using bisection or similar methods.

$$B_e = -\frac{\frac{\partial c(\mathbf{x})}{\partial x_e}}{\lambda \frac{\partial V(\mathbf{x})}{\partial x_e}} \tag{3.11}$$

With that, each element can have its relative density updated according to equation 3.12. This approach limits how much each element can be updated by controlling the move m parameter. The value η, often set to 0.5, is used to control how quickly and stable the solution can converge [**?**].

$$\rho_{k+1} = \begin{cases} \max\left((1-m)\rho_k,\ \rho_{min}\right) & \text{if } \rho_k B_k^{\eta} \leq \max\left((1-m)\rho_k,\ \rho_{min}\right) \\ \min\left((1+m)\rho_k,\ \rho_{max}\right) & \text{if } \rho_k B_k^{\eta} \geq \max\left((1+m)\rho_k,\ \rho_{max}\right) \\ \rho_k B_k^{\eta} & \text{otherwise} \end{cases} \tag{3.12}$$

The generalized optimality criteria method (GOCM) extends the standard version by allowing multiple inequality constraints with improved computational efficiency [**?**]. A more general version of the Lagrangian is given below in equation 3.13, which also includes a slack variable. This variable is non-zero when the constraint is inactive.

$$\mathcal{L}(\mathbf{x}, \lambda, \mathbf{s}) = c(\mathbf{x}) + \sum_{i=1}^{N} \lambda_i (g_i(\mathbf{x}) + s_i^2) \tag{3.13}$$

A critical point is found when the gradient of the Lagrangian is zero. This results in equation 3.14 [**?**]. Because of the third condition, only the active constraints need to be considered [**?**].

$$\begin{aligned} &\nabla_{\mathbf{x}} c(\mathbf{x}) + \sum_{i=1}^{N} \lambda_i \nabla_{\mathbf{x}} g_i = 0 \\ &g_i(\mathbf{x}) + s_i^2 = 0,\ i \in [1, N] \\ &\nabla_i s_i = 0 \end{aligned} \tag{3.14}$$

One of the ideas proposed by Kim, et al., the Lagrange multipliers don't need to be satisfied for every iteration. As a result, very little computational resources are needed for each iteration to find the Lagrange multipliers. Instead, they are updated each iteration until convergence [**?**]. Patnaik et al., [**?**] proposed a few options for this. Two of which are given in equation 3.15.

The third option is proposed by Kim et al., [**?**] which aims to control how quickly the Lagrange multiplier is found. The relative densities can then be updated according to equation 3.12

$$\begin{aligned} \lambda_i^{k+1} &= \lambda_i^k(1+\alpha^k p_0 g_i) && \text{Linear form} \\ \lambda_i^{k+1} &= \lambda_i^k(g_i)^{\alpha^k p_0} && \text{Exponential form} \\ \lambda_i^{k+1} &= \lambda_i^k(1+p_0(g_i^k+\Delta g_i^k)) && \text{Kim et al., (2023)} \end{aligned} \tag{3.15}$$

Structural Optimization

Structural topology optimization is a very useful tool in engineering. It is capable of giving a designer a rough idea of how to design a structure by maximizing its stiffness, or by minimizing its stress. The goal function, as shown in equation 3.16 is to minimize the compliance, which has the effect of maximizing stiffness.

$$\begin{aligned} \min \quad & c(\rho) = \mathbf{u}^T\mathbf{f} \\ s.t \quad & \mathbf{K}(D_e(\mathbf{x}))\mathbf{u} = \mathbf{f} \\ & \int_\Omega \rho(x)\, d\Omega \leq V \end{aligned} \tag{3.16}$$

The gradient of the compliance, as given by Bendsøe, et al., 2004 [**?**] is shown, without derivation, in equation 3.17. Note that this only considers the theory of small displacements. Non-linear problems are more complex but are not required for this thesis.

$$\frac{\partial c}{\partial \rho_e} = -\mathbf{u}^T \frac{\partial \mathbf{K_e}}{\partial \rho_e}\mathbf{u} \tag{3.17}$$

Thermal Optimization

Thermal topology optimization has become a well-researched topic with many methods now existing [**?**] [**?**]. Thermal compliance as shown in equation 3.18, which is analogous to structural compliance, is the goal function being minimized. By minimizing the thermal compliance, the temperature at the heat flux boundary is minimized.

$$\begin{aligned} \min \quad & c(\rho) = \mathbf{T}^T\mathbf{f} \\ s.t \quad & \mathbf{K}(k_e(\mathbf{x}))\mathbf{T} = \mathbf{f} \\ & \int_\Omega \rho(x)\, d\Omega \leq V \end{aligned} \tag{3.18}$$

The gradient of the compliance, as given by Joo, et al., 2017 [**?**] is shown, without derivation, in equation 3.19. They found the derivative of the convection matrix $\mathbf{K}_h$ to be negligible, and the derivative of the forcing array is known to be zero. As a result, only the conductivity matrix $\mathbf{K}_c$ has an impact on the total gradient.

$$\frac{\partial c}{\partial \rho_e} = 2\mathbf{T}^T \frac{\partial \mathbf{f}_e}{\partial \rho_e} - \mathbf{T}^T \left(\frac{\partial \mathbf{K}_{e,h}}{\partial \rho_e} + \frac{\partial \mathbf{K}_{e,c}}{\partial \rho_e} \right) \mathbf{T} = -\mathbf{T}^T \frac{\partial \mathbf{K}_{e,c}}{\partial \rho_e} \mathbf{T} \tag{3.19}$$

In this thesis, domains are considered to be purely conductive. In spacecraft systems, the force of gravity is absent, so no natural convection will occur. This means that the convection matrix is irrelevant, and no assumptions need to be made for it. Of course, this is not the case for phase change domains, where latent heat effects are present, but the case for pure conduction will be considered as a highly simplified phase change domain.

Optimization of Lattice Structures

Optimization of lattice structures is essentially topology optimization considering orthotropic material properties. During the optimization loop, the orientation of the cells is considered to achieve greater stiffness. This can be achieved by aligning the unit cells along the principal stress axes [**?**]. A rotation matrix such as the one in equation 3.20b can be used to rotate the material tensor, as shown in equation 3.20a.

$$\mathbf{C}(x,\theta) = \mathbf{R}^T(\theta)\mathbf{C}(x)\mathbf{R}(\theta) \tag{3.20a}$$

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{3.20b}$$

The optimization loop is similar to a typical topology optimization, but during each iteration, the principal stresses need to be determined as well as their angle. The material tensor can then be rotated and recomputed until the change in angle is below a tolerance. The principle axis can be computed using equation 3.21.

$$\tan(2\theta_p) = \frac{2\tau_{xy}}{\sigma_x - \sigma_y} \tag{3.21}$$

In this thesis, this method has not been implemented. It had been decided early on that the added complication of considering the cell orientation was not as important as the multi-functional component of the thesis. A simple substep could be implemented, however, in future work.

After an optimal structure has been generated, it is de-homogenized as a post-processing step. Many methods have been proposed for this [**?**] [**?**] [**?**]. The de-homogenization step is required to return the finite element mesh into a meaningful lattice representation. The cited examples, however, are quite complex. In this thesis, the cell size was decided to be kept constant. That combined with no cell alignment means that the mesh can be projected back onto a lattice mesh, and some methodology can be determined for averaging the element relative densities over each unit cell. This is beyond the scope of this thesis, but a simple example has been implemented for visualization purposes in future chapters.

Multi-functional Topology Optimization

Topology optimization has been successfully applied to multi-functional problems. Many methods have evolved for optimizing lattice unit cells [**?**] [**?**] [**?**]. This thesis focuses on optimizing a domain given an already homogenized unit cell, so a similar approach to Pejman and Najafi, [**?**] has been adopted. Takezawa, et al., [**?**] also developed a method for structural optimization using stress and conduction constraints, where some ideas have also been adopted.

To generate a Pareto front, the weight method has been used, similar to how Challis, et al., [**?**] described. Here, the objective function is to minimize equation 3.22, where w_i are weights that sum to one of the relative importance of each separate function c_i.

$$J = -\left(w_1 c_1 + w_2 c_2\right) \tag{3.22}$$

Pejman and Najafi [**?**] proposed a method to obtain a Pareto front by first computing the single field problems to determine the bounds. The compliance of the single functions can then be normalized, and a Pareto front can be generated by incrementing the weights from 0 to 1. This methodology has been adapted for this thesis in chapter 5. The gradient of the objective function ends up being a sum of the gradients of the single functional components with a weight on each.

Figure 3.8 shows each step proposed by Pejman and Najafi [**?**]. In part A, the bounds of the Pareto front are determined by the single functional components. The maximum of the opposite function occurs when the other function is a minimum. Part B shows the Pareto front being normalized. Points are then evenly spaced along the Pareto front to obtain intermediate points in part C.

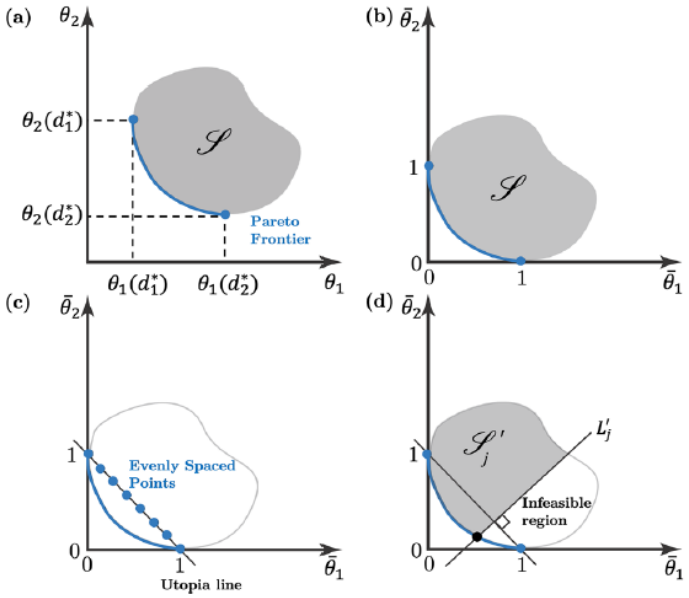


Figure 3.8: Illustrated steps proposed by Pejman and Najafi, 2023 [**?**]

3.4 Numerical Methods for Phase Change Problems

In chapter 2.2, the Stefan problem was briefly introduced. In this section, some numerical methods will be introduced which are capable of solving phase-change problems. Many methods exist to solve phase change problems such as those discussed by Zeneli et al. [**?**]

Front-tracking methods for instance explicitly track the phase transition front [**?**]. These methods first compute the location of the boundary and then solve the heat equation and other required equations ensuring the proper boundary conditions are satisfied. They generally use an adaptive or a deforming mesh to try to capture the phase transition front. This method has been successfully applied to the finite element method [**?**]. This method will not be discussed further, as having an adaptive or deforming mesh makes the topology optimization module extremely difficult to implement.

Front-fixing methods are similar to front-tracking methods. They explicitly track the phase transition front, but also fix the grid to the front. Due to this, this method has a high degree of complexity and is recommended for simple geometries [**?**]. This method will also not be discussed further, as it makes the topology optimization module very complex to implement.

The final, and most common family of methods are the fixed-domain methods. These methods are also much simpler to implement, and still have a high degree of accuracy, even without having a mesh that closely tracks the phase transition front. One of the most common methods is the enthalpy method [**?**]. In this method, the phase transition front is indirectly tracked by solving a modified heat equation in terms of enthalpy 3.23. This method could be easily adapted

to topology optimization as it uses a fixed mesh. However, in this thesis, the apparent heat capacity method, which is yet to be discussed, was selected for its temperature formulation.

$$\frac{\partial H}{\partial t} = \alpha_H \nabla^2 H = \alpha c_p \nabla^2 T \tag{3.23}$$

The level set method has also successfully been used to capture phase-change problems [**?**]. In the level-set method, the transition boundary is tracked using a signed distance function, and described by equation 3.24. Where the function is zero, is where the phase transition boundary lies. The level set function changes in time by some velocity, which is governed by the Stefan condition.

$$\varphi = \begin{cases} -d, & \text{Solid} \\ 0, & \text{Interface} \\ d, & \text{Liquid} \end{cases} \tag{3.24}$$

Heat Capacity Method

The heat capacity method is similar to the enthalpy method in that it solves a slightly modified heat equation. However, it is formulated directly around the temperature making it easy to implement. The heat-capacity method (as well as other fixed domain methods) makes use of a so-called "mushy zone" which relaxes the jump condition at the phase transition boundary. Figure 3.9 illustrates how the mushy-zone can be modeled with different functions. The plot describes the phase fraction evolution, where zero represents a solid and one represents a liquid. By taking the derivative of these functions, a Dirac-delta function is approximated.

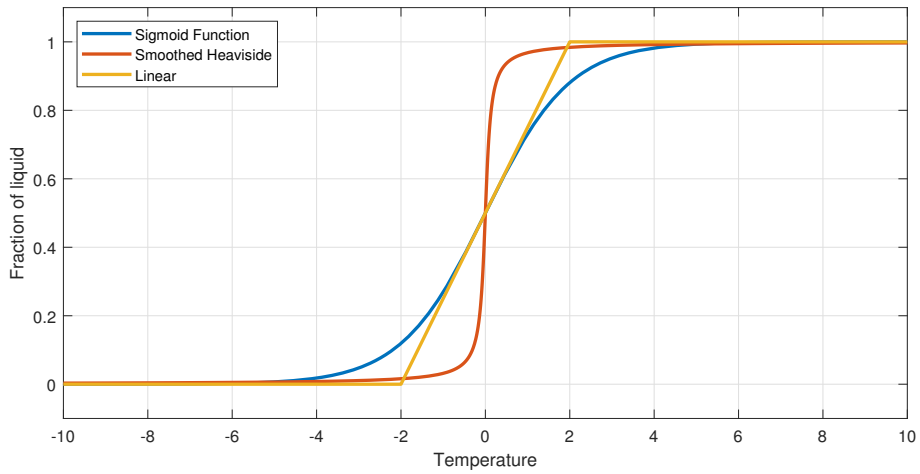


Figure 3.9: Various phase transition functions

The apparent heat capacity method [**?**] has been utilized for this thesis. The heat capacity method works by having the heat capacity, as well as other material properties, be temperature-dependant variables, as shown in figure 3.10. Across the mushy zone, the heat capacity jumps as a result of the latent heat of melting. The material properties can be smoothed to improve the solver's stability using the suggested functions in figure 3.9. The sigmoid function has been used with particular success [**?**].

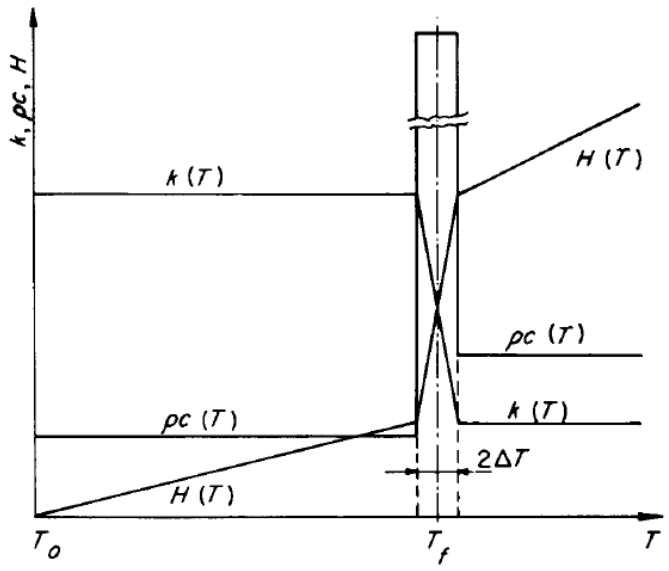


Figure 3.10: Material properties variation over phase change boundary [**?**]

In the apparent heat capacity method, the time-dependent heat equation shown in equation 3.25 is solved. Where ρ is the material density, c_{app} is the apparent heat capacity, and k is the thermal conductivity. Some materials, such as paraffin wax have a relatively constant thermal conductivity, so for this analysis, it will be considered constant.

$$\rho c_{app}(T)\frac{\partial T}{\partial t} = \nabla \cdot (k\nabla T) + q \tag{3.25}$$

The heat capacity is defined by $c_{app} = dU/dT$ for a constant volume process, where U is the internal energy. In this case, the specific internal energy u is a sum of the heat capacity c_p and latent heat L effects [**?**], as shown in equation 3.26.

$$u = \int_{T_{ref}}^{T} c_p dT + L f_p \tag{3.26}$$

Equation 3.26 introduces the phase fraction f_p which can be defined in various ways, as figure 3.9 shows. Some equations are given below in equation 3.27. The phase fraction ranges from zero to one and is used to track how much a material point has melted. In the equations below, the variable a can control how tight the phase transition is.

$$f_p(T) = \frac{1}{1+e^{-a(T-T_m)}}, \quad \text{Sigmoid Function} \tag{3.27a}$$

$$f_p(T) = \lim_{a\to 0} \frac{1}{\pi}\tan^{-1}\left(\frac{T-T_m}{a}\right) + \frac{1}{2}, \quad \text{Smooth Heaviside} \tag{3.27b}$$

$$f_p(T) = \begin{cases} 0, & \text{In solid phase} \\ \frac{1}{2\Delta T}(T-T_m), & \text{In mushy zone} \\ 1, & \text{In liquid phase} \end{cases}, \quad \text{Linear Function} \tag{3.27c}$$

Substituting the apparent heat capacity into the heat equation, the continuous description of the problem is described as shown in equation 3.28 [**?**]. In this equation, the latent heat effects are only present when the phase fraction is in the mushy zone ($0 < f_p < 1$). This equation is also clearly non-linear and can be severely non-linear when the latent heat effects are large, and the mushy zone is small.

$$\rho\left(c_p + L\frac{\partial f_p}{\partial T}\right)\frac{\partial T}{\partial t} = k\nabla^2 T + f \tag{3.28}$$

After discretization, a discrete system of equations can be found. Equation 3.29a uses the derivation by Celentano et al. [**?**] and equation 3.29b gives the finite element matrices. Note that $\dot{\mathbf{L}} = \mathbf{L}^{n+1} - \mathbf{L}^n$.

$$\mathbf{K}T + \mathbf{C}\dot{T} + \dot{\mathbf{L}} = \mathbf{F} \tag{3.29a}$$

$$\begin{aligned} \mathbf{K}^e &= \int_{\Omega^e} (\nabla \mathrm{N})^T k \nabla \mathrm{N}\, d\Omega^e + \int_\Gamma h \mathrm{N}\mathrm{N}^T d\,\Gamma \\ \mathbf{C}^e &= \int_{\Omega^e} \rho c_p \mathrm{N}\mathrm{N}^T\, d\Omega^e \\ \mathbf{L}^e &= \int_{\Omega^e} \mathrm{N}\rho L f_p\, d\Omega^e \\ \mathbf{F}^e &= \int_{\Omega^e} \mathrm{N}\rho b\, d\Omega^e + \int_\Gamma \mathrm{N} h T_\infty\, d\Gamma \end{aligned} \tag{3.29b}$$

Using backward Euler, the equation can be rewritten in the residual form shown in equation 3.30a. Taking the derivative of this gives the Jacobian shown in equation 3.30b. Note that the subscript i denotes an iteration index of an iterative scheme such as Newton's method.

$$\mathbf{R}(T_i^{n+1}) = \mathbf{F}\Delta t + \mathbf{C}T^n - (\mathbf{L}_i^{n+1} - \mathbf{L}^n) - (\mathbf{C} + \mathbf{K}\Delta t)T_i^{n+1} \tag{3.30a}$$

$$\mathbf{J}(T_i^{n+1}) = -\frac{\partial \mathbf{R}}{\partial T}\bigg|_i^{n+1} = \mathbf{K}\Delta t + \mathbf{C} + \frac{\partial \mathbf{L}}{\partial T}\bigg|_i^{n+1} \tag{3.30b}$$

An iterative scheme can then be used to determine the temperature. First, the descent direction is computed using equation 3.31.

$$\Delta T = [\mathbf{J}(T_i^{n+1})]^{-1}\mathbf{R}(T_i^{n+1}) \tag{3.31}$$

The descent direction can then be used in an iterative scheme such as Newton's method ($\alpha = 1$), damped Newton's method ($0 < \alpha < 1$), or a line search to determine a step-size α.

$$T_{i+1}^{n+1} = T_i^{n+1} + \alpha\Delta T \tag{3.32}$$

Effective Heat Capacity Method

The effective heat capacity method [**?**] is another heat capacity method worth mentioning that considers the temperature distribution between the nodes in a mesh. It resolves the issue that a narrow mushy zone may occur inside the element, as illustrated in figure 3.11. The left figure shows an element that would not capture the apparent heat capacity, and would essentially skip any melting effects. The right shows a node that happens to capture the apparent heat capacity, but it would overestimate its effects. This method can be used to minimize the size of the mushy zone, but as topology optimization does not require highly accurate simulations, this method has not been used and will therefore not be discussed further.

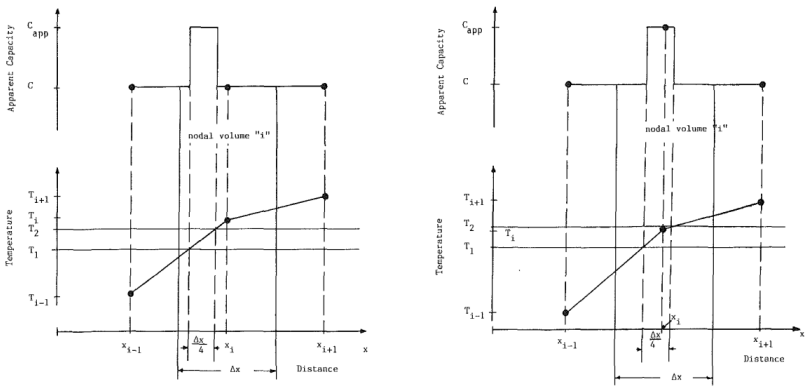


Figure 3.11: Heat capacity across a 1D element [**?**]. Element failing to capture mushy-zone (left), and an element whose node captures the apparent heat capacity (right)

4 Single Functional Optimization

Before attempting to optimize the full multi-functional problem, the single-functional problems are optimized. The approach utilized in this thesis is to use the topology optimization techniques discussed in chapter 3.3 and apply them to the lattice structure. The lattice properties can be obtained from equations given by Bühring et al., 2022 [**?**] and Soika, 2023 [**?**]. In the following chapters, the simple setup shown in figure 4.1 has been used. The figure shows where the thermal boundary will be (red) and where a distributed load will be defined (blue). The overall domain had a mesh generated using *Gmsh* [**?**], and the finite element method has been used for all simulations. Lastly, *Paraview* [**?**] has been used for visualization. For this thesis, the Julia programming language has been used [**?**].

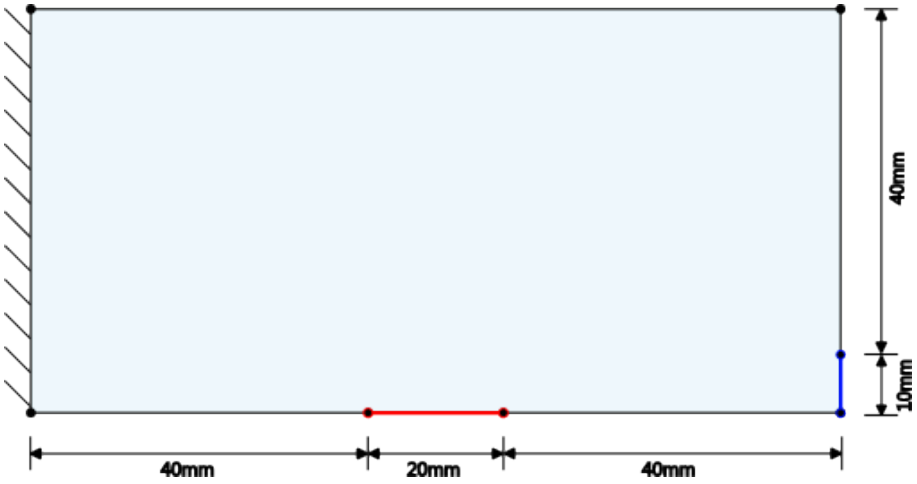


Figure 4.1: Simple simulation setup with thermal and structural boundary conditions

4.1 Methodology

Mapping of Lattice Properties

One of the main ideas of this thesis was to use the lattice properties directly as the material properties in the simulations. This is achieved by recognizing the relative density of the finite elements can be mapped to the porosity of the unit cells. The porosity depends on the selected cell, but generally, the porosity is a function of the cell height, strut radius, and aspect ratio. For this thesis, a fixed cell height has been used, as well as a fixed aspect ratio. Therefore, the

porosity is only a function of the strut radius. As an example, the porosity of the bcc unit cell is given in equation 4.1 [**?**].

$$\varepsilon = 1 - 2\tan^2(\Omega)\frac{r^2}{h^2}\left(\pi\frac{4}{sin(\Omega)} - \frac{16r}{3h}\left(\frac{2.993}{\sin(\pi - 2\Omega)} + \frac{3.340}{\sin(\pi/2 - \Omega)}\right)\right) \tag{4.1}$$

The valid range for the unit cells porosity ε is typically said to be from 2.5% to 20%. However, the relative density ρ in a topology optimization needs to range from 0% to 100%. Therefore a linear mapping, such as the one shown in equation 4.2, is used.

$$\varepsilon = \rho(\varepsilon_{\min} - \varepsilon_{\max}) + \varepsilon_{\min} \tag{4.2}$$

To determine the lattice properties, the strut radius is required. Once the required porosity is found from the relative density, the strut radius can be found quickly using a root finding method, such as those discussed in chapter 2.4.

Implementation

With all the required tools implemented, the optimization methodology closely resembles that of the classical SIMP method. However, some minor adaptations have been made. The first is the use of orthotropic material properties, and the second is a slight modification of how the element sensitivities are computed.

One aspect that was proposed in this thesis was the use of automatic differentiation (AD). Automatic differentiation is a powerful tool in some scenarios, but when used for the full program, it became a performance bottleneck and provided no benefits. However, AD has been successfully used with high efficiency in two areas. The first is when computing the strut radius of the cells. To determine the strut radius, Newton's method was used which requires a gradient. The gradient can be efficiently computed using the library *ForwardDiff* [**?**]. Another area where the AD is very useful is for computing the finite element derivatives. Each element is a function of only the relative density, so *ForwardDiff* [**?**] can be efficiently used here to compute both the element matrix and its derivative. Regardless of the problem being solved, the following steps shown in figure 4.2 are followed.

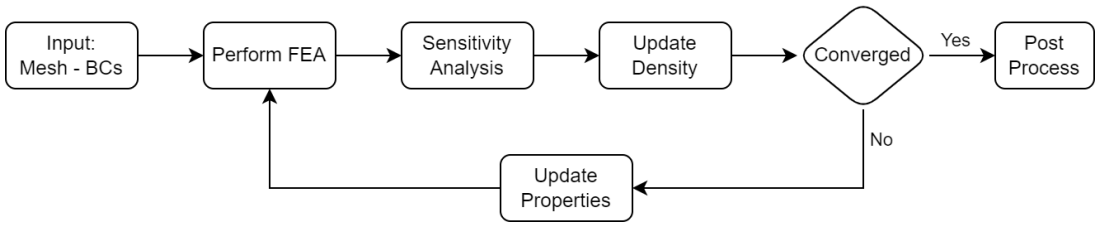


Figure 4.2: High-level optimization flow-chart

4.2 Refinement Studies

Mesh Refinement

One problem with finite element analysis is a phenomenon known as mesh dependency. This occurs because the solution obtained using the finite element method is an approximation of a continuous domain. The finite element method discretizes the domain into elements where a set of polynomials is solved. As the elements become smaller through refinement, the approximation approaches the true solution. By comparing the results obtained on different meshes, a measure of the mesh dependency can be obtained. If the results are the same, then mesh independence is achieved, and the results are more likely to represent the true solution.

Four mesh refinement levels have been used in the following study that have been summarised in table 4.1. The same filter radius needed to be used on each mesh as changing the filter leads to a different equation being solved, as has been mentioned in chapter 3.3. The porosity range for these studies was set from 0% to 20% to obtain clearer results.

	Number of elements	Element size
Coarsest	200	5.0mm
Coarse	800	2.5mm
Medium	3200	1.25mm
Fine	5000	1.0mm

Table 4.1: Mesh data

Structural Problem

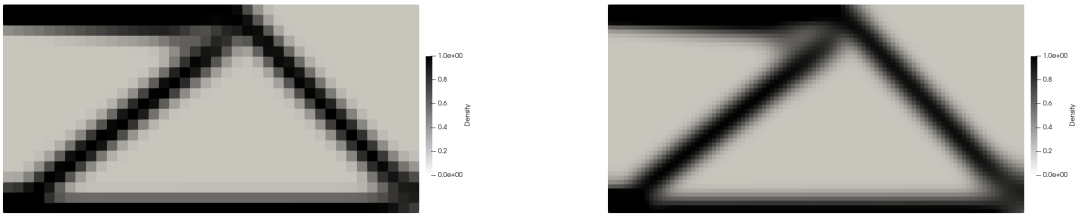
Just by observation, results obtained for the coarse and fine mesh, shown in figure 4.3 for the structural problem, appear to be the same. To further ensure the results are similar, the structural compliance of each structure has been computed as shown in table 4.2, and the relative error w.r.t the fine mesh has been determined.

Mesh Level	Coarsest	Coarse	Medium	Fine
Compliance	16.6386×10^{-3}	15.5507×10^{-3}	15.6140×10^{-3}	15.6204×10^{-3}
Relative Error	6.52%	0.44%	0.04%	-

Table 4.2: Compliance results from mesh refinement study

As was mentioned, the coarse mesh already produces structures that closely resemble the fine mesh. Both structures shown in figure 4.3 have the same features, so if the goal is to obtain a

structure, without much regard for the accuracy of the simulation, the coarse mesh is already appropriate, given a large filter radius. However, using a large filter radius leads to local minima being found as will be shown in the filter radius refinement study, so a decision regarding the trade-offs needs to be made.



(a) Optimized coarse mesh (b) Optimized fine mesh

Figure 4.3: Mesh refinement study for structural problem of two mesh levels

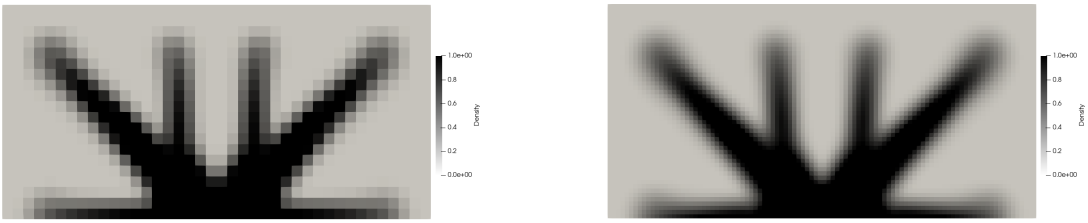
Thermal Problem

Once again, by observation, the results between the coarse and fine mesh appear to be pretty similar, as shown in figure 4.4. To measure how similar the results are, the maximum temperature was taken for each mesh level. The results are essentially the same for each mesh level, so in this regard, any mesh could be chosen.

Mesh Level	Coarsest	Coarse	Medium	Fine
Max Temperature [K]	450.130	450.171	450.176	450.176
Relative Error	0.010%	0.001%	0.000%	-

Table 4.3: Compliance results from mesh refinement study

A similar conclusion to the structural results can also be drawn for the thermal mesh refinement study. That being if only the generated structure is important, and the actual temperature results are not of any concern, the coarse mesh can be used.



(a) Optimized coarse mesh (b) Optimized fine mesh

Figure 4.4: Mesh refinement study for thermal problem of two mesh levels

Filter Refinement Studies

As was discussed in chapter 3.3, the use of a filter is a necessity in topology optimation to avoid the *checkerboard problem*. Furthermore, by using a filter, a mesh-independent result can be obtained, as has already been shown in the previous subsection. One major drawback of using a filter, however, is that a different problem is being solved than the original thermal or structural problem [**?**]. This essentially means the resulting structure depends both on the filter radius and how fine the mesh is, as a finer mesh allows a smaller filter radius to be used which in turn results in higher-resolution structures being generated.

For the filter refinement study, three filter levels were used, as well as the fine mesh which has 1*mm* elements. This means the minimum filter size is 1mm, otherwise, the filter will not intersect with any of the neighboring elements. The chosen filter radii are 5*mm*, 2.5*mm* and 1.5*mm*. The smallest possible filter radius should be used to minimize the number of grey elements, but also to get as close to the true structural or thermal problem as possible.

Structural Problem

The optimized results for the structural problem show distinctly different structures. In these structures, the sensitivities are filtered over too many elements resulting in a completely different structure. The only advantage of using the large filter radius is that mesh-independent results can be obtained for very coarse meshes. To have a measure of how different the results are, the compliance of each structure has been tabulated in table 4.4. The relative error w.r.t the filter radius of 1.5mm is also given. The results show that the filter has a significant impact on the results.

(a) Filter radius of 5mm

(b) Filter radius of 2.5mm

(c) Filter radius of 1.5mm

Figure 4.5: Filter refinement study for the structural problem

Filter Radius	5mm	2.5mm	1.5mm
Compliance	$17.5207 \times 10^{-}3$	$16.0625 \times 10^{-}3$	$15.8576 \times 10^{-}3$
Relative Error	10.49%	1.29%	-

Table 4.4: Compliance results from filter refinement study

Thermal Problem

The structures obtained from the thermal topology optimization each produce six main branches, but when the filter radius is reduced, smaller branches are allowed to form. To get a measure of the difference between the results, the thermal compliance has been computed for each problem, and tabulated in table 4.5. While these results have no real-world application, it is interesting to see that the compliance increases slightly with the coarsest mesh. The reason they have no real-world application is because the simulation has a uniform *convection* throughout the domain. In reality, convection would only occur on the boundary of the branches that form.

(a) Filter radius of 5mm

(b) Filter radius of 2.5mm

(c) Filter radius of 1.5mm

Figure 4.6: Filter refinement study for the thermal problem

Filter Radius	5mm	2.5mm	1.5mm
Compliance	1012.79	1012.73	1012.74

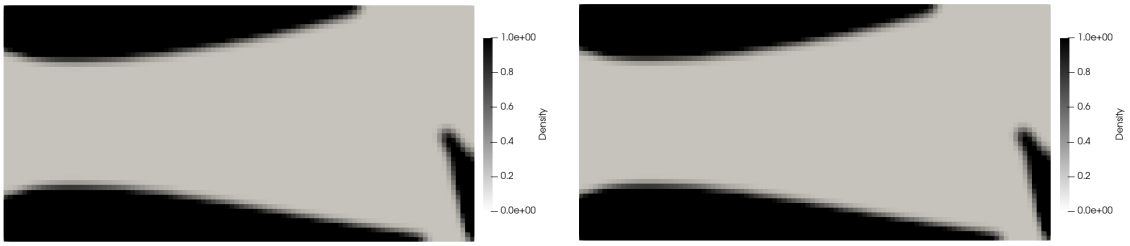
Table 4.5: Themal compliance results of filter refinement study

Scaled Boundary Condition Invarience

Recall that the compliance is computed as $c = u^T f$, where u could be the displacements or temperature vector, and f is the forcing array or right-hand side of a linear system of equations. The gradient of this vector is, generally, given in equation 4.3.

$$\frac{\partial c}{\partial \rho_e} = u_e^T \frac{\partial K_e}{\partial \rho_e} u_e \tag{4.3}$$

This gradient does not consider how much the structure deforms, only the general shape that it deforms into. That is to say, it only considers its linear deformation pattern. Therefore, to minimize compliance, only the material properties are considered. To illustrate this point, two structures have been generated with a distributed load of $1N/m$ and $100kN/m$, as shown in figure 4.7. The generated structures show no difference.



(a) Distributed load of $1N/m$ (b) Distributed load of $100kN/m$

Figure 4.7: Illustration of the scaled boundary force invarience

If instead non-linear properties were considered, a non-linear simulation would need to be computed. In this case, the generated structure could be different, as is shown in literature [**?**]. This is also likely the case for the phase change problem, as the forcing array has a significant impact on how long it takes to melt the domain. For a slow process with a low heat flux, the energy storage will resemble a sensible energy storage. In such a case, the thermal conductivity might have more of an impact on the optimal structure.

4.3 Structural Optimization

The structural optimization can be implemented in essentially the same way as typical topology optimization, as discussed in chapter 3.3. The only big difference in terms of implementation is the use of an orthotropic material tensor, shown in equation 4.4. The elastic moduli in

each direction are then obtained using the functions determined by Bühring et al., [**?**]. In this implementation, the orientation of the unit cells is not optimized.

$$D = \begin{pmatrix} \frac{E_x}{1-\nu_{xy}\nu_{yx}} & \frac{\nu_{xy}E_y}{1-\nu_{xy}\nu_{yx}} & 0 \\ \frac{\nu_{xy}E_y}{1-\nu_{xy}\nu_{yx}} & \frac{E_y}{1-\nu_{xy}\nu_{yx}} & 0 \\ 0 & 0 & G_{xy} \end{pmatrix} \tag{4.4}$$

In chapter 3.3, the OCM and GOCM methods were discussed. While this implementation only has one constraint (that being mass), the GOCM method is preferred as it has a slight performance benefit. It may also be possible to add other constraints in the future by keeping it generic.

Observations

Some results have been obtained for the structural optimization using the lattice properties. The first result, shown in figure 4.8 shows the optimized structure obtained using a porosity range of 5% to 20%. The structure that is obtained does not resemble a typical topology optimization result. Usually, a topology optimization produces a structure that resembles a truss. This result instead has disconnected regions of low porosity. It is important to realize that these empty regions contain cells of 5% porosity, so in reality, each region is connected by relatively stiff cells.

Figure 4.8: Optimized structure using lattice properties with a porosity of 5% to 20%

The reason for the '*disconnected*' regions results from the small range in the elastic modulus. In typical topology optimization, the elastic modulus is usually set to one, and the minimum is zero. To prevent division by zero singularities, the minimum relative density is chosen to be a small number in the order of 1×10^{-3}. This means when a penalization value of 3 is used, the

minimum value of the elastic modulus is 1×10^{-9}. This range of elastic modulus means that the regions of zero relative density have a negligible contribution to the stiffness of the structure. When using the lattice properties, the range of the elastic modulus is on the order of 1×10^5 to 1×10^6. In this case, the empty regions are not negligible, and contribute a significant amount of stiffness to the structure.

To validate this fact, a new structure, shown in figure 4.9, has been generated that uses a larger range in elastic modulus. In this optimization, the porosity was varied from 0% to 20%, meaning the empty regions in the structure are void of any lattice material. The resulting structure resembles a typical topology optimization. At first glance, this appears to be a more reasonable result. However, it is important to keep in mind that compliance is a function of the material properties. Therefore, a different function is being minimised which should produce a slacker structure.

Figure 4.9: Optimized structure using lattice properties with a porosity range of 0% to 20%

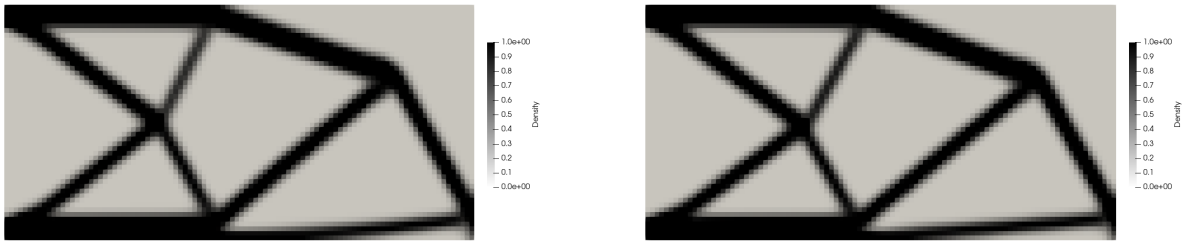
After obtaining both structures, the lattice properties were set back to the range of 5% to 20% and the compliance of the structure was computed. As is to be expected, the first structure shown in figure 4.8 had a lower compliance (meaning a higher stiffness). It is expected to perform better because the compliance is minimized directly considering the lattice properties.

	Porosity 5% - 20%	Porosity 0% - 20%
Compliance	9.3696×10^{-3}	9.9976×10^{-3}
Maximum Displacement	0.4950mm	0.5313mm

Table 4.6: Results comparison between different structures with different porosity range

As was shown in figure 3.3, the lattice properties are slightly non-linear. To see whether these effects have any impact on the optimization, another structure was generated using a linear

interpolation between the maximum and minimum values from the lattice. Note that for these structures, the porosity range was set from 0% to 20%.



(a) Structure generated using a linear interpolation of the lattice properties

(b) Structure generated using the lattice properties directly

Figure 4.10: Examples of optimized domains for structural and thermal problems

The two structures only have minor differences. The displacements and compliance are also almost identical. As a result of this, the lattice properties do not have to be used directly, which could give a slight performance benefit. For this thesis, however, the lattice properties will be used directly to ensure the results are all comparable.

	Interpolated Structure	Lattice Properties
Compliance	14.2887×10^{-3}	14.2702×10^{-3}
Maximum Displacement	0.7510mm	0.7501mm

Table 4.7: Results comparison between different structures with different porosity range

4.4 Thermal Optimization

For the thermal topology optimization, two approaches were investigated. The first, which is used in literature, maximizes the thermal conductivity in the domain by minimizing thermal compliance. This method has been discussed in chapter 3.3. The second approach is to minimize the maximum temperature after a certain time has elapsed in a phase change simulation. As with the structural problem, an orthotropic material tensor, as shown in equation 4.5 is required.

$$D = \begin{pmatrix} k_x & 0 \\ 0 & k_y \end{pmatrix} \tag{4.5}$$

Maximising Thermal Conductivity

This approach has been attempted in literature to some degree [**?**]. The main drawback with phase change materials is their low thermal conductivity, and hence low thermal diffusivity. How evenly a phase change material melts is directly related to these values. So the idea of maximizing thermal conductivity in the domain is to attempt to have the entire domain melt as evenly as possible. This should help keep the heat source at a constant temperature for a longer duration.

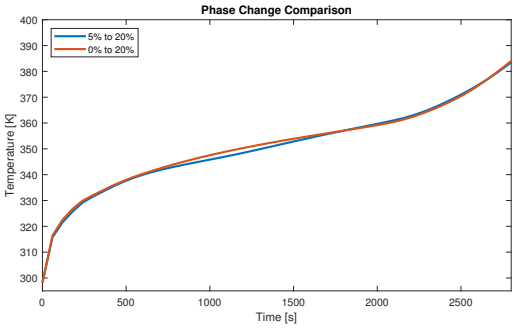
Observations

A collection of results has been obtained for the thermal topology optimization using the lattice properties. The first result, as shown in figure 4.11, which uses the recommended lattice porosity range, has been obtained. Much like the structural topology, the structure does not resemble typical topology optimization results. However, unlike the structural optimization result, these results are more likely to be unreasonable. This is for two reasons. The first problem with this topology optimization is that it has no consideration of the phase change problem, which is what the structures will be used for. The second reason is there are large empty regions containing only the low-conductivity PCM and low-conductivity cells. This means that when the PCM in the dark regions has fully melted, the heat has to diffuse through a low diffusivity front. This would increase the temperature gradient resulting in a high temperature at the heat source, but a domain that still has regions of solid phase.

Figure 4.11: Thermal topology optimization using lattice properties with a porosity of 5% to 20%

To test this theory, a different structure was generated. To generate this structure, different material properties were needed. To keep the results comparable, lattice properties in the range of 0% to 20% were used. Note that this was *only* to generate a different structure, and when performing a phase change simulation, the porosity range was set back to the range 5% to 20%.

(a) Thermal topology optimization using lattice properties with a porosity of 0% to 20%



(b) Temperature evolution over phase change simulation

Figure 4.12: Topology optimization results obtained using a larger porosity range (left) and the resulting phase change simulation compared to the optimal porosity range (right)

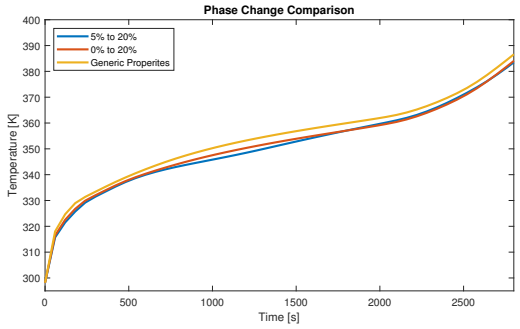
The generated structure is shown in figure 4.12a. As is shown, the branches in the structure reach further into the void regions. This should result in the temperature being kept more constant for longer. One slight problem with this resulting structure, however, is that there are gray regions of intermediate density. Because of the nature of the latent heat curve shown in figure 3.3, these elements are undesirable and may result in the PCM melting faster.

The temperature evolution plot, shown in figure 4.12b, agrees with the above-discussed theory. The structure generated using a porosity range of 0% to 5% does rise faster, and remain constant for longer. However, because of the number of *gray* elements, the total latent heat in the system is slightly lower, as shown in table 4.8, and is likely the cause of the slightly increased temperature at the end of the phase change simulation. However, the difference is minimal, which implies that the material distribution is still near-optimal. In this case, compared to the structural topology optimization, it is less clear which structure is *'better'* because both have essentially the same final temperature.

Another note for these structures is that the range in thermal conductivity is quite small, even when the porosity is 0%. This is because the phase change material has a fairly high thermal conductivity compared to the maximum. This provides only a small change in the order of magnitude which is why there are minimal branches generated. To determine whether using the lattice material properties provides any benefit, a final structure has been generated that uses generic thermal conductivity in the range of zero to one. The resulting structure, shown in figure 4.13a, has many small branches as well as a lot of gray elements. As has been mentioned, these gray elements are undesirable due to the non-linear nature of the latent heat curve shown in figure 3.3. As a result, this structure has the lowest total latent heat.

Comparing all the structures in a phase change simulation shows that the lattice properties have a significant benefit in minimizing the temperature. However, this is most likely because the total latent heat available to each system varies. Figure 4.13b shows that the structure generated with generic properties performs the worst right from the start. At this point, the system is conduction dominant, so it could imply that the generic properties do not capture the

(a) Typical thermal topology optimization using generic material properties



(b) Temperature evolution of all the optimized structures

Figure 4.13: Topology optimization results obtained using generic material properties (left) and the resulting phase change simulation compared to all other structures (right)

true conductivity well enough to optimize the system. This means that the poor placement of high porosity elements around the heat source prevents the initial heat flux from being diffused quickly through the structure.

Structure	5% - 20% Porosity	0% - 20% Porosity	Generic properties
Total Latent Heat [MJ]	740.207	738.489	734.139

Table 4.8: Total available latent heat in each structure

To further show that the structure generated for the porosity range of 5% to 20% is the best option, another plot has been generated to show the phase fraction of the central node on the heat flux boundary. The plot includes the phase fraction obtained for another structure that was generated for the porosity range of 10% to 20%. At the time this node is melting, most of the domain is below the melting temperature, so the domain can be considered as conduction dominant. This is likely why the 5% to 20% structure performs the best.

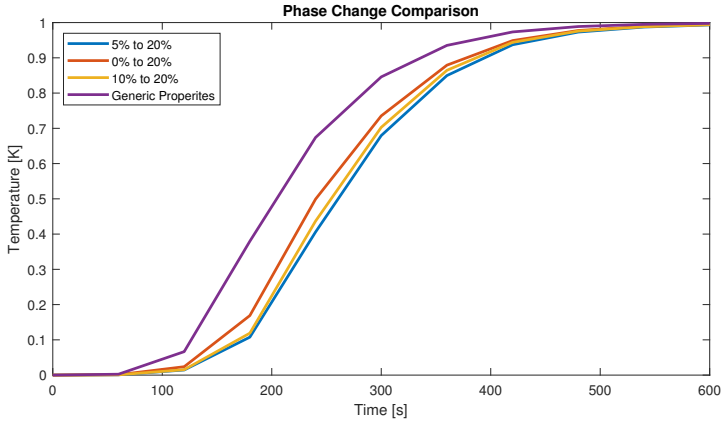


Figure 4.14: Phase fraction at the hottest node in domain

Minimising Phase Change Compliance

This approach appears to be a novel idea where the transient effects are taken into account when finding an optimum. This would be similar to a non-linear topology optimization. The problem with this method is that there is no simple way to compute the derivative of the compliance. Recall that the compliance is defined as shown in equation 3.18. However, in the transient phase change problem, the derivation of the element sensitivities can not be used. Beginning with the gradient of the compliance, shown in equation 4.6, a new derivation can be achieved which is given in Appendix B.

$$\frac{\partial c}{\partial \rho} = \frac{\partial \mathbf{T}^T}{\partial \rho}\mathbf{f} + \mathbf{T}^T \frac{\partial \mathbf{f}}{\partial \rho} \tag{4.6}$$

The main problem with this equation is the $\frac{\partial \mathbf{T}}{\partial \rho}$ which can not be easily computed. Automatic differentiation could be used, but using the *ForwardDiff* library is far too slow to be viable. The main issue is that in the forward pass, each dual variable is traced through the entire program, but most of them have no effect on the final result as the forcing vector $\mathbf{f}$ is mostly zeros. Reverse mode AD could be used instead, but because multiple Newton iterations are needed to resolve the non-linearity very high amounts of memory are required. It is for these reasons the gradient was derived, and has been computed manually.

Implementation

The latent heat vector and latent heat matrix can not be computed using automatic differentiation because they require the gradient of the temperature vector w.r.t the relative density. To compute them using AD would mean the entire simulation needs to be computed using AD which is what is being avoided. Here the element's relative density will be denoted using φ to avoid confusion with the material density ρ. Recall that N is the element shape functions, L is the latent heat, and f_{pc} is the phase fraction. Equation 4.7 shows the gradient of the element latent heat vector, and a similar gradient can be obtained for the latent heat matrix.

$$\begin{aligned}
L_e &= \int_{\Omega^e} N\rho(\varphi)L(\varphi)f_{pc}(\varphi)\, d\Omega^e \\
\frac{\partial L_e}{\partial \varphi} &= \int_{\Omega^e} N\frac{\partial}{\partial \rho_e}\left(\rho(\varphi)L(\varphi)f_{pc}(T)\right)\, d\Omega^e \\
&= \int_{\Omega^e} N\left(Lf_{pc}\frac{\partial \rho}{\partial \varphi} + \rho f_{pc}\frac{\partial L}{\partial \varphi} + \rho L\frac{\partial f_{pc}(T)}{\partial \varphi}\right)\, d\Omega^e \\
&= \int_{\Omega^e} N\left(Lf_{pc}\frac{\partial \rho}{\partial \varphi} + \rho f_{pc}\frac{\partial L}{\partial \varphi} + \rho L\frac{\partial f_{pc}(T)}{\partial T}\frac{\partial T}{\partial \varphi}\right)\, d\Omega^e
\end{aligned} \tag{4.7}$$

Some specific implementation stages have been used in this optimization problem to obtain results as quickly as possible. Because the global matrices are assembled as a sum of the element matrices, the derivative of the matrices w.r.t any element's relative density is simply the derivative of the element matrix. This is illustrated in equation 4.8. These element derivatives can be efficiently computed using *ForwardDiff* while computing each finite element matrix.

$$\frac{\partial \mathbf{K}}{\partial \rho_e} = \frac{\partial \mathbf{K_e}}{\partial \rho_e} \tag{4.8a}$$

$$\frac{\partial \mathbf{C}}{\partial \rho_e} = \frac{\partial \mathbf{C_e}}{\partial \rho_e} \tag{4.8b}$$

Observations

The results obtained for the phase change compliance are quite interesting because the results depend on how long the simulation runs. Three example cases have been generated. The first case runs for a longer time than required to have the domain fully melt. This essentially means the gradient is conduction-dominant. The second case only runs for as long as it takes to fully melt the domain. The third case elapses for a fixed time that does not allow the domain to fully melt.

For these structures, a coarse mesh needed to be used to produce results in a reasonable time. Even though the gradients are computed much faster, they still take at least a couple of hours to fully produce. As a result of using a coarser mesh, the filter radius had to be increased. For that reason, the results will be compared to a new pure conduction optimized structure shown in figure 4.15.

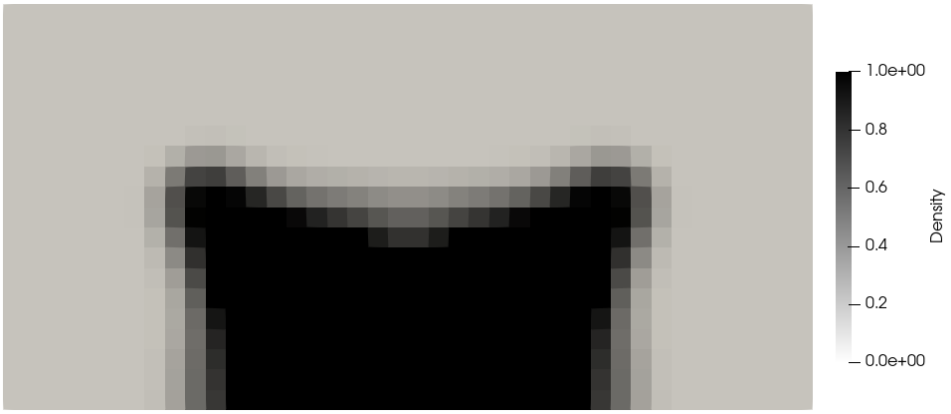


Figure 4.15: Pure conduction optimized structure

The first example, which will be referred to as the conduction dominant case, is shown in figure 4.16. The produced structure is similar, though much softer than the pure conduction case. Very faint *'ears'* are forming, similar to those that formed in the pure conduction case.

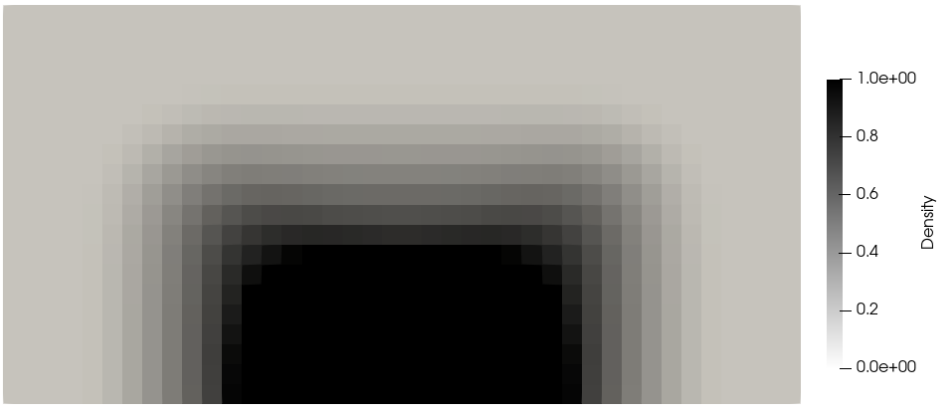


Figure 4.16: Conduction dominant structure

The second example, which will be referred to as the latent heat dominant case, is shown in figure 4.17. The produced structure is much different from the previous two. In this case, the latent heat effects don't have time to be overpowered by the conduction effects. This structure contains two strong branches, which should in theory sustain a larger melting front. This means that the temperature should be held constant for a longer time.

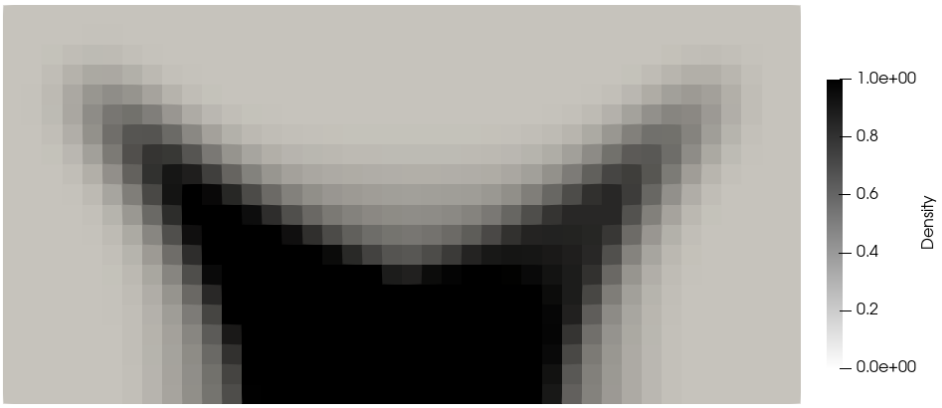


Figure 4.17: Latent heat dominant structure

The third example, which will be referred to as the incomplete melting case, is shown in figure 4.18. This structure was optimized for the end time of 1800 seconds, or 30 minutes. This structure is very similar to the conduction dominant structure, except it has a larger *"smoothing"* radius. This likely means the melting front is smoother, which should subject more elements to melting at once. What is obviously expected is that this structure should perform the best for the time that it was fixed, but will likely perform worse overall.

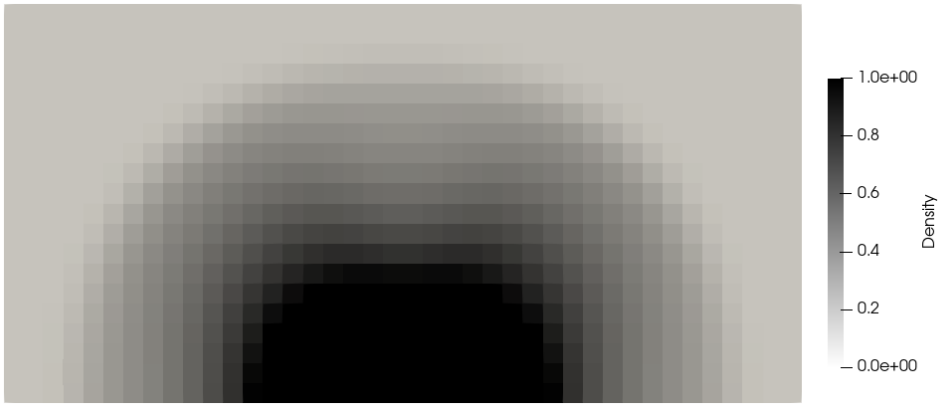


Figure 4.18: Incomplete melting structure

To be able to compare all the structures that have been obtained, a phase change simulation has been performed for each. Figure 4.19 shows the maximum temperature over time for each structure. The results are similar for each, but what is most interesting is that the pure conduction case performs the worst. The total latent heat available to that structure was the greatest of each of the examples. This highlights the fact that total latent heat does not ensure the best possible structure, but rather the placement of the *grey* elements is more important. It also shows that optimizing for phase change can provide a benefit.

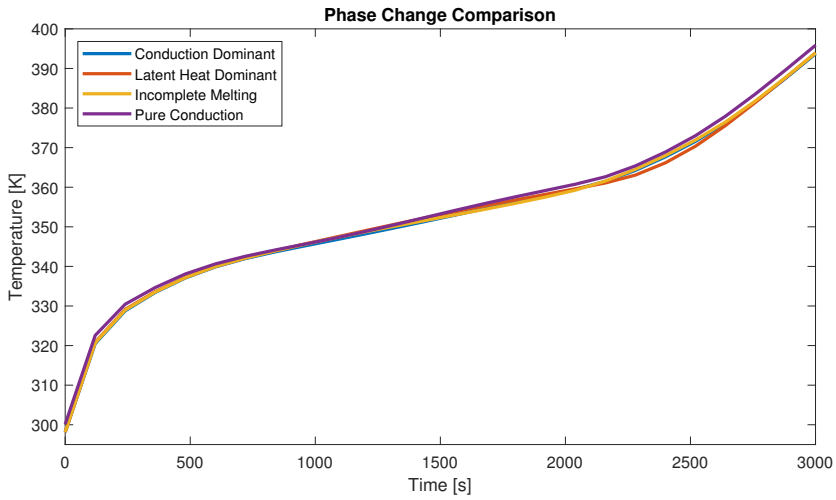


Figure 4.19: Phase change simulation results

The differences between the phase change optimal structures, on the other hand, are almost insignificant. What is most interesting, however, is that the incomplete melting case does perform best until the time it was optimized for. This is to be expected but highlights an important consideration that needs to be made if the structure is not expected to fully melt.

What is not very clear from the plot is that the latent heat dominant example does keep the temperature lower for longer, but after enough time has elapsed, the temperature rises to the same as the conduction dominant case. This result is also important, because if the goal is to keep the temperature constant for as long as possible, then a latent-heat-optimized structure should be generated. In this example, it was kept cooler for around 200 seconds longer, which could provide some benefit to some systems.

The time cost of producing the phase change optimized structures is quite significant. Many aspects of the code could be optimized and parallel computing could be implemented, but even so, for 3D structures with a fine mesh, the time to produce a structure could take many hours and would likely exceed a day.

4.5 Remarks

From the results obtained for the single functional components of this thesis, it is already clear that using lattice properties plays an important role in the kinds of structures that are produced. One can not simply neglect the lattice properties and use generic properties, as could be done for traditional topology optimization. This is because the *empty* regions with a relative density of 0 still provide a significant amount of stiffness or conduction which helps to minimize the compliance.

Based on results from both the refinement studies that were conducted, a trade-off between mesh size and filter radius needs to be reached. Using a coarse mesh produces the same structure as the finest mesh, but results in a large filter radius being required. This reduces the performance of the structure, especially in the structural problem. Using the finest mesh, however, increases the computational cost. Furthermore, high-resolution structures are produced that can not be realized in the lattice structure after homogenization.

It is also clear that optimizing for phase change can provide some benefits. However, this benefit comes at a high computational cost that could severely limit its usability to general uses. In the future, more work needs to be done to simplify, accelerate and optimize the performance of this module to make it scalable to 3D domains.

5 Multi-Functional Optimization

With the single-functional optimization methods working, they can now be combined into a multi-functional optimization. To achieve this, the two separate functions need to be combined into a single function where an optimum can be determined. The weight method has been discussed in chapter 2.4 and has been utilized for this thesis.

Another method that could be used is the constraint method, where one function acts as a constraint on the other. However, the problem with this method is that the resulting optimal structure will satisfy one constraint, but not the other. The trade-off between the two may be too strict on the unconstrained function. Figure 5.1 shows that the constraint method can find points on the Pareto front, but they do not have a minimum error from an unobtainable utopia point. It is still possible to iterate towards the true minimum, but the weight was chosen for its elegance.

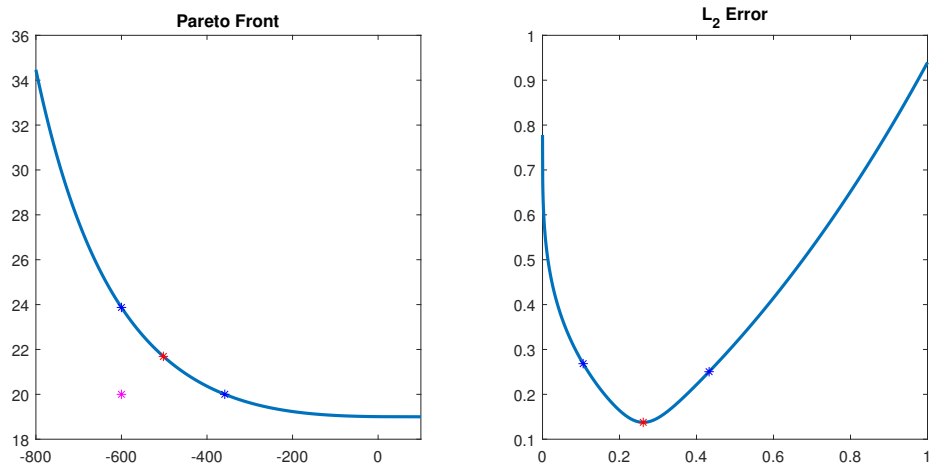


Figure 5.1: Pareto front showing points obtained from constraint method (blue) and the real optimum (red) given an unobtainable utopia point (magenta).

5.1 Pareto Front Generation

As was mentioned, in this thesis, the weight method is used for generating Pareto optimal points. The sum of the weights should add to one [**?**]. The weight method converts the multiple objective functions into a single equation that can then be optimized using techniques already discussed in chapter 2.4. Equation 5.1 shows the single objective function applied to the normalized thermal (c_t^*) and structural (c_s^*) compliance.

$$J(w,\rho) = wc_t^*(\rho) + (1-w)c_s^*(\rho) \tag{5.1}$$

To obtain reasonable results, the functions are normalized so that the compliance of each is of a similar order. This also ensures the gradients of each separate objective function are of a similar order. For the sensitivity analysis, equation 5.1 is differentiated w.r.t the relative density, ρ, as shown in equation 5.2.

$$\frac{\partial J}{\partial \rho} = w\frac{\partial c_t^*}{\partial \rho} + (1-w)\frac{\partial c_s^*}{\partial \rho} \tag{5.2}$$

With that, a Pareto front can be generated by varying the weight between 0 and 1, where the extremes would return to the single-function objective. Figure 5.2 shows the Pareto front generated for setup shown in 4.1. In this simulation, conductivity and stiffness were maximized for a given weight.

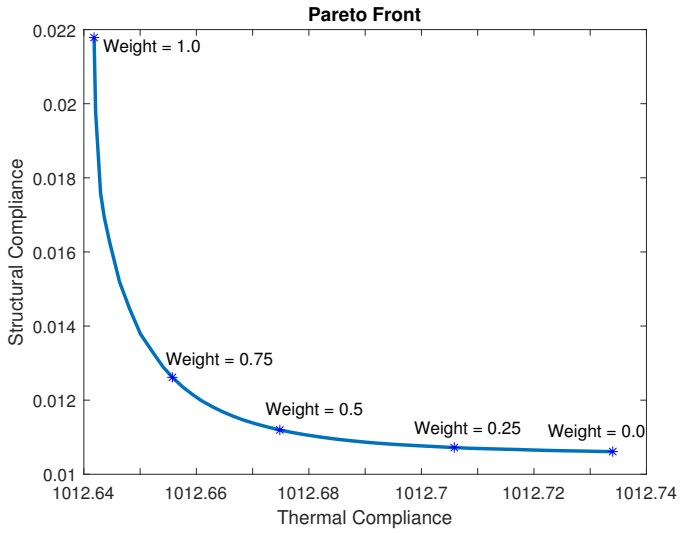
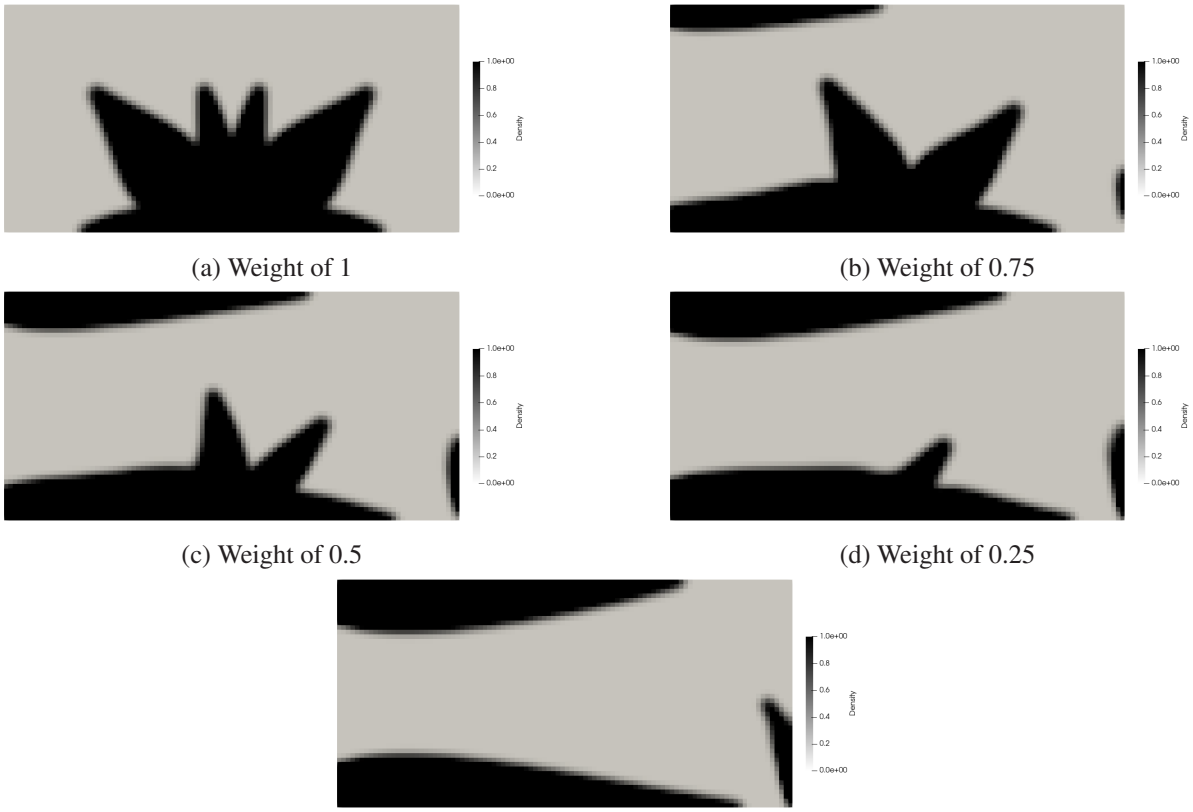


Figure 5.2: Pareto Front for Given Domain

Figure 5.3 shows the gradual transition from a thermal optimum to a structural optimum. The porosity range for these structures was set to 5% to 20%. With this range, as has already been discussed, the structures can have *'disconnected'* regions, and a lack of branches and struts. These results are valid, as has already been shown in chapter 4.



(a) Weight of 1

(b) Weight of 0.75

(c) Weight of 0.5

(d) Weight of 0.25

(e) Weight of 0

Figure 5.3: The gradual progression from thermal to structural optimum obtained from the Pareto front generation

5.2 Determining Optimum Trade-off

While the Pareto-front can be generated for a given number of weights, it is a time-consuming process that is not guaranteed to find a true optimum. Instead, a combination of methods can be used to get as close to a minimum as possible. It is important to realize that the compliance of each problem is not known without first performing a topology optimization. The shape of the Pareto front is also not known without constructing it as discussed previously. Instead, a strategy to iteratively find an optimum has been proposed in this thesis.

Assuming the Pareto-front is convex, there will be one point of minimal error. Therefore, the rate of convergence will depend on how accurate the initial guess is. To even begin to have an idea of an initial guess, the bounds of the Pareto front should be known. It is unlikely they can be determined from experience because depending on the forces, heat flux, and shape of the domain, the bounds of the Pareto front could be anything. Instead, the bounds are first determined by optimizing the single-functional problems, as has been done by Pejman and Najafi, [**?**]. This gives the range of compliance for each problem, which also allows the single-functional problems to be normalized.

To make an educated initial guess of the optimal point, the utopia point is projected onto the line segment that can be constructed from the bounding points. Due to the shape of the Pareto front, this guess is usually quite inaccurate, but at least a rough estimate is now possible. Figure 5.4 illustrates the bounding points being determined, the line segment being constructed, and the projected utopia point.

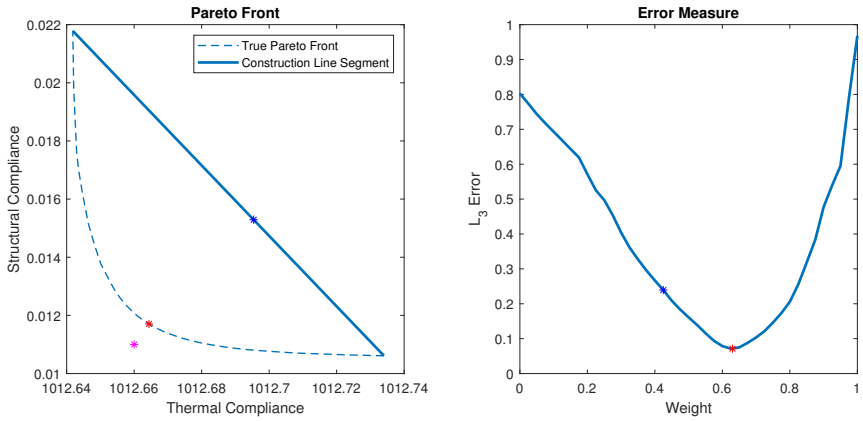


Figure 5.4: Initial guess (blue point) based on projecting the utopia point (magenta) onto the construction line (solid blue)

To iterate towards the true optimum, the quadratic fit method, described by Kochenderfer, et al., [**?**] has been utilized. This method is a gradient-free bracketing method that uses information from three points to construct a parabola and then finds its minimum.

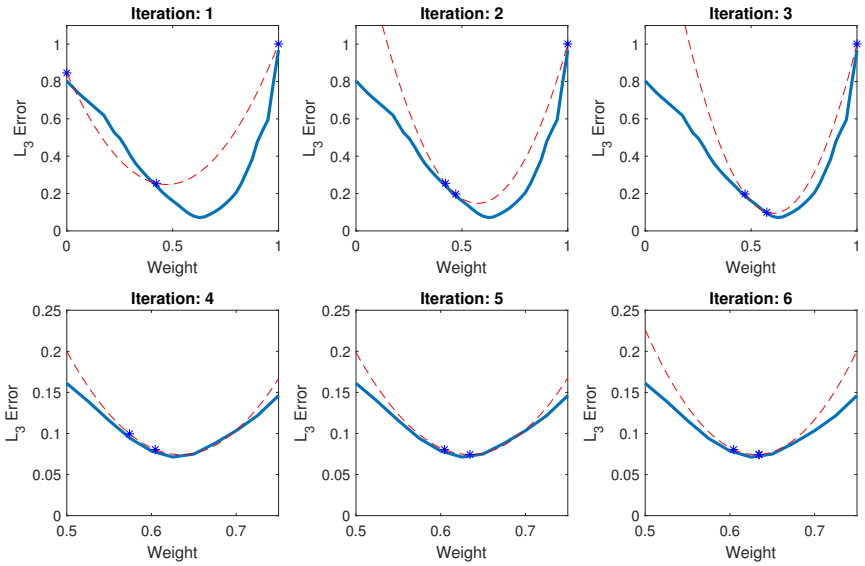


Figure 5.5: Quadratic fit convergence showing the real error norm (solid blue) and the fitted parabolas (dashed red) constructed by three points (red star)

It turns out that generally the error measure very roughly resembles a parabola, so this method can find the true optimal point, given a reasonable tolerance, in very few iterations. This method works by updating the bounds of the parabola each iteration, and will eventually collapse around the optimal point. Figure 5.5 shows how this method converges toward the optimal point. The convergence plot shows that it is already near the optimal point after three iterations, and after five iterations the optimal point is already determined within a low tolerance.

Limitations

This method does have some drawbacks. When the optimal point is, or is near the bounding points, a couple of problems can arise. Figure 5.6 illustrates when this problem can arise. The first problem is that the projection method that is used means the initial guess may be outside the allowable weight range. To solve this issue, the initial guess can be limited to a range such as $w_{\text{guess}} \in [0.1, 0.9]$. The quadratic fit method can then continue as normal.

The second issue with having the optimal point near the bounding points is that the quadratic fit method might find a minimum that is outside the allowable weight range. To overcome this issue, a bisection is performed instead to update the central point in the parabola. This can be repeated until the method converges to the true optimum, or until the quadratic fit method finds a valid optimal point. Note that the weight range can not be limited for this case, because the optimum could be the bounding point.

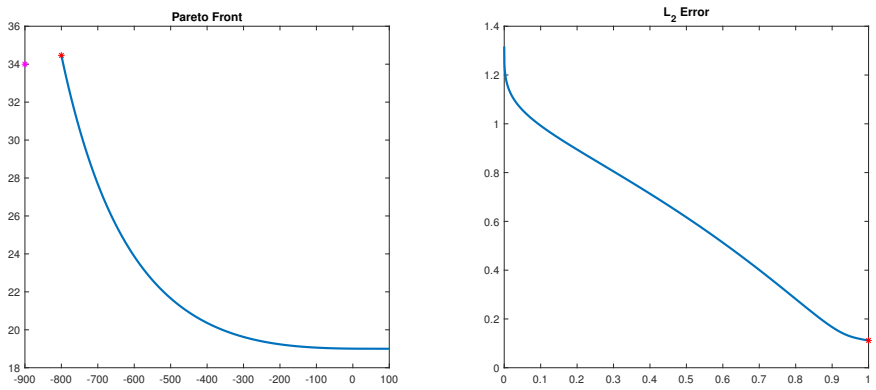


Figure 5.6: Limitation where the bounding point is the optimal point

Another limitation is that many local minima could be present when the function is concave. In such a situation, a more accurate initial guess is required to ensure it converges to the correct minimum. Through some experimentation, however, the Pareto front is usually convex. This is because the two functions tend to contribute to each other, rather than compete.

5.3 Maximizing Thermal Conductivity and Stiffness

The algorithms developed in chapter 4 were for the minimization of thermal and structural compliance, as well as the minimization of phase-change compliance. This section discusses results obtained using the minimization of thermal and structural compliance.

Test Case 1

To illustrate the effectiveness of the selected method, two test cases have been produced for the same Pareto front. The first utopia point was set to $U(1012.66, 0.011)$. The bounds of the Pareto front a then determined, and the compliances are normalised. The optimal point is then iteratively determined. This case converges to a reasonable tolerance in three iterations.

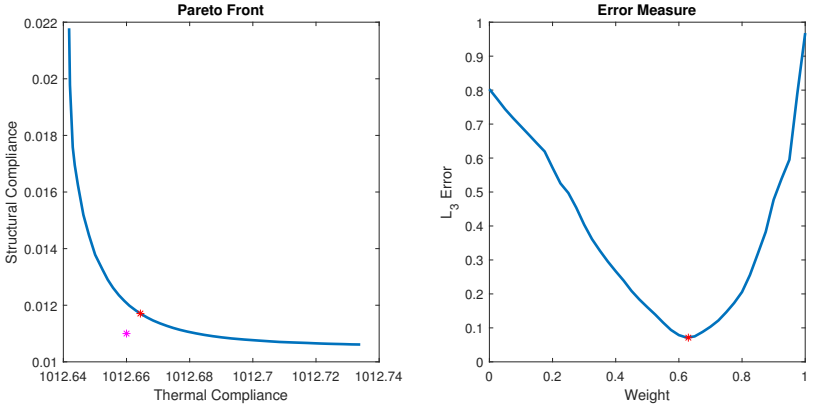


Figure 5.7: Optimum obtained (red) for utopia point $U(1012.66, 0.011)$ (magenta)

Figure 5.8 shows the structure that was generated. Compared to figure 5.3, this structure lies somewhere between structures b and c as is expected.

Figure 5.8: Pareto-optimal structure obtained for utopia point $U(1012.66, 0.011)$

Test Case 2

In this test case, the utopia point was set to $U(1012.64, 0.014)$. This case has a much sharper error norm but still converges to the optimal point in very few iterations. The optimal point is determined with a coarse tolerance in only six iterations, and to a high resolution in nine.

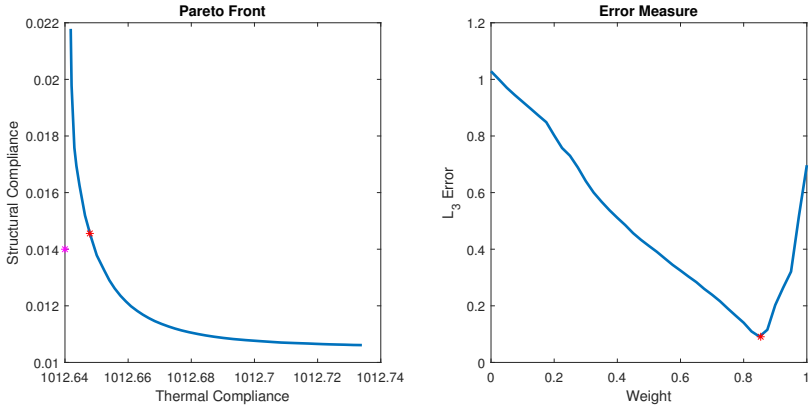


Figure 5.9: Optimum obtained (red) for utopia point $U(1012.64, 0.014)$ (magenta)

Figure 5.10 shows the structure generated. This structure lies between structures a and b which is expected.

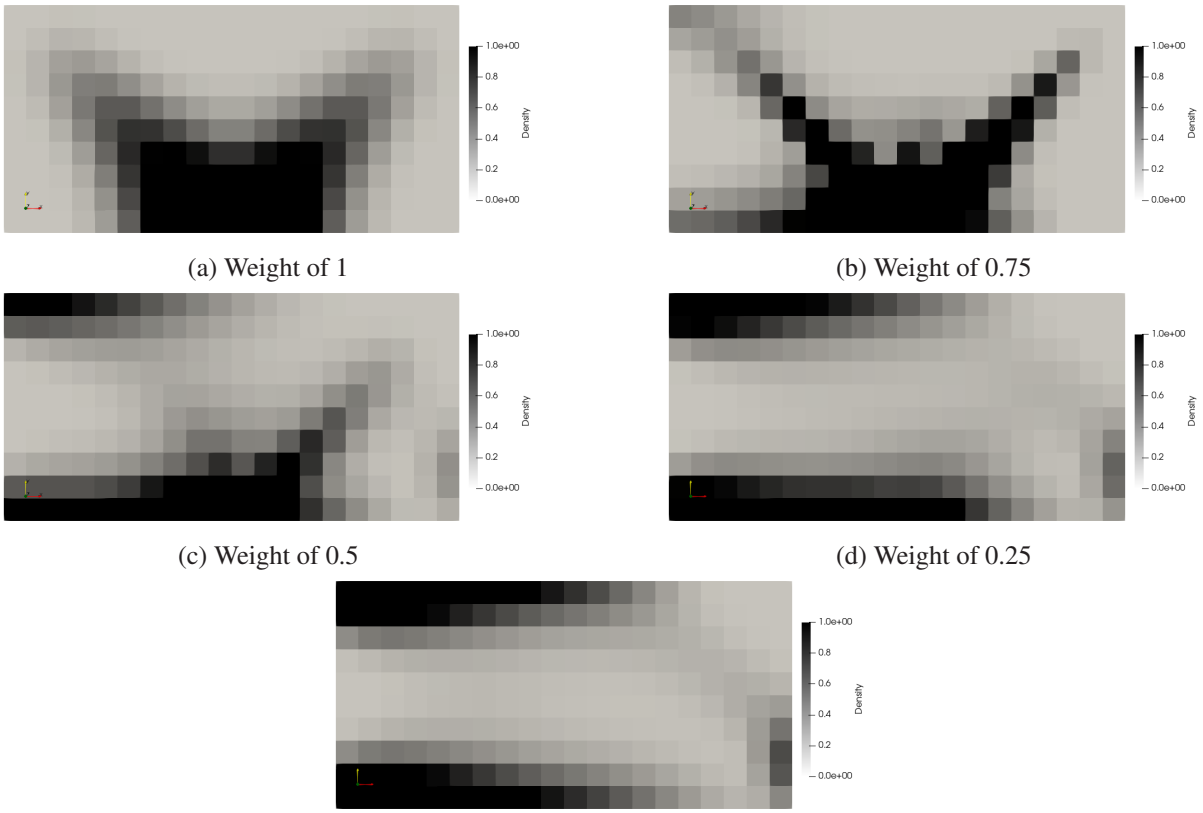
Figure 5.10: Pareto-optimal structure obtained for utopia point $U(1012.64, 0.014)$

The two test cases that have been used for the simple domain show that an optimum can be found. Both optimal structures were found in only a few iterations. However, to ensure generability, more test cases have been produced with different boundary conditions in chapter 6.

5.4 Minimising Temperature and Maximising Stiffness

This section discusses results obtained where the phase-change compliance and structural compliance were objective functions. This section expands on observations made in chapter 4 where the phase change compliance optimization provided some benefits. Unfortunately, to generate a Pareto front in a reasonable amount of time, the coarsest mesh had to be used. This makes it difficult to compare results with the previous section, but the overall structure is still visible.

Figure 5.11 shows some of the intermediate structures that have been generated using the Pareto front generation tool. These structures were optimized until the time the domain fully melted, as this provided the most benefit as shown in figure 4.19. The progression from thermal to structural optimum is clear. Because of the chosen end time, the thermal optimum has two branches that fill much of the domain.



(a) Weight of 1

(b) Weight of 0.75

(c) Weight of 0.5

(d) Weight of 0.25

(e) Weight of 0

Figure 5.11: The gradual progression from phase-change to structural optimum from the Pareto front generation

To compare the results to the conductivity optimal structures, a new set of structures was generated for the former case, and the phase change results have been plotted in figure 5.12. This plot shows different weight ratios. The plot shows that the weight has a much less linear relationship

with the phase-change optimizer than with the conductivity optimizer. These results shown in the plot are therefore not directly relatable, but they do show that the limits in optimums are comparable.

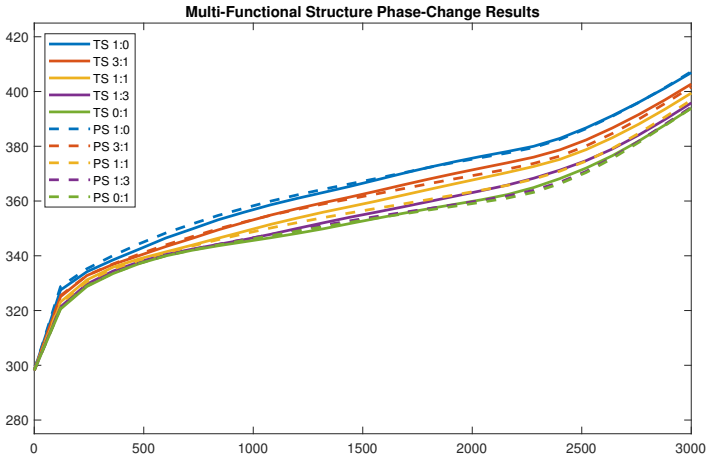


Figure 5.12: Phase change results showing the maximum temperature in the domain for the maximization of thermal conductivity and stiffness (TS/solid lines), and optimization of phase change compliance and stiffness (PS/dashed lines)

Test Case

Finally, a Pareto optimal structure has been generated for the utopia point $U(212500, 0.012)$. As has been discussed, first the bounds of the Pareto front are determined, the compliances are normalized, and the optimal point is iteratively determined. The algorithm returned in six iterations but with relatively poor accuracy.

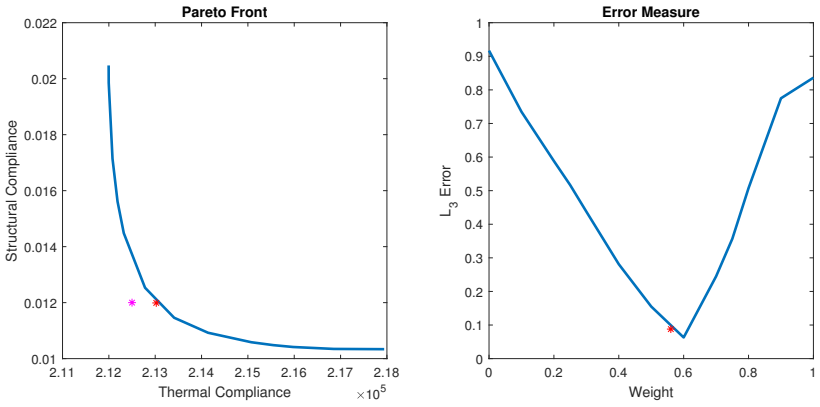


Figure 5.13: Optimum obtained (red) for utopia point $U(212500, 0.012)$ (magenta)

Figure 5.14 shows the structure that is obtained. It resembles a structure between figure 5.11 b and c. This is expected because the optimal weight is found to be around $w = 0.56$.

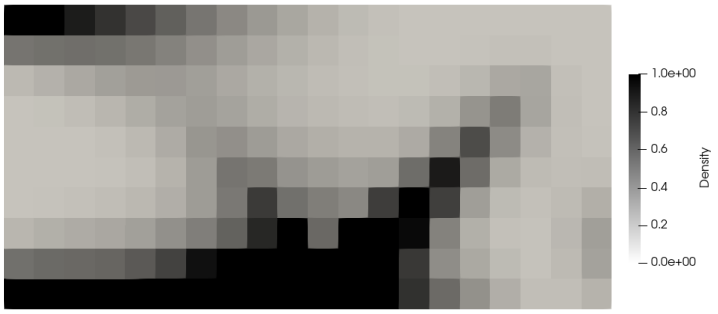


Figure 5.14: Pareto-optimal structure obtained for utopia point $U(212500, 0.012)$

5.5 Remarks

While the phase-change-structural optimization algorithm does provide a small benefit, the computational costs make it prohibitive to use more extensively. One of the major limiting factors is that the mesh needs to be quite coarse to obtain a structure in a reasonable amount of time. As the benefit is so negligible, in the following chapter, only structures obtained for the maximization of thermal conductivity and stiffness will be generated.

As was mentioned in this thesis, de-homogenization was not a major focus of this thesis because the types of cells were already defined. To illustrate a basic de-homogenization of the structure, figure 5.15. This structure is simple to generate because the positions and radius of each unit cell are determined by the mesh. This figure also helps to illustrate that the empty regions in the relative density plot are actually populated by high-porosity unit cells.

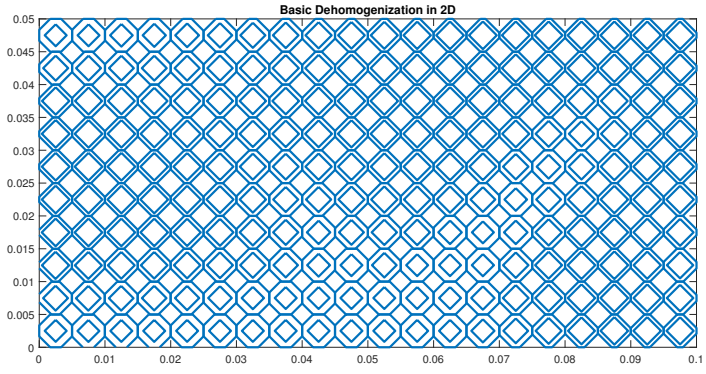


Figure 5.15: Basic de-homogenization of structure shown in figure 5.14

6 Results

Three further test cases have been produced to ensure the solver is general enough to solve a variety of different problems. Generally, the domain for spacecraft systems will be rectangular, so the same overall domain has been used. Just the position of the boundary conditions has been altered.

Each test case has a generated Pareto front for thermal and structural compliance. A Pareto optimal point is then found for each test case and is compared to the Pareto front to ensure it is reasonable.

6.1 Case One

Test case one has been designed to attempt to have two competing objective functions. The idea is that material needs to be distributed to the top and bottom of the domain which should take away from each objective. Figure 6.1 shows the domain with the two boundary conditions.

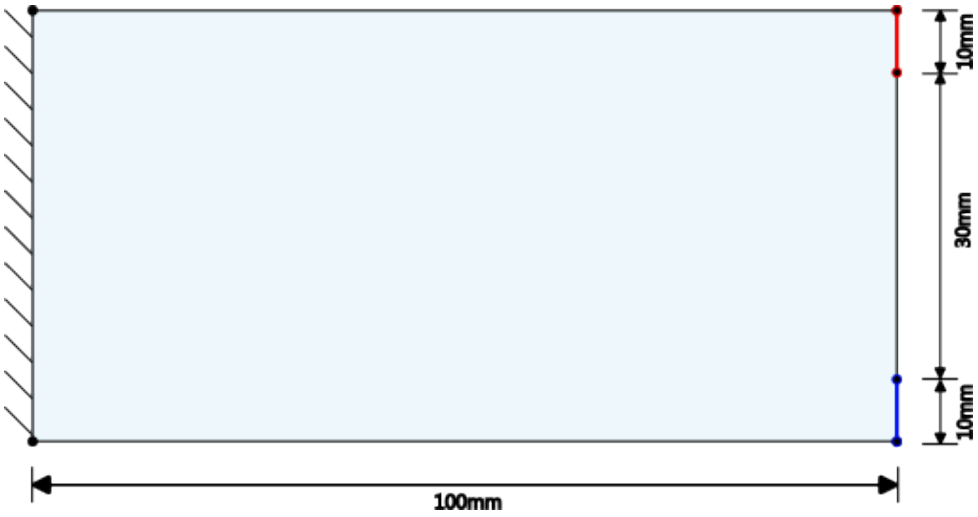


Figure 6.1: Test Case One Domain with a thermal load (red) and structural load (blue)

Figure 6.2 shows the obtained Pareto front for this setup. Despite the fact that the two boundary conditions were intended to be competing, it turns out they contribute to each other, as is observed in the Pareto front for the weights between $w = 0.25$ to $w = 0.75$ are close to each other.

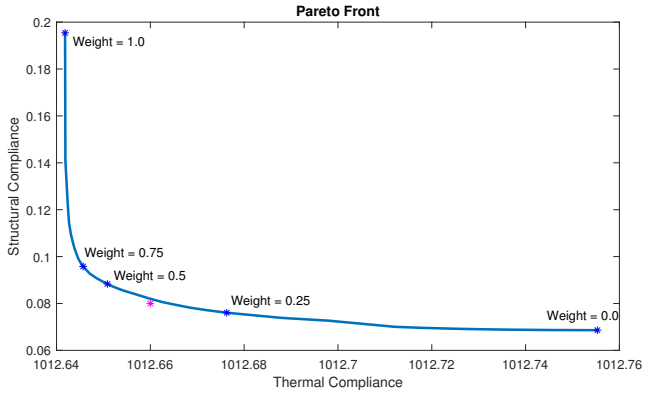


Figure 6.2: Pareto front for test case one showing the chosen utopia point (magenta)

Figure 6.3 shows some optimal structures that were recorded. The structure optimized for a weight of $w = 0.5$ shows that although the boundary conditions are not aligned, the generated structure produces a shape that allows the two boundary conditions to assist each other in minimizing the total objective function.

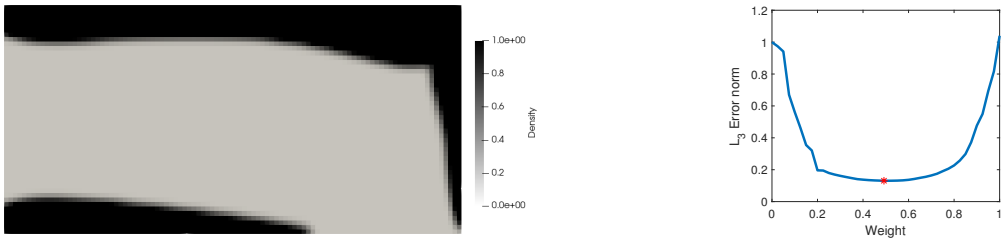
(a) Optimized structure for $w = 0.0$

(b) Optimized structure for $w = 0.5$

(c) Optimized structure for $w = 1.0$

Figure 6.3: Optimized structures for test case one

Figure 6.4 shows the Pareto optimal point determined for the utopia point $U(312.6, 0.01)$. Under these boundary conditions, the developed algorithm had no issues converging to the selected utopia point, even though the error norm plot has quite a wide well. The optimal point was found in just two iterations due to the initial guess being so accurate.



(a) Pareto optimal structure

(b) Error norm plot

Figure 6.4: Optimal structure obtained for utopia point $U(312.6, 0.01)$ (left) as well as the error norm plot (right) produced for verification

6.2 Case Two

Test case two has been designed to test different boundary conditions. The bottom left node is fixed, while the bottom right node can slide in the x direction. A large distributed load across the top of the domain is also used. The thermal boundary condition is placed along the bottom in the center, meaning these boundary conditions should contribute to each other for the overall goal function.

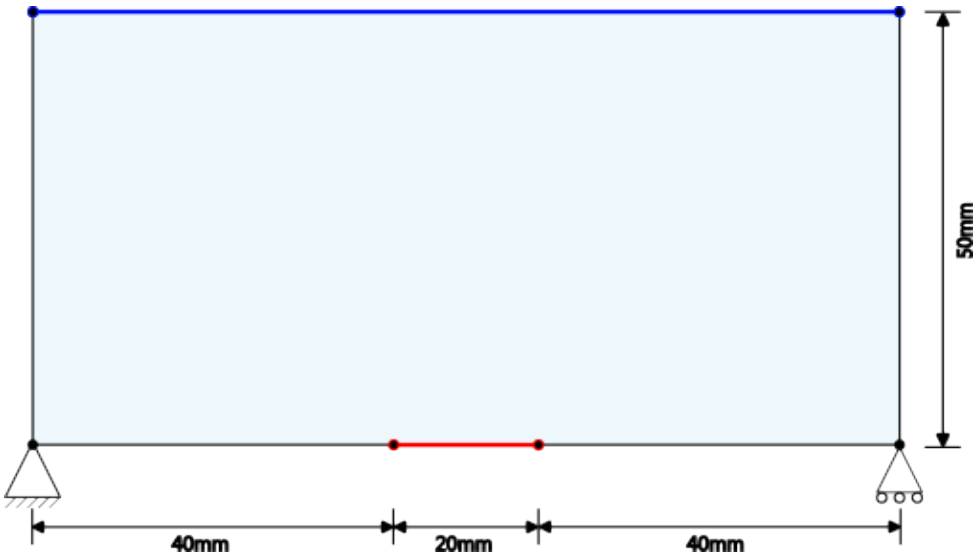


Figure 6.5: Test Case Two Domain

Figure 6.6 shows the Pareto front obtained for the boundary conditions of this case. It turns out that the boundary conditions partially compete with each other, unlike what was hypothesized. This occurs because as the thermal optimum develops, it removes a significant amount of material from where it is needed for the structural optimum.

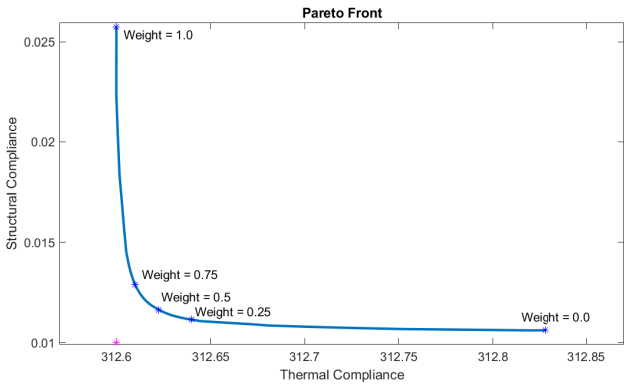


Figure 6.6: Pareto front for test case two showing the chosen utopia point (magenta)

Figure 6.7 also shows some optimal structures that were generated. They show that the thermal optimum essentially grows from the structural optimum. This backs up the idea that the two boundary conditions contribute to each other.

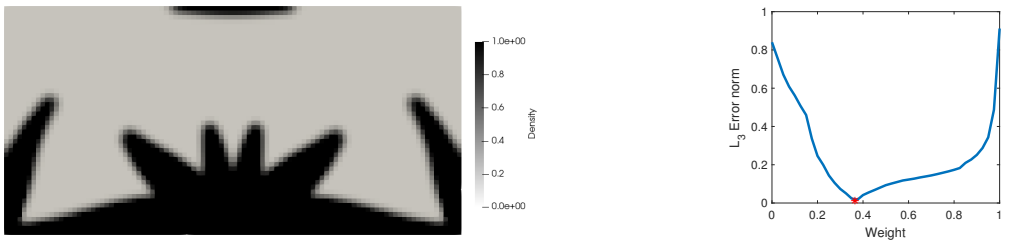
(a) Optimized structure for $w = 0.0$

(b) Optimized structure for $w = 0.5$

(c) Optimized structure for $w = 1.0$

Figure 6.7: Optimized structures for test case two

Figure 6.8 shows the optimized structure and error norm plot. Again, the minimum error is found quite easily due to a sharpening towards the optimum. Even though these are simple boundary conditions, this result in particular shows that error and Pareto front aren't always going to produce results that are expected. This test case also shows that the boundary conditions that should contribute to each other can end up competing when they are too far apart. Too much material is required by each separate function which results in one function, in this case, the thermal function, dominating the kind of structure that can form. The optimum was determined within a good tolerance in five iterations but fully converged in nine.



(a) Pareto optimal structure

(b) Error norm plot

Figure 6.8: Optimal structure obtained for utopia point $U(312.6, 0.01)$ (left) as well as the error norm plot (right) produced for verification

6.3 Case Three

Test case three has been designed as an extension to test case two to ensure that expected results are obtained for asymmetric boundary conditions. Based on the results from test cases one and two, it is expected that the boundary conditions contribute to each other than in case two. This is because they already partially contribute, but in this case, the material can be distributed closer to each boundary condition so that neither function can be dominant.

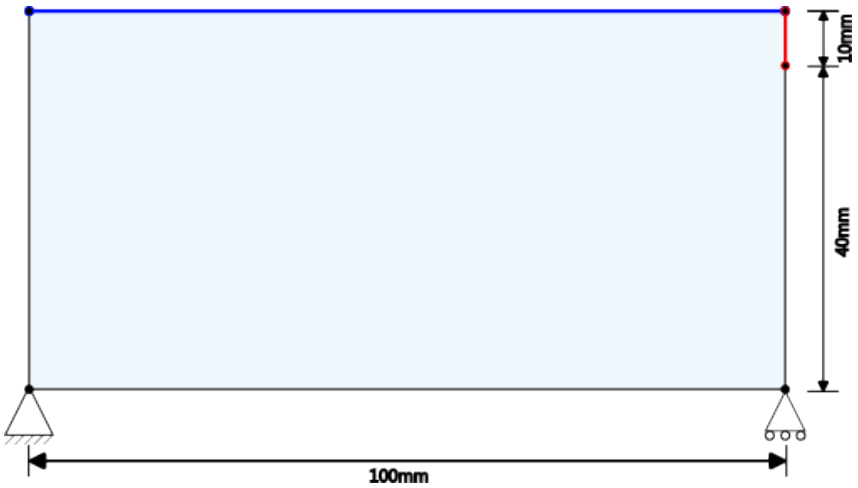


Figure 6.9: Test Case Three Domain

Figure 6.10 shows the Pareto front obtained for this final test case. Again the weights between $w = 0.25$ and $w = 0.75$ are close to each other because, as mentioned in this section's opening, the two boundary conditions contribute to each other. In fact, these two boundary conditions contribute so well with each other that between the weights $w = 0.0$ to $w = 0.25$ and $w = 0.75$ to $w = 1.0$, weakly Pareto optimal points are essentially found.

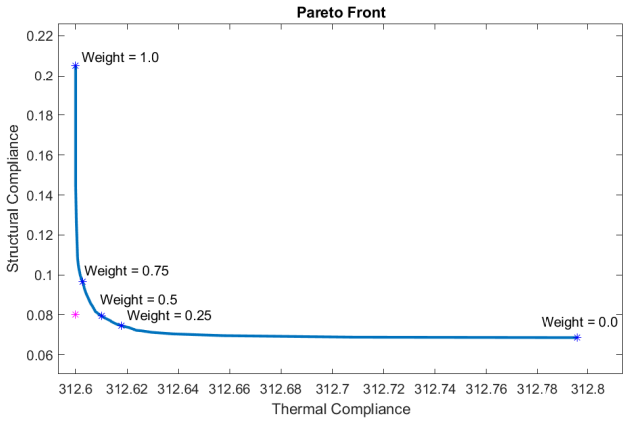


Figure 6.10: Pareto front for test case three showing the chosen utopia point (magenta)

Figure 6.11 shows how the two boundary conditions contribute to each other, especially on the right-hand side of the domain. They contribute because the thermal boundary condition allows the material to remain where it is required by the structural optimum on the right of the domain. This contrasts with case 2 which uniformly removed material from the top of the domain, limiting the structural compliance.

Figure 6.12 shows the optimized structure and error norm plot. In this case, the minimum is close, but not exactly the minimum. The initial guess was good, but the algorithm converged

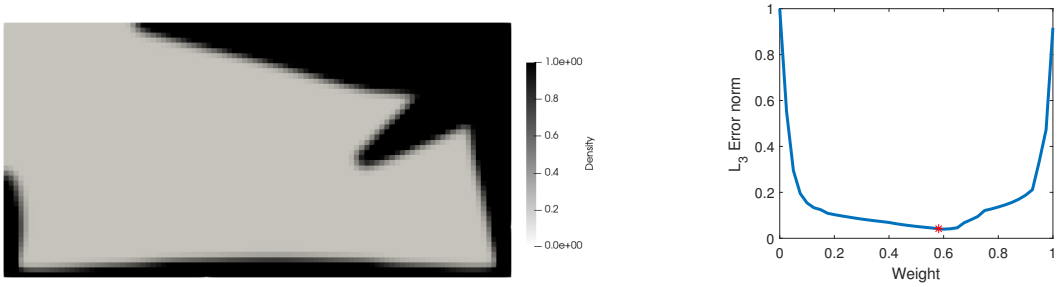
(a) Optimized structure for $w = 0.0$

(b) Optimized structure for $w = 0.5$

(c) Optimized structure for $w = 1.0$

Figure 6.11: Optmized structures for test case three

too slowly. The error is still within a reasonable tolerance, but improvements should be made to ensure a true global minimum can be found.



(a) Pareto optimal structure

(b) Error norm plot

Figure 6.12: Optimal structure obtained for utopia point $U(312.6, 0.01)$ (left) as well as the error norm plot (right) produced for verification

6.4 Remarks

Of the boundary conditions that were devised, all produced a convex Pareto-front. As the algorithm is sensitive to the initial guess, this information can be useful when trying to determine the initial weight. Instead of projecting the point onto a line segment, the point could instead be projected onto the reciprocal function $y = k/x$, where k can be used to control how sharp the shape of the curve is. The most generic form of this function is given in equation 6.1.

$$y = \frac{k}{x+a} + b \tag{6.1}$$

The variables a and b are used to shift the function. They can be determined by solving a pair of simultaneous equations. If y is the normalized compliance of the first function, and x for the second, then a system of simultaneous can be determined as shown in equation 6.2.

$$\begin{aligned} \frac{k}{a} + b &= 1 \\ \frac{k}{1+a} + b &= 0 \end{aligned} \tag{6.2}$$

By rearranging the second equation, an equation for b can be found which can then be used to determine a using the quadratic formula, as shown in equation 6.3.

$$\begin{aligned} b &= \frac{k}{1+a} \\ a &= \frac{-1+\sqrt{1+4k}}{2} \end{aligned} \tag{6.3}$$

With that, the utopia point, U, can be projected onto the curve by defining the goal function shown in equation 6.4. The value x can be determined using a gradient descent or similar method.

$$\text{argmin}_{x\in[0,1]} \| (y,x) - U \|_p \tag{6.4}$$

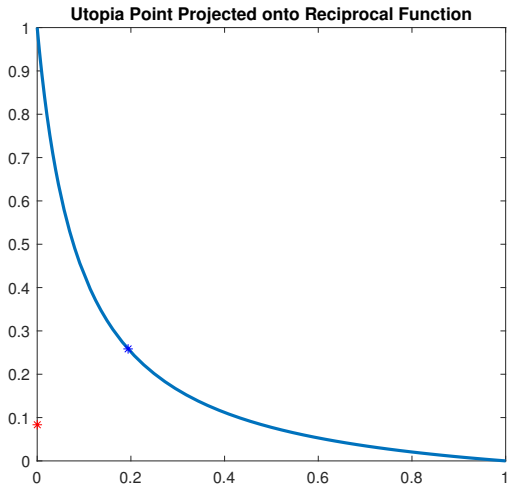


Figure 6.13: Projection (blue) of utopia point (magenta) onto the reciprocal function (blue line)

Once x has been determined, the distance along the curve from $x = 0$ needs to be determined. It is this distance that corresponds to the weight. This can be found by solving equation 6.5. This method assumes the weights to be even distributed along the curve, which is generally not

the case. This approach could be modified with a weighting function to reduce the calculated *distance* along the curve near the endpoints.

$$S(x) = \frac{1}{S(1)} \int_0^x \sqrt{(1 + \left(\frac{dy}{dx}\right)^2)}\, dt \tag{6.5}$$

To resolve the issue discussed in case 3, the quadratic fit method could be accelerated by considering gradient information obtained by previous iterations. A gradient descent sub-step could be performed to improve convergence in shallow wells. However, certain checks would need to be performed to ensure minimal overshooting occurs.

7 Conclusion

7.1 Conclusion

In this thesis, a methodology to produce multi-functional topology-optimized structures has been developed and investigated. Structural, thermal and phase change considerations have been incorporated, and the results they produce have been discussed. While this thesis focussed on their use in spacecraft structures, the methodology can also be applied to Earth-bound systems.

The thesis began by looking into the theoretical background of all the ingredients required. Heat transfer and the Stefan problem were the roots of what would be built on. Structural mechanics was also the most important topic that was required to produce traditional topology-optimized structures. Finally, optimization methods were discussed which would be used throughout this thesis. While automatic differentiation was not as useful as hoped, it was implemented in many stages for computing certain gradients efficiently.

The state of the art was then researched which would provide the groundwork for what this thesis would build on. Homogenized properties for the cellular structures were provided that were used in all topology optimizations. Latent heat energy storage materials were found, and paraffin was selected for this thesis as it was a simple material to implement that could be widely varied for specific use cases. Topology optimization algorithms were found, most importantly the structural and thermal algorithms. Multifunctional topology optimization methods were found in the literature that could be adapted to this specific case.

Finally, a methodology for the structural-thermal and structural-phase-change optimization has been developed. The algorithms are so general that they can be applied to any material or any homogenized unit cell. Automatic differentiation, while not useful for solving the full script, was extremely useful and efficient in computing element matrices and simple program derivatives. Some test cases were also investigated to ensure different boundary conditions could be solved.

7.2 Discussion

While This thesis developed and investigated a methodology for multi-functional optimization, it did not explicitly consider the de-homogenization of the produced structure. The structures that are produced contain elements much smaller than a unit cell. This means that their reactive density needs to be somehow averaged over the volume of a unit cell in a de-homogenization process. This will naturally produce a different stiffness and maximum temperature.

This thesis was also focused on space-craft structures. This means the phase change simulation only considers conduction and no convection. Topology optimization has been successfully performed considering the Navier-Stokes equations [**?**], but this time cost is significant. In future work, phase change optimized structures should consider convection to be able to produce highly optimized structures for use on Earth. Such structures could be used for energy storage as a low-cost and efficient alternative to chemical batteries.

A Lattice Properties

The properties provided here have been obtained for the bcc and f_2cc,z unit cells. The properties have been obtained by Soika [**?**] and Bühring, et al., [**?**].

Where for the thermal properties:
ε is the cell porosity.
r is the strut radius.
h is the cell height.
R_i is the thermal resistance for different regions.
k_s is the thermal conductivity of the solid material.
k_f is the thermal conductivity of the phase change material.
k_i is the effective thermal conductivity in each basis direction.

Where for the structural properties:
E is Young's modulus of the lattice material.
E_i is the effective Young's modulus in each basis direction.
r is the strut radius.
h is the cell height.
ω is the cell strut incliniation.

The properties follow on the next page.

bcc Unit Cell

Thermal Properties

$$\Omega = \arctan\left(\frac{\tan(\omega)}{\sqrt{2}}\right)$$
$$\varepsilon = 1 - 2\tan^2(\Omega)\frac{r^2}{h^2}\left[\pi\frac{4}{\sin(\Omega)} - \frac{16r}{3h}\left(\frac{2.993}{\sin(\pi - 2\Omega)} + \frac{3.340}{\sin(\pi/2 - \Omega)}\right)\right]$$
$$V_4 = 2\frac{16}{3\sin(\pi - 2\Omega)}r^3 - 12\left(\sqrt{8} - \sqrt{6}\right)r^3$$

1, 2 Direction	3 Direction
$t_4 = \left(V_4 \sin(\Omega)\cos^2(\Omega)\right)^{1/3}$	$t_4 = \left(V_4 \sin^2(\Omega)\cos(\Omega)\right)^{1/3}$
$a = \sqrt{\frac{2}{\cos^2(\Omega)} + \frac{1}{\sin^2(\Omega)}}$	$a = \sqrt{\frac{1}{\cos^2(\Omega)} + \frac{2}{\sin^2(\Omega)}}$
$L_{\text{str}} = \frac{h}{2\sin(\Omega)} - at_4$	$L_{\text{str}} = \frac{h}{2\sin(\Omega)} - at_4$
$R_1 = R_3 = \frac{2\cos^2(\Omega)}{4.277\sin(\Omega)t_4 k_s}$	$R_1 = R_3 = \frac{2\sin^2(\Omega)}{1.903\cos(\Omega)t_4 k_s}$
$R_2 = \frac{L\text{str}}{\pi r^2 k_s}$	$R_2 = \frac{L\text{str}}{\pi r^2 k_s}$
$R_{\text{str}} = 2(R_1 + R_2 + R_3)$	$R_{\text{str}} = 2(R_1 + R_2 + R_3)$
$R_s = \frac{R_{\text{str}}}{4}$	$R_s = \frac{R_{\text{str}}}{4}$
$k_{1,2} = \varepsilon k_f + \frac{1}{hR_s}$	$k_{1,2} = \varepsilon k_f + \frac{2\tan^2(\Omega)}{hR_s}$

Structural Prorperties

$$E_{1,2} = \frac{2\sqrt{2}\pi E r^2}{h^2}\left[1 + 12\frac{r^2}{h^2}\left(\sin^2(\omega) + 1\right)\tan^2(\omega)\right] \times sin^2(\omega)\cos(\omega)$$
$$E_3 = \frac{8\pi E r^2}{h^2}\left[1 + 12\frac{r^2}{h^2}\cos^2(\omega)\right] \times \sin^3(\omega)\tan^2(\omega)$$

f_2cc,z Unit Cell

Thermal Properties

$$\varepsilon = 1 - \tan^2(\omega)\frac{r^2}{h^2}\left[\pi\left(1+\frac{4}{\sin(\omega)}\right) - \frac{16r}{3h}\left(\frac{2.993}{\sin(\pi-2\omega)} + \frac{3.667}{\sin(\pi/2-\omega)}\right)\right]$$
$$V_2 = \frac{16}{3\sin(\pi-2\omega)}r^3$$
$$V_4 = 2V_2 - 12r^3\left(\sqrt{8}-\sqrt{6}\right)$$

1,2 Direction	3 Direction
$t_2 = (V_2\sin(\omega)\cos^2(\omega))^{1/3}$	$t_2 = (V_2\sin^2(\omega)\cos(\omega))^{\frac{1}{3}}$
$t_4 = (V_4\sin(\omega)\cos^2(\omega))^{1/3}$	$t_4 = (V_4\sin^2(\omega)\cos(\omega))^{\frac{1}{3}}$
$a = \sqrt{\frac{2}{\cos^2(\omega)} + \frac{1}{\sin^2(\omega)}}$	$a = \sqrt{\frac{1}{\cos^2(\omega)} + \frac{2}{\sin^2(\omega)}}$
$L_{\text{str},1/2} = \frac{h}{2\sin(\omega)} - a\left(\frac{t_2+t_4}{2}\right)$	$L_{\text{str},1/2} = \frac{h}{2\sin(\omega)} - a\left(\frac{t_2+t_4}{2}\right)$
$R_1 = \frac{2\cos^2(\omega)}{1.732\sin(\omega)t_4k_s}$	$R_1 = \frac{2\sin^2(\omega)}{1.448\cos(\omega)t_4k_s}$
$R_2 = \frac{L_{\text{str}}}{\pi r^2 k_s}$	$R_2 = \frac{L_{\text{str}}}{\pi r^2 k_s}$
$R_3 = \frac{\cos^2(\omega)}{2.836\sin(\omega)t_2k_s}$	$R_3 = \frac{\sin^2(\omega)}{1.168\cos(\omega)t_2k_s}$
$R_{\text{str}} = 2(R_1+R_2+R_3)$	$R_{\text{str}} = 2(R_1+R_2+R_3)$
$R_s = \frac{R_{\text{str}}}{2}$	$R_s = \frac{\pi r^2 k_s}{h} + \frac{4}{R_{\text{str}}}$
$k_{1,2} = \varepsilon k_f + \frac{1}{hR_s}$	$k_{1,2} = \varepsilon k_f + \frac{1}{hR_s}$

Structural Properties

$$E_1 = \frac{2\pi E r^2}{h^2}\left[1 + 12\frac{r^2}{h^2}\sin^2(\omega)\tan^2(\omega)\right] \times \sin^3(\omega)$$
$$E_3 = \frac{\pi E r^2}{h^2}\left[1 + 4\sin^3(\omega)\left(1 + 12\frac{r^2}{h^2}\cos^2(\omega)\right)\right] \times \tan^2(\omega)$$

B Phase Change Sensitivity

Recall that the residual form to solve the phase change problem is given as shown below. This was derived and given by [**?**]

$$R(T^{n+1}) = f\Delta t + \mathbf{C}T^n - (L^{n+1} - L^n) - (\mathbf{C} + \mathbf{K}\Delta t)T^{n+1}$$

Linear Gradient

When the system of equations is linear, the gradient can be computed as follows. Equation B.1 shows the simplified residual equation when the latent heat vectors are equal. The equation essentially becomes a linear system of equations that can be solved.

$$\begin{aligned} T^{n+1} &= (\mathbf{C} + \mathbf{K}\Delta t)^{-1}(f\Delta t + \mathbf{C}T^n) \\ T^{n+1} &= \mathbf{A}^{-1}b \end{aligned} \tag{B.1}$$

The gradient of this temperature vector can then be determined as follows in equation B.2. One thing to note from this derivative is that the derivative of the previous temperature vector is required. This means that the derivatives need to be tracked throughout each iteration. This has the effect of accumulating the transient effects throughout the entire simulation.

$$\begin{aligned} \frac{\partial T^{n+1}}{\partial \rho} &= \frac{\partial}{\partial \rho}\mathbf{A}^{-1}b \\ &= \frac{\partial \mathbf{A}^{-1}}{\partial \rho}b + \mathbf{A}^{-1}\frac{\partial b}{\partial \rho} \end{aligned} \tag{B.2a}$$

$$\begin{aligned} \frac{\partial \mathbf{A}^{-1}}{\partial \rho_i}b &= -\mathbf{A}^{-1}\frac{\partial \mathbf{A}}{\partial \rho_i}\mathbf{A}^{-1}b \\ &= -\mathbf{A}^{-1}\frac{\partial}{\partial \rho_i}(\mathbf{C} + \mathbf{K}\Delta t)\mathbf{A}^{-1}b \\ &= -\mathbf{A}^{-1}\left(\frac{\partial \mathbf{C}}{\partial \rho_i} + \Delta t\frac{\partial \mathbf{K}}{\partial \rho_i}\right)\mathbf{A}^{-1}b \end{aligned} \tag{B.2b}$$

$$\begin{aligned}\frac{\partial b}{\partial \rho} &= \frac{\partial}{\partial \rho}(f\Delta t + \mathbf{C}T^n) \\ &= \Delta t\frac{\partial f}{\partial \rho} + \frac{\partial}{\partial \rho}\mathbf{C}T^n \\ &= \frac{\partial \mathbf{C}}{\partial \rho}T^n + \mathbf{C}\frac{\partial T^n}{\partial \rho}\end{aligned} \tag{B.2c}$$

Non-Linear Gradient

The computation for the non-linear gradient needs to consider the latent heat vectors, as these are no longer equal and have a significant effect on the gradient. Due to the non-linearity, the temperature is iteratively determined using a Newton iteration, where the Jacobian, $\mathbf{J}$, of equation B is required. Using these, the temperature of the next time step can be determined.

$$\begin{aligned}T_{i+1}^{n+1} &= T_i^{n+1} + (\Delta T)_i \\ T_{i+1}^{n+1} &= T_0^{n+1} + \sum_{i=0}^{n}(\Delta T)_i \\ \text{Where: } T_0^{n+1} &= T^n\end{aligned} \tag{B.3a}$$

$$(\Delta T)_i = [\mathbf{J}(T_i^{n+1})]^{-1}R(T_i^{n+1}) \tag{B.3b}$$

The gradient of the temperature vector can then be determined as shown in equation B.4. Some complications arise with the latent heat vector and matrix as the gradient of the previous temperature vector is required. These were briefly discussed in chapter 4.

$$\frac{\partial T_{i+1}^{n+1}}{\partial \rho} = \frac{\partial T^n}{\partial \rho} + \frac{\partial}{\partial \rho}(\sum_{i=0}^{n}(\Delta T)_i) \tag{B.4a}$$

$$\begin{aligned}\frac{\partial}{\partial \rho}(\Delta T)_i &= \frac{\partial}{\partial \rho}\mathbf{J}^{-1}R \\ &= \frac{\partial \mathbf{J}^{-1}}{\partial \rho}R + \mathbf{J}^{-1}\frac{\partial R}{\partial \rho}\end{aligned} \tag{B.4b}$$

$$\begin{aligned}
\frac{\partial \mathbf{J^{-1}}}{\partial \rho_e} R &= -\mathbf{J}^{-1} \frac{\partial \mathbf{J}}{\partial \rho_e} \mathbf{J}^{-1} R \\
&= -\mathbf{J}^{-1} \frac{\partial}{\partial \rho_e} \left(\mathbf{C} + \mathbf{K}\Delta t + \left.\frac{\partial L}{\partial T}\right|_i^{n+1} \right) \mathbf{J}^{-1} R \\
&= -\mathbf{J}^{-1} \left(\frac{\partial \mathbf{C}}{\partial \rho_e} + \Delta t \frac{\partial \mathbf{K}}{\partial \rho_e} + \frac{\partial}{\partial \rho_e} \left(\left.\frac{\partial L}{\partial T}\right|_i^{n+1} \right) \right) \mathbf{J}^{-1} R
\end{aligned} \tag{B.4c}$$

$$\begin{aligned}
\frac{\partial R}{\partial \rho} &= \frac{\partial}{\partial \rho} (f\Delta t + \mathbf{C}T^n - (L^{n+1} - L^n) - \mathbf{A}T_i^{n+1}) \\
&= \Delta t \frac{\partial f}{\partial \rho} + \frac{\partial}{\partial \rho} \mathbf{C}T^n - \frac{\partial}{\partial \rho} (L^{n+1} - L^n) - \frac{\partial}{\partial \rho} (\mathbf{A}T_i^{n+1}) \\
&= \frac{\partial \mathbf{C}}{\partial \rho} T^n + \mathbf{C} \frac{\partial T^n}{\partial \rho} - \frac{\partial L^{n+1}}{\partial \rho} + \frac{\partial L^n}{\partial \rho} - \frac{\partial \mathbf{A}}{\partial \rho} T_i^{n+1} - \mathbf{A} \frac{\partial T_i^{n+1}}{\partial \rho}
\end{aligned} \tag{B.4d}$$

C Source Code

Main Files

The following files are the main files.

main.jl

This is the main Julia file. This file acts as a .ini file that loads all the required data. It imports a mesh and preallocates some matrices that are required for the topology optimization functions.

```
#=
  Currently this file acts as an input file for the "optimiseFunctions.jl" file.
  1) First run this file to load all the include functions
  2) Run the optimseFunctions.jl file
  3) Call the functions in the optimseFunctions.jl file from the terminal
=#

include("fileio.jl")
include("assembly.jl")
include("topologyOptimisation.jl")

using Printf
using Plots

# Setup
quad = getQuadratureRules(Quad1);        # Get quadrature rules for 1st order quadrelaterals
shape = getShapeAndDerivatives(Quad1, quad);  # Get shape functions for 1st order quad

# Import mesh. (ncon = node connectivity, pos = nodal positions, boundaries = maps boundary name to boundary elements,
        boundaryNodes = maps boundary name to boundary nodes)
meshpath = "/Users/spiacqua/Downloads/Thesis-main/res/meshes/CombinedDomain.msh";
savepath = "/Users/spiacqua/Downloads/Thesis-main/res/results/";
ncon, pos, boundaries, boundaryNodes = importMsh(meshpath);

# Mesh stats
nElements = size(ncon[boundaries["Domain"]])[1];
nNodes = size(pos)[1];

# Initialise domain
K_T,f_T = getSparsityPatternHeat(ncon);   # Gets sparcity pattern of global matrix and forcing array    Heat problem
Ke_T = zeros(4,4,nElements);         # Initialise element matrices (used in sensitivity analysis)  Heat problem
K_S,f_S = getSparsityPatternSolid(ncon);  # Gets sparcity pattern of global matrix and forcing array    Solid problem
Ke_S = zeros(8,8,nElements);         # Initialise element matrices (used in sensitivity analysis)  Solid problem

HF = getWeightFactorMatrix(ncon[boundaries["Domain"]], pos, 1.5e-3);  # Weight factor matrix (filter matrix to filter
        sensitivities)
# neighbours = getNeighbours(ncon[boundaries["Domain"]]);          # Get element neighbours
elements = initElementMatrices(ncon[boundaries["Domain"]], pos, quad, shape); # Precompute the element matrices

celltype = "f2ccz";
```

phaseChange.jl

This file has the phase change methods for solving the phase change problem.

```
include("assembly.jl")
include("fileio.jl")
using Printf
using SparseArrays
using Ferrite
using ReverseDiff

## Main solve function
function solve(endtime, dt, initialTemp, reldense, savename, itrsPerSave = 1)
  temperature = initialTemp .* ones(nNodes);
  global timeElapsed = 0.0;
  saveCounter = 1;
    global p = 1.0; global useLattice = true;

  # Save initial conditions and init pvd
  if itrsPerSave > 0
    pvdpath = "res/results_pc/";
    while handleOverwrite(pvdpath * savename * ".pvd") == true
      println("You are about to overwrite an existing file. Would you like to continue? (y/n)");
      answer = readline();
      if answer == "y"
        break;
      elseif answer == "n"
        println("Enter a new save name");
        savename = readline();
      else
        println("Unrecognised key");
      end
    end
    savepath = pvdpath * savename * ".pvd"
    initExport2pvd(savepath, pos, ncon, timeElapsed, temperature, reldense);
  end


  # Assemble constant matrices
  K,_ = getSparsityPatternHeat(ncon[boundaries["Domain"]]);
  C,_ = getSparsityPatternHeat(ncon[boundaries["Domain"]]);
  F = Array{Float64}(undef, nNodes);
  globalAssemblyPC!(ncon[boundaries["Domain"]], pos, reldense, K,C,F, quad, shape);
  thermalBCs!(K, F);

  # Main transient iteration loop
  deltaTemp = zeros(nNodes);
  while timeElapsed < endtime
    prevTemp = temperature;
    temperature = solveTimestep(dt, temperature, deltaTemp, reldense, K,C,F);
    deltaTemp = temperature .- prevTemp;
    timeElapsed += dt;
    Printf.@printf "Time: %.2f, Temperature Range (w.r.t T_m): %.2f - %.2f\n" timeElapsed minimum(temperature)-tm maximum(
            temperature)-tm

    # Save timestep
    if saveCounter == itrsPerSave || (timeElapsed >= endtime && itrsPerSave > 0)
      export2pvd(savepath, pos, ncon, timeElapsed, temperature, reldense);
      saveCounter = 0;
    end

    saveCounter += 1;
  end

  return temperature, temperature' * F;
end


## Main solve with gradients
function solveWithGradient(endTime, dt, initialTemp, reldense)
  temperature = initialTemp .* ones(nNodes);
  global timeElapsed = 0.0;
  saveCounter = 1;
    global p = 1.0; global useLattice = true;

  K,_ = getSparsityPatternHeat(ncon[boundaries["Domain"]]);
  C,_ = getSparsityPatternHeat(ncon[boundaries["Domain"]]);
  F = Array{Float64}(undef, nNodes);
  globalAssemblyPC!(ncon[boundaries["Domain"]], pos, reldense, K,C,F, quad, shape);
  handleNeumannBCs!(ncon[boundaries["Heat"]], pos, 25000.0, F, 1);

  dKe = zeros(4,4, nElements);
  dCe = zeros(4,4, nElements);
  getDerivatives(dKe, dCe, reldense);

  timeElapsed = 0.0;
  dT = zeros(nNodes,nElements);
  temperature = initialTemp .* ones(nNodes);
  while timeElapsed < endTime
    temperature,dT = solveTimestepGradient(dt, temperature, reldense, K,C,F, dKe,dCe, dT);
```

```
    timeElapsed += dt;
    Printf.@printf "Time: %.2f, Temperature Range (w.r.t T_m): %.2f - %.2f\n" timeElapsed minimum(temperature)-tm maximum(
        temperature)-tm

    saveCounter += 1;
  end

  return temperature, temperature' * F, dT' * F;
end;

function getDerivatives(dKe, dCe, reldense)
  for e = 1:nElements
    elePoints = pos[ncon[boundaries["Domain"][e]],:];
    _,_,_, dK, dC = assembleElementdPC(elePoints, reldense[e], quad, shape)

    dKe[:,:,e] .= dK;
    dCe[:,:,e] .= dC;
  end
end;

function assembleElementdPC(element, reldense, quad, shape)
  Ke = zeros(typeof(reldense), 4,4);
  Ce = zeros(typeof(reldense), 4,4);
  dKe = zeros(typeof(reldense), 4,4);
  dCe = zeros(typeof(reldense), 4,4);
  fe = zeros(4);

  D = DiffResults.DiffResult(Matrix{Float64}(undef, 2,2), Matrix{Float64}(undef, 2,2));
  cp = DiffResults.DiffResult(0.0, [0.0]);
  rho = DiffResults.DiffResult(0.0, [0.0]);
  ForwardDiff.derivative!(D, getKmatrix, reldense);
  for q = 1:quad.nQuadPoints
    detJ, dNdX = getJacobian(q, shape, element);
    integrator = quad.weights[q] * detJ

    ForwardDiff.gradient!(cp, getCp, [reldense]);
    ForwardDiff.gradient!(rho, getRho, [reldense]);

    fe += zeros(4);
    Ke += integrator .* (dNdX' * D.value * dNdX);
    Ce += (integrator * rho.value * cp.value) .* (shape.N[q,:] * shape.N[q,:]');

    dKe += integrator .* (dNdX' * DiffResults.derivative(D) * dNdX);
    dCe += integrator * (rho.value * DiffResults.derivative(cp)[1] +
      cp.value * DiffResults.derivative(rho)[1]) .* (shape.N[q,:] * shape.N[q,:]');
  end

  return Ke, Ce, fe, dKe, dCe;
end;

function computeLinearGradient(ncon, A, C, b, dKe, dCe, dt, T, dTprev)
  dT = zeros(nNodes, nElements);
  dB = zeros(nNodes, nElements);
  invA = inv(Matrix(A)); x = A \ b;
  for i = 1:nElements
    conn = ncon[i];
    Asparse = dt.*dKe[:,:,i] .+ dCe[:,:,i];

    dT[:,i] = -invA[conn,:]' * (Asparse * x[conn]);
    dB[conn,i] = dCe[:,:,i] * T[conn];
  end

  return dT .+ (invA*(dB .+ C*dTprev));
end;

function computeNonlinearGradient(reldense, ncon, A, J, C, r, dKe, dCe, dLPrev, dt, T, Tn, dTprev, dTi)
  dT = zeros(nNodes, nElements);
  dR = zeros(nNodes, nElements);
  invJ = inv(Matrix(J)); x = J \ r;

  dL,ddLe = computeLatentHeatGradient(ncon, elements, reldense, T, dTi, true);
  ddL,_ = getSparsityPatternHeat(ncon);
  assembler = start_assemble(ddL);
  for i = 1:nElements
    fill!(ddL, 0.0)
    for e = 1:nElements
      assemble!(assembler, ncon[e], ddLe[:,:,i,e]);
    end
    conn = ncon[i];
    dA = dt.*dKe[:,:,i] .+ dCe[:,:,i];
    assemble!(assembler, conn, dA);

    dT[:,i] = -invJ * (ddL * x);
    dR[conn,i] = dCe[:,:,i] * Tn[conn] - dA * T[conn];
  end
```

```
  dR += (C * dTprev - dL + dLPrev - A * dTi)
  return dT .+ (invJ * dR);
end;

function computeLatentHeatGradient(ncon, elements, reldense, T, dT, matrix::Bool)
  dL = zeros(nNodes, nElements);
  ddLe = [];
  if matrix == true
    ddLe = zeros(4,4, nElements, nElements);
  end

  L = DiffResults.DiffResult(0.0, [0.0]);
  rho = DiffResults.DiffResult(0.0, [0.0]);

  for e = 1:nElements
    conn = ncon[e];

    ForwardDiff.gradient!(L, getL, [reldense[e]]);
    ForwardDiff.gradient!(rho, getRho, [reldense[e]]);
    for i = 1:4
      pf, dpf = getPhaseFraction(T[conn[i]]);
      ddpf = getdds(T[conn[i]]);
      dL[conn[i],e] += elements[e].fe[i] * (L.value*pf*DiffResults.derivative(rho)[1] +
        rho.value*pf*DiffResults.derivative(L)[1]);

      elements[e].fe[i]
      dL[conn[i],:] += elements[e].fe[i] * rho.value*L.value*dpf .* dT[conn[i],:];

      if matrix == true
        ddLe[i,:,e,e] += elements[e].Me[i,:] .* (L.value*dpf*DiffResults.derivative(rho)[1] +
          rho.value*dpf*DiffResults.derivative(L)[1]);
        for j = 1:4
          ddLe[i,j,:,e] += elements[e].Me[i,j] * rho.value*L.value*ddpf .* dT[conn[i],:];
        end
      end
    end
  end
  return dL,ddLe;
end;

function solveTimestepGradient(dt, prevTemp, reldense, K,C,F, dKe,dCe, dTprev)
  newTemp = prevTemp; # Predict the new temp base on previous solutions
  res = 1.0; tol = 1e-4; maxItrs = 10000; itr = 0;

  # Solve linear system
  if isInLinearRange(prevTemp)
    vecPart = (F .* dt) .+ (C * prevTemp);
    matPart = (K .* dt) .+ C;
    linTemp = matPart \ vecPart;

    # Ensure solution is in linear region
    if isInLinearRange(prevTemp)
      gradTemp = computeLinearGradient(ncon[boundaries["Domain"]], matPart, C, vecPart, dKe, dCe, dt, prevTemp, dTprev);
      return linTemp, gradTemp;
    end
  end

  # Preallocate matrices
  dL, L = getSparsityPatternHeat(ncon[boundaries["Domain"]]);
  Lprev = getLatentHeatVector(ncon[boundaries["Domain"]], reldense, elements, prevTemp);

  vecPart = (F .* dt) .+ (C * prevTemp) .+ Lprev; # Made global for linesearch (ReverseDiff limitation)
  matPart = (C .+ K .* dt)                # Made global for linesearch (ReverseDiff limitation)

  # Perform newton iterations
  gradTemp = dTprev;
  dLprev,_ = computeLatentHeatGradient(ncon[boundaries["Domain"]], elements, reldense, prevTemp, dTprev, false);
  while res > tol && itr < maxItrs
    iterateLatentHeat!(ncon[boundaries["Domain"]], reldense, elements, dL, L, newTemp);
    residual = (vecPart .- L) .- (matPart) * newTemp;
    jacobian = matPart .+ dL;
    direction = jacobian \ residual;

    gradTemp += computeNonlinearGradient(reldense, ncon[boundaries["Domain"]], matPart, jacobian, C, residual, dKe, dCe,
            dLprev, dt, newTemp, prevTemp, dTprev, gradTemp);

    stepsize = 1.0;
    newTemp =  newTemp .+ stepsize .* direction;
    res = sqrt(residual'*residual);
    itr += 1;
    Printf.@printf "Newton Iteration %i: res = %.2e, alpha = %.2f\n" itr res stepsize
  end;
  return newTemp, gradTemp;
end;
function isInLinearRange(T)
```

```
  return (maximum(T) < tm-tr || minimum(T) > tm+tr);
end;


## Solves a single timestep
function solveTimestep(dt, prevTemp, deltaTemp, reldense, K,C,F)
  # useLineSearch = true;
  global newTemp = prevTemp# + 0.5*deltaTemp;  # Predict the new temp base on previous solutions
  global Tprev = prevTemp;
  res = 1.0; tol = 1e-4; maxItrs = 10000; itr = 0;

  # Solve linear system
  if maximum(prevTemp) < tm-tr || minimum(prevTemp) > tm+tr
    residual = (F .* dt) .+ (C * prevTemp);
    jacobian = (K .* dt) .+ C;
    newTemp = jacobian \ residual;

    # Ensure solution is in linear region
    if maximum(newTemp) < tm - tr || minimum(newTemp) > tm+tr
      return newTemp
    end
  end

  # Preallocate matrices
  dL, L = getSparsityPatternHeat(ncon[boundaries["Domain"]]);
  Lprev = getLatentHeatVector(ncon[boundaries["Domain"]], reldense, elements, prevTemp);

  global vecPart = (F .* dt) .+ (C * prevTemp) .+ Lprev;  # Made global for linesearch (ReverseDiff limitation)
  global matPart = (C .+ K .* dt)              # Made global for linesearch (ReverseDiff limitation)

  # Perform newton iterations
  while res > tol && itr < maxItrs
    iterateLatentHeat!(ncon[boundaries["Domain"]], reldense, elements, dL, L, newTemp);
    residual = (vecPart .- L) .- (matPart) * newTemp;
    jacobian = matPart .+ dL;
    direction = jacobian \ residual;

    # stepsize, m = linesearchBacktrack(newTemp, 0.9, direction);
    m = 0.0; stepsize = 1.0;
    newTemp =  newTemp .+ stepsize .* direction;
    res = sqrt(residual'*residual);
    itr += 1;
    Printf.@printf "Newton Iteration %i: res = %.2e, alpha = %.2f, m = %.2e\n" itr res stepsize m
  end;
  return newTemp;
end;


## ----- Line Search Functions ----- ##
function residualNorm2(x)
  L = getLatentHeatVector(ncon[boundaries["Domain"]], reldense, elements, x);
  residual = (vecPart .- L) .- (matPart) * x;
  return residual' * residual;
end;

function linesearchBacktrack(x, stepsize, direction)
  decay = 0.8;  # decays the stepsize
  beta = 0.01;  # Used for Armijo condition
  diff = DiffResults.DiffResult(0.0, Vector{Float64}(undef, nNodes)); # ReverseDiff diffresult object

  itr = 0;
  maxItrs = 4;

  m = 0;
  ReverseDiff.gradient!(diff, residualNorm2, x);
  fx = diff.value;
  gradfx = DiffResults.gradient(diff);
  m = direction' * gradfx;
  while itr < maxItrs
    fs = residualNorm2(x + stepsize.*direction);
    if fs > fx + beta * stepsize * m
      stepsize *= decay;
    else
      break;
    end
    itr += 1;
  end

  return stepsize, m;
end;


## ----- Helper functions ----- ##
function handleOverwrite(savename)
  if isfile(savename)
    return true;
```

```
  end
  return false;
end;
```

optimizeFunctions.jl

This file has all the main optimization functions and solve functions. This file can generate the Pareto fronts, and structures for the different topology optimization cases.

```
include("phaseChange.jl")
#=
  These functions are boundary condition helpers which are used in the topology Optimization
  function. This way the optimiseTopology function doesn't have to be edited
=#
function thermalBCs!(K, f)
  handleNeumannBCs!(ncon[boundaries["Heat"]], pos, 25000.0, f, 1)
end
function solidBCs!(K, f)
  handleDirichletBCs!(boundaryNodes["Left"], xyDof, [0.0, 0.0], K, f, 2)
  handleNeumannBCs!(ncon[boundaries["TractionBottom"]], pos, [0.0, -2000.0], f, 2)
  # f[2*4+1] = -1.0
end

#=
  Computes the compliance vs compliance pareto front
=#
function generateParetoFront(volfrac = 0.3)
  nPoints = 41;
  weights = range(0.0,1.0, nPoints);

  c = zeros(nPoints, 2);
  for i = 1:nPoints
    weight = weights[i];
    println("[Pareto Generation] Iteration: $i, Weight $weight");
    reldense, _, _ = optimiseTopology(volfrac, weights[i], "", false);
    _, cT = solveHeat(reldense, "", false);
    _, cS = solveSolid(reldense, "", false);
    c[i,:] = [cT, cS];
  end

  return c;
end;

#=
  Determines the pareto optimal structure for a given
    utopia point
=#
function getParetoOptimalStrucure(utopia, volfrac)
  reldense, cTmin = optimiseTopologyThermal(volfrac, "", false);
  _, cSmax = solveSolid(reldense, "", false);

  reldense, cSmin = optimiseTopologySolid(volfrac, "", false);
  _, cTmax = solveHeat(reldense, "", false);

  # Used to normalise
  cSrange = cSmax - cSmin;
  cTrange = cTmax - cTmin;

  # Initialise parameters
  guess = getInitialGuess([cTmax,cSmin]./[cTrange, cSrange], [cTmin,cSmax]./[cTrange, cSrange], utopia./[cTrange, cSrange]);
  println("Initial guess is $guess");

  reldense, _, _ = optimiseTopology(volfrac, guess, "", false);
  _, cT = solveHeat(reldense, "", false);
  _, cS = solveSolid(reldense, "", false);

  a = 0.0; b = guess; c = 1.0;
  ga = goalFunction(cTmax, cSmin, utopia, cTrange, cSrange);
  gb = goalFunction(cT, cS, utopia, cTrange, cSrange);
  gc = goalFunction(cTmin, cSmax, utopia, cTrange, cSrange);
  parabola = [a,b,c, ga,gb,gc];

  # Find optimal point
  change = 1.0; progress = guess; itr = 0; xOpt = 0.0;
  while change > 0.01
    itr += 1;
    prev = [a,b,c];
    guess = 0.5 * (ga*(b^2-c^2) + gb*(c^2-a^2) + gc*(a^2-b^2)) / (ga*(b-c) + gb*(c-a) + gc*(a-b));
```

```
    # Bisection if outside limits
    if guess < 0
      a = 0;
      b = (b - a) / 2;
      ga = gb;
    elseif guess > 1
      b = (c - b) / 2;
      c = 1;
      gc = gb;

    else
      reldense, _, _ = optimiseTopology(volfrac, guess, "", false);
      _, cT = solveHeat(reldense, "", false);
      _, cS = solveSolid(reldense, "", false);
      gx = goalFunction(cT, cS, utopia, cTrange, cSrange);
      if guess > b
        if gx > gb
          c = guess;
          gc = gx;
        else
          a = b;
          b = guess;
          ga = gb;
          gb = gx;
        end
      elseif guess < b
        if gx > gb
          a = guess;
          ga = gx;
        else
          c = b;
          b = guess;
          gc = gb;
          gb = gx;
        end
      end
    end

    g = [ga,gb,gc];
    xv = [a,b,c];
    parabola = [parabola;; xv;g]

    xOpt = xv[findmin(g)[2]];
    change = maximum(abs.(prev .- xOpt));
    progress = [progress; xOpt];

    Printf.@printf "Itr: %i - Change: %.2f - Error: %.2e - Guess: %.2f - Compliance: %.6e, %.6e\n" itr change minimum(g) xOpt
            cT cS
  end

  return reldense, cT, cS, xOpt, progress, parabola;
end;

function goalFunction(cT,cS, utopia, cTrange,cSrange, p=3)
  err = (abs((cT - utopia[1])/cTrange)^p + abs((cS - utopia[2])/cSrange)^p)^(1.0/p);
end;

function getInitialGuess(p1, p2, px)
  vec = p1 - p2;
  veclength = sqrt(vec[1]^2 + vec[2]^2);
  vec = vec ./ veclength;

  vecpx = p1 - px;
  distance = vecpx[1]*vec[1] + vecpx[2]*vec[2];

  return distance / veclength;
end;

#=
  Topology Optimization function
  Optimises topology using the SIMP method

  Inputs:
    - k (temporary variable which will be used for porosity of cell)
    - volFrac (desired volume fraction)

  Outputs:
    - relDense (array of element relative densities)
    - c (domain compliance)
=#
function optimiseTopology(volFrac, weight, savename = "", save = true)
  global p = 1.0
  itr = 0; pmax = 3; change = 1.0

  w1 = weight;
  w2 = 1.0 - weight;
```

```
# GOCM Stuff
lambda = 1.0;
constraint = 0.0;
first = true; f0 = []

# Preallocate some arrays
u_T = []; c_T = 0.0;
u_S = []; c_S = 0.0;
relDense = volFrac * ones(Float64, nElements)

while (itr < 100)
  # Main FE solve
  FEAtime = @elapsed begin
    # Solve thermal problem
    global elementType = 0
    globalAssembly!(relDense, ncon[boundaries["Domain"]], Ke_T, K_T, f_T, quad, shape, 1)
    thermalBCs!(K_T, f_T)
    u_T = K_T \ f_T # Solve for temperature
    c_T = u_T' * f_T

    # Solve solid-elastic problem
    global elementType = 1
    globalAssembly!(relDense, ncon[boundaries["Domain"]], Ke_S, K_S, f_S, quad, shape, 2)
    solidBCs!(K_S, f_S)
    u_S = K_S \ f_S # Solve for temperature
    c_S = u_S' * f_S
  end


  # Compute element sensitivities (Computes dc/de)
  sensitivityTime = @elapsed begin
    # Increase penalisation parameter after 10 iterations to improve convergence
    if itr > 10 && p < pmax
      p *= 1.02
    end
    sensitivity_T = sensitivityAnalysis(ncon[boundaries["Domain"]], relDense, p, u_T, Ke_T, HF, nElements, 1)
    sensitivity_S = sensitivityAnalysis(ncon[boundaries["Domain"]], relDense, p, u_S, Ke_S, HF, nElements, 2)

    if first == true
      f0 = [norm(sensitivity_T), norm(sensitivity_S)]
    end
    sensitivity_T = sensitivity_T ./ f0[1];
    sensitivity_S = sensitivity_S ./ f0[2];
    sensitivity = (w1 .* sensitivity_T) .+ (w2 .* sensitivity_S)
  end

  # Update element densities
  bisectionTime = @elapsed begin
    relDenseNew = updateDensity(sensitivity, relDense, volFrac, nElements)
    # constraint, lambda, relDenseNew = updateDensityGOCM(constraint, sensitivity, relDense, volFrac, nElements, lambda)
  end

  # Iterate
  itr += 1
  change = maximum(abs.(relDense - relDenseNew))
  relDense = relDenseNew  # Update relDense

  Printf.@printf "Itr: %i - Conv: %.2e - Compliance: %.2e, %.2e - Constraint: %.2e [TIMERS] FEA = %.4f, sens = %.4f,
          bisection = %.4f\n" itr change c_T c_S (sum(relDense)/(volFrac*nElements)) FEAtime sensitivityTime bisectionTime
end

if save == false
  return relDense, c_T, c_S;
end

if length(savename) == 0
  println("Enter save name");
  savename = readline();
end

while handleOverwrite(savename) == true
  println("You are about to overwrite an existing file. Would you like to continue? (y/n)");
  println("If you would like to cancel press 'c'");
  answer = readline();
  if answer == "y"
    println("Saving")
    break;
  elseif answer == "n"
    println("Enter a new save name");
    savename = readline();
  elseif answer == "c"
    println("Cancelling");
    save = false;
    break;
  else
```

```
      println("Unrecognised key");
    end
  end

  if save == true
    export2vtk(savepath * savename * ".vtu", pos, ncon, u_T, relDense)
    export2vtk(savepath * savename * ".vtu", pos, ncon, u_S, relDense)
  end
  return relDense, c_T, c_S
end;

function optimiseTopologyThermal(volFrac, savename = "", save = true)
  global p = 1.0;
  itr = 0; pmax = 3.0; change = 1.0

  # GOCM Stuff
  lambda = 1.0;
  constraint = 0.0;
  first = true; f0 = []

  # Preallocate some arrays
  u_T = []; c_T = 0.0;
  relDense_T = volFrac * ones(Float64, nElements)

  while (itr < 50)
    # Main FE solve
    FEAtime = @elapsed begin
      # Solve thermal problem
      global elementType = 0
      globalAssembly!(relDense_T, ncon[boundaries["Domain"]], Ke_T, K_T, f_T, quad, shape, 1)
      thermalBCs!(K_T, f_T)
      u_T = K_T \ f_T # Solve for temperature
    end


    # Compute element sensitivities (Computes dc/dde)
    sensitivityTime = @elapsed begin
      # Increase penalisation parameter after 10 iterations to improve convergence
      if itr > 10 && p < pmax
        p *= 1.02
      end
      sensitivity_T = sensitivityAnalysis(ncon[boundaries["Domain"]], relDense_T, p, u_T, Ke_T, HF, nElements, 1)

      # if first == true
      #   f0 = u_T' * f_T
      # end
      # sensitivity_T = sensitivity_T ./ f0
    end

    # Update element densities
    bisectionTime = @elapsed begin
      relDenseNew_T = updateDensity(sensitivity_T, relDense_T, volFrac, nElements)
      # constraint, lambda, relDenseNew_T = updateDensityGOCM(constraint, sensitivity_T, relDense_T, volFrac, nElements,
            lambda)
    end

    # Iterate
    itr += 1
    change = maximum(abs.(relDense_T - relDenseNew_T))
    c_T = u_T' * f_T      # Compute compliance
    relDense_T = relDenseNew_T  # Update relDense

    Printf.@printf "Itr: %i - Conv: %.2e - Compliance: %.2e - Constraint: %.2e [TIMERS] FEA = %.4f, sens = %.4f, bisection =
          %.4f\n" itr change c_T (sum(relDense_T)/(volFrac*nElements)) FEAtime sensitivityTime bisectionTime
  end

  if save == false
    return relDense_T, c_T;
  end

  if length(savename) == 0
    println("Enter save name");
    savename = readline();
  end

  while handleOverwrite(savename) == true
    println("You are about to overwrite an existing file. Would you like to continue? (y/n)");
    println("If you would like to cancel press 'c'");
    answer = readline();
    if answer == "y"
      println("Saving")
      break;
    elseif answer == "n"
      println("Enter a new save name");
      savename = readline();
    elseif answer == "c"
```

```
      println("Cancelling");
      save = false;
      break;
    else
      println("Unrecognised key");
    end
  end

  if save == true
    export2vtk(savepath * savename * ".vtu", pos, ncon, u_T, relDense_T)
  end
  return relDense_T, c_T
end;

function optimiseTopologySolid(volFrac, savename = "", save = true)
  global p = 1.0;
  itr = 0; pmax = 3.0; change = 1.0

  # GOCM stuff
  lambda = 1.0;
  constraint = 0.0;
  first = true; f0 = []

  # Preallocate some arrays
  u_S = []; c_S = 0.0;
  relDense_S = volFrac * ones(Float64, nElements)

  while (itr < 100)
    # Main FE solve
    FEAtime = @elapsed begin
      # Solve solid-elastic problem
      global elementType = 1;
      globalAssembly!(relDense_S, ncon[boundaries["Domain"]], Ke_S, K_S, f_S, quad, shape, 2)
      solidBCs!(K_S, f_S)
      u_S = K_S \ f_S # Solve for temperature
    end


    # Compute element sensitivities (Computes dc/de)
    sensitivityTime = @elapsed begin
      # Increase penalisation parameter after 10 iterations to improve convergence
      if itr > 10 && p < pmax
        p *= 1.02
      end
      sensitivity_S = sensitivityAnalysis(ncon[boundaries["Domain"]], relDense_S, p, u_S, Ke_S, HF, nElements, 2)

      if first == true
        # f0 = u_S' * f_S
        f0 = norm(sensitivity_S);
      end
      sensitivity_S = sensitivity_S ./ f0
    end

    # Update element densities
    bisectionTime = @elapsed begin
      relDenseNew_S = updateDensity(sensitivity_S, relDense_S, volFrac, nElements)
      # constraint, lambda, relDenseNew_S = updateDensityGOCM(constraint, sensitivity_S, relDense_S, volFrac, nElements,
          lambda)
    end

    # Iterate
    itr += 1
    change = maximum(abs.(relDense_S - relDenseNew_S))
    c_S = u_S' * f_S  # Compute compliance
    relDense_S = relDenseNew_S  # Update relDense

    Printf.@printf "Itr: %i - Conv: %.2e - Compliance: %.2e - Constraint: %.2e [TIMERS] FEA = %.4f, sens = %.4f, bisection =
        %.4f\n" itr change c_S (sum(relDense_S)/(volFrac*nElements)) FEAtime sensitivityTime bisectionTime
  end

  if save == false
    return relDense_S, c_S;
  end

  if length(savename) == 0
    println("Enter save name");
    savename = readline();
  end

  while handleOverwrite(savename) == true
    println("You are about to overwrite an existing file. Would you like to continue? (y/n)");
    println("If you would like to cancel press 'c'");
    answer = readline();
    if answer == "y"
      println("Saving")
      break;
```

```
    elseif answer == "n"
      println("Enter a new save name");
      savename = readline();
    elseif answer == "c"
      println("Cancelling");
      save = false;
      break;
    else
      println("Unrecognised key");
    end
  end

  if save == true
    export2vtk(savepath * savename * ".vtu", pos, ncon, u_S, relDense_S)
  end

  return relDense_S, c_S
end;

function optimiseTopologyPhaseChange(volFrac, endTime, savename = "", save = true)
  global p = 1.0;
  itr = 0; pmax = 3.0; change = 1.0

  # GOCM stuff
  lambda = 1.0;
  constraint = 0.0;
  first = true; f0 = []

  # Preallocate some arrays
  u_PC = []; c_PC = 0.0;
  relDense_PC = volFrac * ones(Float64, nElements)

  while (itr < 50)
    # Main FE solve
    FEAtime = @elapsed begin
      # Solve phase-change problem
      dt = 120.0;
      initialTemp = 25.0+273.15;
      u_PC, c_PC, sensitivity_PC = solveWithGradient(endTime, dt, initialTemp, relDense_PC);
    end


    # Filter sensitivities
    sensitivityTime = @elapsed begin
      # Increase penalisation parameter after 10 iterations to improve convergence
      if itr > 10 && p < pmax
        p *= 1.02
      end

      # if first == true
      #   # f0 = u_S' * f_S
      #   f0 = norm(sensitivity_PC);
      # end
      # sensitivity_PC = sensitivity_PC ./ f0

      rowSum = zeros(size(HF)[1]);
      for i = 1:size(HF)[1]
        rowSum[i] = sum(HF[i,:]);
      end
      sensitivity_PC = HF * (relDense_PC .* sensitivity_PC) ./ rowSum ./ max.(1e-3, relDense_PC)
    end

    # Update element densities
    bisectionTime = @elapsed begin
      relDenseNew_PC = updateDensity(sensitivity_PC, relDense_PC, volFrac, nElements)
    end

    # Iterate
    itr += 1
    change = maximum(abs.(relDense_PC - relDenseNew_PC))
    relDense_PC = relDenseNew_PC  # Update relDense

    Printf.@printf "Itr: %i - Conv: %.2e - Compliance: %.2e - Constraint: %.2e [TIMERS] FEA = %.4f, sens = %.4f, bisection =
            %.4f\n" itr change c_PC (sum(relDense_PC)/(volFrac*nElements)) FEAtime sensitivityTime bisectionTime
  end

  if save == false
    return relDense_PC, c_PC;
  end

  if length(savename) == 0
    println("Enter save name");
    savename = readline();
  end

  while handleOverwrite(savename) == true
```

```
    println("You are about to overwrite an existing file. Would you like to continue? (y/n)");
    println("If you would like to cancel press 'c'");
    answer = readline();
    if answer == "y"
      println("Saving")
      break;
    elseif answer == "n"
      println("Enter a new save name");
      savename = readline();
    elseif answer == "c"
      println("Cancelling");
      save = false;
      break;
    else
      println("Unrecognised key");
    end
  end

  if save == true
    export2vtk(savepath * savename * ".vtu", pos, ncon, u_PC, relDense_PC)
  end

  return relDense_PC, c_PC
end;

function solveSolid(relDense_S, savename = "", save = true)
  global p = 1.0;

  global elementType = 1;
  globalAssembly!(relDense_S, ncon[boundaries["Domain"]], Ke_S, K_S, f_S, quad, shape, 2)
  solidBCs!(K_S, f_S)
  u_S = K_S \ f_S

  if save == false
    return u_S, u_S'*f_S;
  end

  if length(savename) == 0
    println("Enter save name or press 'c' to skip");
    savename = readline();
    if savename == "c"
      save = false;
    end
  end

  while handleOverwrite(savename) == true
    println("You are about to overwrite an existing file. Would you like to continue? (y/n)");
    println("If you would like to cancel press 'c'");
    answer = readline();
    if answer == "y"
      println("Saving")
      break;
    elseif answer == "n"
      println("Enter a new save name");
      savename = readline();
    elseif answer == "c"
      println("Cancelling");
      save = false;
      break;
    else
      println("Unrecognised key");
    end
  end

  if save == true
    export2vtk(savepath * savename * ".vtu", pos, ncon, u_S, relDense_S)
  end

  return u_S, u_S'*f_S;
end;

function solveHeat(relDense_T, savename = "", save = true)
  global p = 1.0;

  global elementType = 0;
  globalAssembly!(relDense_T, ncon[boundaries["Domain"]], Ke_T, K_T, f_T, quad, shape, 1)
  thermalBCs!(K_T, f_T)
  u_T = K_T \ f_T

  if save == false
    return u_T, u_T'*f_T;
  end

  if length(savename) == 0
    println("Enter save name");
    savename = readline();
```

```
  end

  while handleOverwrite(savename) == true
    println("You are about to overwrite an existing file. Would you like to continue? (y/n)");
    println("If you would like to cancel press 'c'");
    answer = readline();
    if answer == "y"
      println("Saving")
      break;
    elseif answer == "n"
      println("Enter a new save name");
      savename = readline();
    elseif answer == "c"
      println("Cancelling");
      save = false;
      break;
    else
      println("Unrecognised key");
    end
  end

  if save == true
    export2vtk(savepath * savename * ".vtu", pos, ncon, u_T, relDense_T)
  end

  return u_T, u_T'*f_T;
end;


#===== Helper Functions =====#
function handleOverwrite(savename)
  if isfile(savepath * savename * ".vtu")
    return true;
  end
  return false;
end;

#optimiseTopologyThermal(0.3, "", false)
```

topologyOptimisation.jl

This file has the main topology optimization algorithms such as the filtering and sensitivity analysis methods.

```
include("latticeProperties.jl")
using ForwardDiff

function sensitivityAnalysis(ncon, relDense, p, u, Ke, HF, nElements, nDofs)
  # Note objective function is min: c = T' * f; s.t: int p_e = gamma * f (where p_e is the element density, gamma is the
        volume fraction)
  # Compute element sensitivities for each element. dc/dp
  sensitivity = Vector{Float64}(undef, nElements)
  Threads.@threads for e = 1:nElements
    uElement = zeros(nDofs * 4)
    for i = 1:4
      for j = 1:nDofs
        uElement[nDofs*i - (nDofs - j)] = u[nDofs*ncon[e][i] - (nDofs - j)]
      end
    end
    sensitivity[e] = -uElement' * (Ke[:,:,e]) * uElement
  end
  rowSum = zeros(size(HF)[1]);
  for i = 1:size(HF)[1]
    rowSum[i] = sum(HF[i,:]);
  end

  return HF * (relDense .* sensitivity) ./ rowSum ./ max.(1e-3, relDense) # Filter sensitivities
end


function updateDensity(sensitivity, relDense, volFrac, nElements)
  # Bisection method to find optimum Lagrange Multiplier
  # This is essentially a basic root finder
  # Refer to the paper (https://doi.org/10.3390/app11073175) for equation numbers
  # Currently this uses the 'non-generalised' version as I only have 1 constraint
  relDenseNew = []
  lambda1 = 0.0; lambda2 = 1.0e9;
  move = 0.2  # This parameter controls how much an element density can change. Too large and a local optimum is found too
        quickly which isnt close to the global optimum
```

```
  while (lambda2 - lambda1) / (lambda1 + lambda2) > 1e-4  # While root not found
    mid = 0.5 * (lambda1 + lambda2) # Bisect
    B = -sensitivity / mid       # Equation 6
    # xb = relDense .* B.^0.4 # Equation 7 (part 1)
    xb = relDense .* (1.0 .+ (B .- 1.0)./(1.0 * 2.0));

    # Equation 7 (fully implimented)
    relDenseNew = max.(0.01, max.(relDense.-move, min.(1.0, min.(relDense.+move, xb))))

    # This checks the dL/dlambda = int p_e - gamma = 0 condition and updates the bisection accordingly
    if sum(relDenseNew) - (volFrac * nElements) > 0
      lambda1 = mid
    else
      lambda2 = mid
    end
  end

  return relDenseNew
end

#=
  This GOCM function updates the density using the generalised version of
  the above function.

  The constraint is not necessarily satisfied in all iterations.
  In its current form, only the relative volume constraint is implimented.
  This will be generalised to be able to include number of constraints assuming
    their derivatives are known. For example, a minimum displacement can be
    included if the derivative can be found. This will require more work
=#
function updateDensityGOCM( # Inputs
    gl,          # Previous constraint
    sensitivity,     # Derivative of objective function w.r.t relDense
    relDense,        # Relative density of all elements
    volFrac,       # Desired volume fraction
    nElements,       # Number of elements
    lambda)        # Lagrange multiplier(s)


  esp = 0.05; move = 0.2;
  g = sum(relDense) / (nElements * volFrac) - 1;  # Constraint
  dg = g - gl                    # Change in constraint from Previous iteration

  tol = 1e-10
  # Determine update parameter (essentially controls how fast the lagrange multiplier can update)
  if (g > tol && dg > tol) || (g < -tol && dg < -tol);     p0 = 1.0;
  elseif (g > tol && dg > -esp) || (g < -tol && dg < esp);  p0 = 0.5;
  else;                            p0 = 0.0;
  end


  lambda = lambda * (1.0 + p0 * (g + dg));  # Update lagrange multiplier
  D = -sensitivity ./ (lambda / nElements); # Determine scale factor
  return g, lambda, max.(0.001, max.(relDense.-move, min.(1.0, min.(relDense.+move, relDense .* abs.(D) .^0.4))));  # return
          new constraint, lambda, and updated relative density
end
```

latticeProperties.jl

This file has all the lattice properties and related methods. This is also where the lattice material properties are defined.

```
#=
  In this initial implimentation, the cell strut radius will be determined
  base on the porosity. This is because the SIMP method works by penalising
  the relative density of an element. The porosity of the cell and relative
  density of the element have a strong corrilation as each define the
  material properties.
=#

using ForwardDiff

#----- Global variables for the lattice structure -----#
# Structure properties
phi = 45.0 * pi / 180.0;  # Angle of the struts
h_cell = 5.0         # Height of the cell
h=h_cell;
# Material properties
k_pcm = 0.21;    # PCM conductivity
```

```
k_cell = 160.0;   # Cell strut material conductivity
E_cell = 70.0e6;  # Cell strut material elastic modulus

rho_pcm = 814.0;  # Density of pcm
rho_cell = 2700.0;  # Denisty of lattice
cp_pcm = 2180.0;  # Specific heat (at constant pressure)
cp_cell = 895.0;  # Specific heat of lattice
L_pcm = 244000.0; # Latent heat of melting

tm = 273.15 + 29.0; # Melting temperature
tr = 1.0;      # Half Mushy zone temperature range (T_mushy = [tm-tr : tm+tr])

#----- Cell Property Getters -----#
#=
  Maps the relative density of the elements in the SIMP method
    to a reasonable range of cell volume fraction using a
    linear interpolation
=#
function reldense2volfrac(reldense)
  minVF = 0.0;  # Minimum volume fraction
  maxVF = 0.50; # Maximum volume fraction

  return minVF + (maxVF - minVF) * reldense;
end

#=
  Gets the volume fraction (V_struct/V_total) of the cell
  I.e, this is 1 - porosity
=#
function getVolfrac(r)

  theta = atan(tan(phi) / sqrt(2))
  if celltype == "bcc"
    VolFrac = 2*tan(theta^2* (r^2/h_cell^2) * (pi * (1+4/sin(theta)) -16/3* r/h_cell * (3.137/sin(pi-2*theta) + 4.923/sin(pi
          /2-theta))));
  elseif celltype == "f2ccz"
    VolFrac = tan(phi)^2*r^2/h_cell^2*(pi*(1+4/sin(phi))-16/3*r/h*(2.935/sin(pi-2*phi)+3.667/sin(pi/2-phi)));
  end
  return VolFrac
end;

#=
  Determines the radius that gives the target porosity
    using the Newton Raphson method
=#
function determineRadius(targetVolFrac, guess = 1.0)
  # Objective function
  func(r) = getVolfrac(r) - targetVolFrac
  res = 1.0; tol = 1e-8;
  itr = 0; maxItrs = 100;

  # Newton Raphson method used to find actual radius
  while abs(res) > tol && itr < maxItrs
    res = func(guess)
    g = ForwardDiff.derivative(func, guess)

    guess = guess - (res / g)
  end

  return guess
end

#----- Material Property getters -----#
#=
  Gets the elastic modulus of the cell
  The only input is the cell struct radius as all other properties
    are currently fixed
=#
function getElasticModulus(r)
  E = [];
  if celltype == "bcc"
    prefactor = 2.0 * sqrt(2) * pi * E_cell * r^2 / h_cell^2;
    postfactor = sin(phi)^2 * cos(phi)
    E = prefactor * (1 + 12*r^2/h_cell^2 * (sin(phi)^2 - 1) * tan(phi)^2) * postfactor;
  elseif celltype == "f2ccz"
    prefactor = 2.0*pi*E_cell*r^2/h_cell^2;
    postfactor = sin(phi)^3;
    E = prefactor * (1 + 12*r^2/h_cell^2 * (sin(phi)^2) * tan(phi)^2) * postfactor;
  end
  return E
end;
function getElasticModulus3(r)
  if celltype == "bcc"
  prefactor = 8.0 * pi * E_cell * (r^2 / h_cell^2);
  postfactor = sin(phi)^3 * tan(phi)^2
  E = prefactor * (1 + 12 * r^2/h_cell^2 * cos(phi)^2) * postfactor;
```

```
  elseif celltype == "f2ccz"
  prefactor= pi*E_cell*r^2/h_cell^2;
  postfactor = tan(phi)^2;
  E = prefactor * (1+4*sin(phi)^3*(1 + 12*r^2/h_cell^2 * cos(phi)^2)) * postfactor;
  end
  return E
end;

function getElasticityMatrix(r);
  nu = 0.3;
  E12 = getElasticModulus(r);
  E3 = getElasticModulus3(r);
  G13 = E12 / (2 * (1 + nu));

  return (1 / (1 - nu^2)) .* [E12;   nu*E3; 0;;
                  nu*E3; E3;      0;;
                  0;     0;       G13*(1-nu^2)];
end;

#=
  Gets the ETC of the cell
  The only input is the cell struct radius as all other properties
    are currently fixed
=#
function getConductivity(r)
  theta = atan(tan(phi) / sqrt(2));
  epsilon = 1 - getVolfrac(r);
  if celltype == "bcc"

    V4 = (2 * (16 / (3 * sin(pi - 2*theta))) * r^3) - (12 * (sqrt(8) - sqrt(6)) * r^3);
    t4 = (V4 * sin(theta) * cos(theta)^2) ^ (1/3)
    Lstr = (h_cell / (2 * sin(theta))) - (t4 * sqrt((2 / cos(theta)^2) + (1 / sin(theta)^2)))

    R1 = (2 * cos(theta)^2) / (4.277 * sin(theta)* t4 * k_cell);
    R2 = Lstr / (pi * r^2 * k_cell)
    R3 = R1;
    Rstr = 2 * (R1 + R2 + R3)
    Rs = Rstr / 4

    k = (epsilon * k_pcm) + (1 / (h_cell * Rs));
  elseif celltype == "f2ccz"
    Ï 4 = 1.732;
        Ï 2 = 2.836;
        V_2Str = 16/3/sin(pi-2*phi)*r^3; # Volumen wenn sich 2 Streben schneiden
        V_4Str = V_2Str*2-12*(8^0.5-6^0.5)*r^3;
        b_2 = (V_2Str*sin(phi)*cos(phi)^2)^(1/3);
        b_4 = (V_4Str*sin(phi)*cos(phi)^2)^(1/3);

        L_box = b_4/2*(2/cos(phi)^2+1/sin(phi)^2)^0.5+b_2/2*(2/cos(phi)^2+1/sin(phi)^2)^0.5;
        L_ges = h/(2*sin(phi))-L_box;
        #eps = 1-2*tan(omega)**2/h**3*(4*pi*(2*L_ges)*r**2+2*V_4Str)

        R1 = cos(phi)^2/(Ï 4*sin(phi)*b_4*k_cell);
        R2 = L_ges/(pi*r^2*k_cell);
        R3 = cos(phi)^2/(Ï 2*sin(phi)*b_2*k_cell);

        R_lat = (2/(2*(R1+R2+R3)))^(-1);

        V_ie = pi*r^2*(h/sin(phi)*2+h);
        h_ie = V_ie*tan(phi)/h^2;
        R_pcm = (h/tan(phi)-h_ie)/(h^2/tan(phi)*epsilon*k_pcm)+h_ie/(h^2/tan(phi)*k_cell);

        Î»_s = 1/(R_lat*h);
        Î»_p = 1/(R_pcm*h);
        k = Î»_s + Î»_p; # [W/(m*K)]
  end
  return k;
end;
function getConductivity3(r)
  theta = atan(tan(phi) / sqrt(2));
  epsilon = 1 - getVolfrac(r);
  if celltype == "bcc"
    V4 = (2 * (16 / (3 * sin(pi - 2*theta))) * r^3) - (12 * (sqrt(8) - sqrt(6)) * r^3);
    t4 = (V4 * sin(theta) * cos(theta)^2) ^ (1/3)
    Lstr = (h_cell / (2 * sin(theta))) - (t4 * sqrt((1 / cos(theta)^2) + (2 / sin(theta)^2)))

    R1 = (2 * sin(theta)^2) / (1.903 * cos(theta)* t4 * k_cell);
    R2 = Lstr / (pi * r^2 * k_cell)
    R3 = R1;
    R_ges = 1/2*(R1+R2+R3);
        Î»_s = 1/(R_ges*h);
        Î»_p = epsilon*k_pcm;
        k = Î»_s + Î»_p; # [W/(m*K)]
  elseif celltype == "f2ccz"
    Ï 4 = 1.448;
```

```
        Ï 2 = 1.168;
        V_2Str = 16/3/sin(pi-2*phi)*r^3; # Volumen wenn sich 2 Streben schneiden
        V_4Str = V_2Str*2-12*(8^0.5-6^0.5)*r^3;
        b_2 = (V_2Str*sin(phi)^2*cos(phi))^(1/3);
        b_4 = (V_4Str*sin(phi)^2*cos(phi))^(1/3);
        L_box = b_4/2*(1/cos(phi)^2+2/sin(phi)^2)^0.5+b_2/2*(1/cos(phi)^2+2/sin(phi)^2)^0.5;
        L_ges = h/(2*sin(phi))-L_box;
        #eps = 1-2*tan(omega)**2/h**3*(4*pi*(2*L_ges)*r**2+2*V_4Str)
        R1 = 4/2*sin(phi)^2/(Ï 4*cos(phi)*b_4*k_cell);
        R2 = L_ges/(pi*r^2*k_cell);
        R3 = sin(phi)^2/(Ï 2*cos(phi)*b_2*k_cell);
        RZ = h/(pi*r^2*k_cell);
        R_lat = (4/(2*(R1+R2+R3))+1/RZ)^(-1);
        Î»_s = tan(phi)^2/(R_lat*h);
        Î»_p = epsilon*k_pcm;
        k = Î»_s + Î»_p; # [W/(m*K)]
  end
  return k;
end;

function getConductivityMatrix(r);
  k12 = getConductivity(r);
  k3 = getConductivity3(r);

  return [k12;0;;0;k3];
end

#=
  Gets the density, apparent specific heat and latent heat of an element
=#
function getRhoCpL(reldense)
  VolFrac = reldense2volfrac(reldense);
  rho = VolFrac * rho_cell + (1 - VolFrac) * rho_pcm
  cp = ((VolFrac * rho_cell) / rho * cp_cell) + (((1 - VolFrac) * rho_pcm) / rho * cp_pcm);
  L = (1 - VolFrac) * rho_pcm / rho * L_pcm
  return rho, cp, L;
end;

function getRho(reldense)
  VolFrac = reldense2volfrac(reldense[1]^p);
  rho = VolFrac * rho_cell + (1 - VolFrac) * rho_pcm
  return rho;
end;

function getCp(reldense)
  VolFrac = reldense2volfrac(reldense[1]^p);
  rho = VolFrac * rho_cell + (1 - VolFrac) * rho_pcm
  cp = ((VolFrac * rho_cell) / rho * cp_cell) + (((1 - VolFrac) * rho_pcm) / rho * cp_pcm);
  return cp;
end;

function getL(reldense)
  VolFrac = reldense2volfrac(reldense[1]^p);
  rho = VolFrac * rho_cell + (1 - VolFrac) * rho_pcm
  L = (1 - VolFrac) * rho_pcm / rho * L_pcm
  return L;
end;


using DelimitedFiles
function exportCells(reldense)
  h = pos[ncon[boundaries["Domain"]][1][3],2] - pos[ncon[boundaries["Domain"]][1][1],2];

  nCells = nElements;
  cellRadii = determineRadius.(reldense2volfrac.(reldense));

  cellPos = zeros(nCells,2);
  for i = 1:nCells
    cellPos[i,:] = 0.5 * (pos[ncon[boundaries["Domain"]][i][3],1:2] + pos[ncon[boundaries["Domain"]][i][1],1:2]);
  end

  writeData = [h;nCells;cellRadii;cellPos[:,1];cellPos[:,2]];
  writedlm("dehomogenize_data.csv", writeData, ",");
end;
```

Helper Files

The following files are helper files for things such as shape functions and file io.

shapeFunctions.jl

This file defines the shape functions and related methods.

```
#=
  This struct stores the finite element matrices which can be precomputed
    for efficiency. Their basic forms are computed and the material
    properties can be multiplied with these matrices in the assembly stage
=#
mutable struct Element
  Me::AbstractMatrix{Float64}; # Mass matrix (Me = int N'*N dOmega)
  Ke::AbstractMatrix{Float64}; # Stiffness matrix (Ke = int gradN*gradN' dOmega)
  fe::AbstractVector{Float64}; # Forcing array (fe = int N dOmega)

  Element(size::Int64) = new(zeros(size,size), zeros(size,size), zeros(size));
end;

#=
  This function initialises the finite element matrices.
=#
function initElementMatrices(ncon, coords, quad, shape)
  elementMatrices = Vector{Element}(undef, nElements);
  for e = 1:nElements
    Me = zeros(4,4);
    Ke = zeros(4,4);
    fe = zeros(4);

    elePoints = coords[ncon[e],:]

    # Loop over quadrature points
    for q = 1:quad.nQuadPoints
      detJ, dNdX = getJacobian(q, shape, elePoints); # Gets the determinant and shape function deriviatives
      dNdX1 = dNdX[1,:]; # For easy access
      dNdX2 = dNdX[2,:]; # For easy access
      integrator = quad.weights[q] * detJ; # Integration constant

      # Loop over element nodes
      for i = 1:4
        fe[i] += integrator * shape.N[q,i];
        for j = 1:4
          Me[i,j] += integrator * shape.N[q,i] * shape.N[q,j];
          Ke[i,j] += integrator * (dNdX1[i] * dNdX1[j] + dNdX2[i] * dNdX2[j]);
        end
      end
    end

    # Assign matrices
    elementMatrices[e] = Element(4);
    elementMatrices[e].Me = Me;
    elementMatrices[e].Ke = Ke;
    elementMatrices[e].fe = fe;
  end

  return elementMatrices;
end;


# Struct to store quadrature rules
struct QuadratureRules
  nQuadPoints::Int16
  weights::Vector
  quadPoints::Matrix
end;

# Enum for selecting type of element
@enum ElementType begin
  Quad1 = 1;
  Lin1 = 2;
end;

# Gauss weughts and positions
function getQuadratureRules(type::ElementType)
  if type == Quad1
    QuadratureRules(4, [1.0,1.0,1.0,1.0], 1/sqrt(3.0) * [-1.0;-1.0;;  1.0;-1.0;;  1.0;1.0;;  -1.0;1.0])
  elseif type == Lin1
    QuadratureRules(1, [2.0], [0.0;;])
  end
end;

# Struct to store the shape functions and derivatives
struct ShapeFunctions1D
  N::Matrix
  dNdXi::Matrix
end;
```

```
struct ShapeFunctions2D
  nNodes::Int32
  N::Matrix
  dNdXi1::Matrix
  dNdXi2::Matrix
end;
struct ShapeFunctions3D
  nNodes::Int32
  N::Matrix
  dNdXi1::Matrix
  dNdXi2::Matrix
  dNdXi3::Matrix
end;

# Gets the shape functions of the selected element
function getShapeFunctions(type::ElementType, quad::QuadratureRules)
  if type == Quad1
    xi1 = quad.quadPoints[1,:]
    xi2 = quad.quadPoints[2,:]
    shapeFuncs = [
      (1.0 .- xi1) .* (1.0 .- xi2)/4.0;;
      (1.0 .+ xi1) .* (1.0 .- xi2)/4.0;;
      (1.0 .+ xi1) .* (1.0 .+ xi2)/4.0;;
      (1.0 .- xi1) .* (1.0 .+ xi2)/4.0]
    return shapeFuncs
  elseif type == Lin1
    xi = quad.quadPoints
    shapeFuncs = 0.5 * [1.0 .- xi;; xi .- 1.0]
    return shapeFuncs
  end
end;

# Gets the shape function gradients
function getShapeDXi1(quad::QuadratureRules)
  xi2 = quad.quadPoints[2,:]
  shapeDxi1 = [
    (xi2 .- 1.0)/4.0;;
    (1.0 .- xi2)/4.0;;
    (1.0 .+ xi2)/4.0;;
     -(1.0 .+ xi2)/4.0]
  return shapeDxi1
end;
function getShapeDXi2(quad::QuadratureRules)
  xi1 = quad.quadPoints[1,:]
  shapeDxi1 = [
    (xi1 .- 1.0)/4.0;;
     -(1.0 .+ xi1)/4.0;;
    (1.0 .+ xi1)/4.0;;
    (1.0 .- xi1)/4.0]
  return shapeDxi1
end;
function getShapeDxi(quad::QuadratureRules)
  dNdXi = [-0.5;; 0.5]
  return dNdXi
end;
function getBmatrix!(B::Matrix, dNdX::Vector, dNdY::Vector)
  for i = 1:4
    B[1,2*i-1]  = dNdX[i];
    B[2,2*i]  = dNdY[i];
    B[3,2*i-1]  = dNdY[i];
    B[3,2*i]  = dNdX[i];
  end
end


function getShapeAndDerivatives(type::ElementType, quad::QuadratureRules)
  if type == Quad1
    ShapeFunctions2D(4, getShapeFunctions(type, quad), getShapeDXi1(quad), getShapeDXi2(quad))
  elseif type == Lin1
    ShapeFunctions1D(getShapeFunctions(type, quad), getShapeDxi(quad))
  end
end;

# Gets the jacobian, value of gradients at a gauss point
function getJacobian(iQuad, shapeFuncs::ShapeFunctions2D, elePoints)
  dNdXi1 = shapeFuncs.dNdXi1[iQuad,:]'
  dNdXi2 = shapeFuncs.dNdXi2[iQuad,:]'
  jacobian = [
    dNdXi1 * elePoints[:,1]; dNdXi1 * elePoints[:,2];;
    dNdXi2 * elePoints[:,1]; dNdXi2 * elePoints[:,2]]

  detJ = (jacobian[1,1] * jacobian[2,2]) - (jacobian[2,1] * jacobian[1,2])
  dNdX = jacobian \ [dNdXi1; dNdXi2]
  return detJ, dNdX
end;
```

assembly.jl

This file handles all the finite element assembly, as well as global system assembly.

```
include("shapeFunctions.jl")
include("latticeProperties.jl")

using Ferrite
using SparseArrays
using LinearAlgebra
using Printf

#=
  This is a "wrapper" function to easily select which element assembly
  function to use. The "elementType" variable is in the global namespace
=#
function assembleElement( # Inputs
  relDense,          # Element relative density
  element,         # Element positions
  quad::QuadratureRules,  # Gauss quarature rules
  shape::ShapeFunctions2D)  # Shape functions

  # If heat equation
  if elementType == 0
    return assembleElementHeat!(relDense, element, quad, shape);

  # If structural problem
  elseif elementType == 1
    return assembleElementSolid(relDense, element, quad, shape);
  end
end

#=
  This function returns the 2D material matrix for a structural problem.

  It will be extended in the future to use the material properties of the
  selected unit cell and the input will be the porosity of the cell.
=#
function getDmatrix(reldense)
  if useLattice == true
    r = determineRadius(reldense2volfrac(reldense^p));
    D = getElasticityMatrix(r);
  else
    nu = 0.3;
    Eeval = 100.0 * reldense^p;
    Eeval3 = 100.0 * reldense^p;
    D = (1 / (1 - nu^2)) * [Eeval; Eeval*nu; 0;; Eeval3*nu; Eeval3; 0;; 0; 0; Eeval*0.5*(1-nu)]
  end
  return D
end


#=
  This is a typical finite element assembly function. It is specifically for
  4 noded finite elements. It assembles the element matrix for a structual
  problem.

  This function returns the finite element matrix and forcing array (Ke, fe)
  as well as KeSave which is used in the sensitivity analysis in the optimise
  topology function
=#
function assembleElementSolid(# Inputs
  relDense,          # Element relative density
  element,         # Element index
  quad::QuadratureRules,  # Gauss quarature rules
  shape::ShapeFunctions2D)  # Element shape functions


  # Initialise
  fe = zeros(8);
  Ke = zeros(typeof(relDense),8,8);
  dKe = zeros(typeof(relDense), 8,8);
  D = DiffResults.DiffResult(Matrix{Float64}(undef, 3,3), Matrix{Float64}(undef, 3,3));

  # Compute Elemental Matrix and Vector
  B = zeros(3,8);
  ForwardDiff.derivative!(D, getDmatrix, relDense);

  # Loop over quadrature points
  for q = 1:quad.nQuadPoints
    detJ, dNdX = getJacobian(q, shape, element)
    getBmatrix!(B,dNdX[1,:], dNdX[2,:])

    integrator = quad.weights[q] * detJ
```

```
    Ke += integrator .* (B' * D.value * B)
    dKe += integrator .* (B' * DiffResults.derivative(D) * B)
  end

  return Ke, dKe, fe;
end;


#=
  This Function will be modified in the future to get the ETC of the element
  taking into account which unit cell it belongs to
=#
function getKmatrix(reldense)
  if useLattice == true
    r = determineRadius(reldense2volfrac(reldense^p));
    D = getConductivityMatrix(r);
  else
    kMax = 160.0;
    kMin = 25e-3;
    kEval = (kMax - kMin) * reldense^p + kMin
    D = [kEval; 0;; 0; kEval];
  end
  return D;
end;


#=
  This is a typical finite element assembly function. It is specifically for
  4 noded finite elements. It assembles the element matrix using a conductivity
  and convection matrix.

  The reason the convection matrix (Kec) is include is because for the thermal
  topology optimisation to work correctly, the elements need to loss heat to
  the environment. The paper was based on natural convection

  This function returns the finite element matrix and forcing array (Ke, fe)
  as well as KeSave which is used in the sensitivity analysis in the optimise
  topology function
=#
function assembleElementHeat!(  # Inputs
    relDense,          # relative density array (length = nElements)
    element,         # Element node positions (size 4x2)
    quad::QuadratureRules,  # Quadrature rules
    shape::ShapeFunctions2D)  # Shape functions


  # Initialise
  fe = zeros(4);
  Ke = zeros(typeof(relDense),4,4);
  dKe = zeros(typeof(relDense), 4,4);
  D = DiffResults.DiffResult(Matrix{Float64}(undef, 2,2), Matrix{Float64}(undef, 2,2));

  # Get material properties
  h = 2.0;
  ForwardDiff.derivative!(D, getKmatrix, relDense);

  # Compute Elemental Matrix and Vector
  for q = 1:quad.nQuadPoints
    detJ, dNdX = getJacobian(q, shape, element) # Get detminant of jacobian and shape function gradients

    integrator = quad.weights[q] * detJ

    fe += (integrator * 25.0 * h) .* shape.N[q,:];
    Ke += integrator .* ((dNdX' * D.value * dNdX) + (shape.N[q,:] * shape.N[q,:]'));
    dKe += integrator .* (dNdX' * DiffResults.derivative(D) * dNdX);
  end

  return Ke, dKe, fe;
end;


#=

=#
function assembleElementPC(element, reldense, quad, shape)
  Ke = zeros(4,4);
  Ce = zeros(4,4);
  fe = zeros(4);

  D = getKmatrix(reldense)
  for q = 1:quad.nQuadPoints
    detJ, dNdX = getJacobian(q, shape, element);
    integrator = quad.weights[q] * detJ

    rho, cp, _ = getRhoCpL(reldense);
```

```
    fe .+= zeros(4);
    Ke .+= integrator .* (dNdX' * D * dNdX);
    Ce .+= (integrator * rho * cp) .* (shape.N[q,:] * shape.N[q,:]');
  end

  return Ke, Ce, fe;
end;

#=

=#
function globalAssemblyPC!(ncon, coords, reldense, K, C, F, quad, shape)
  fill!(K, 0.0); fill!(C, 0.0); fill!(F, 0.0);

  assemblerK = start_assemble(K);
  assemblerC = start_assemble(C);

  for e = 1:nElements
    elePoints = coords[ncon[e],:]
    Ke, Ce, fe = assembleElementPC(elePoints, reldense[e], quad, shape)

    elementMap = ncon[e]
    assemble!(assemblerK, elementMap, Ke);
    assemble!(assemblerC, elementMap, Ce);
    F[elementMap] += fe;
  end
end

#=

=#
## Gets the fraction of the phase based on its temperature
function getPhaseFraction(temp)
  a = 1.25;
  if (temp < tm - tr)
    return 0.0, 0.0;
  elseif (temp > tm + tr)
    return 1.0, 0.0;
  else
    s = 1 / (1 + exp(-a * (temp-tm)));
    ds = (a * exp(-a * (temp-tm))) / (1 + exp(-a * (temp-tm)))^2;
    return s, ds;
  end
end;
function getdds(temp)
  a = 1.25;
  dds = (2*a^2*exp(-2*a*(temp - tm)))/(exp(-a*(temp - tm)) + 1)^3 - (a^2*exp(-a*(temp - tm)))/(exp(-a*(temp - tm)) + 1)^2;
  return dds;
end;

## Gets only the Latent Heat vector
function getLatentHeatVector(ncon, reldense, elements, temp)
  L = Vector{typeof(temp[1])}(undef,nNodes);
  fill!(L, 0.0);
  for e = 1:nElements
    rho, _, LH =  getRhoCpL(reldense[e])
    Le = zeros(typeof(reldense[1]),4);                # Init element matrix
    for i = 1:4
      pf, _ = getPhaseFraction(temp[ncon[e][i]]);     # Get phase fraction
      Le[i] += (rho * LH * pf) .* elements[e].fe[i];    # Compute element latent heat vector
    end

    # Assemble to global latent heat vector
    elementMap = ncon[e];
    L[elementMap] += Le;
  end

  return L;
end;

## Iterates the latent heat vector given a new temperature
function iterateLatentHeat!(ncon, reldense, elements, dL, L, temp)
  # Reset matrix and vector
  fill!(L, 0.0);
  fill!(dL, 0.0);

  assembler = start_assemble(dL, L);
  for e = 1:nElements
    rho, _, LH = getRhoCpL(reldense[e]);
    Le = zeros(typeof(reldense[e]), 4);
    dLe = zeros(typeof(reldense[e]), 4,4);
    for i = 1:4
      pf, dpf = getPhaseFraction(temp[ncon[e][i]]);
      Le[i] = (rho * LH * pf) .* elements[e].fe[i];
      dLe[i,:] = (rho * LH * dpf) .* elements[e].Me[i,:];
```

```
    end

    # Assemble to global matrix and vector
    elementMap = ncon[e]
    assemble!(assembler, elementMap, dLe, Le);
  end
end;


#=
  This is a typical global assemble for finite element matrices.
=#
function globalAssembly!(    # Inputs
    relDense,          # relative density array (length = nElements)
    ncon,              # Node connectivity array (size = nElements x nElementNodes)
    KeSave,            # Will be returned for sensitivity analysis
    K::SparseMatrixCSC,    # Global stiffness matrix
    f::Vector{Float64},    # Global RHS
    quad::QuadratureRules,  # Gauss quadrature rules
    shape::ShapeFunctions2D,  # Element shape functions
    nNodalDofs)        # Number of nodal degrees of freedom (1 for heat 2 for solid)


  nElements = size(ncon)[1]

  # Use Ferrite assembler for efficiency
  assembler = start_assemble(K, f);

  # Initialise system matrix/vector to 0
  fill!(f, 0.0);
  fill!(K, 0.0);

  # Assemble Linear System
  for e = 1:nElements
    element = pos[ncon[e],1:2];
    Ke, dKe, fe = assembleElement(relDense[e], element, quad, shape);
    KeSave[:,:,e] .= dKe;

    # Maps the element connectivity to the global matrix indices
    elementMap = zeros(Int64, nNodalDofs * shape.nNodes)
    for i = 1:shape.nNodes
      for j = 1:nNodalDofs
        elementMap[nNodalDofs * i - (nNodalDofs - j)] = nNodalDofs * ncon[e][i] - (nNodalDofs - j)
      end
    end

    # Assemble element contributions to global matrix
    assemble!(assembler, elementMap, Ke, fe);
  end
end;

#=
  This function handles the Dirichlet BCs by setting the diagonal of the
  stiffness matrix to 1 and the row to 0. It then sets f[dirichletNodes]
  to the value supplied.

  The DOF enum is for selecting which DOF to constrain
=#
@enum DOF begin
  xDof = 1 << 0;
  yDof = 1 << 1;
  zDof = 1 << 2;
  all = (1 << 0) | (1 << 1) | (1 << 2);

  xyDof = (1 << 0) | (1 << 1);
  yzDof = (1 << 1) | (1 << 2);
  xzDof = (1 << 2) | (1 << 0);

end
function handleDirichletBCs!( # Inputs
    constraints::Vector,  # The nodes to be constrained
    dof::DOF,          # Constrained DOF
    value,             # Dirichlet values (size = nNodalDofs x 1)
    K,              # Global stiffness matrix
    f,              # Global RHS
    nNodalDofs)        # Number of nodal DOFs

  nNodes = size(K)[1]

  # Loop over each constraints
  for c in constraints
    d1 = nNodalDofs * c - (nNodalDofs - 1)  # If 1d, this evaluates to c, else 2*c-1 else 3*c-2
    d2 = nNodalDofs * c - (nNodalDofs - 2)  # If 1d, this is irrelevant, else 2*c else 3*c-1
    d3 = nNodalDofs * c - (nNodalDofs - 3)  # If 1d, this is irrelevant, else this is irrelevant else 3*c
    for i = 1:nNodes
      if (Int(dof) & Int(xDof)) != 0; K[d1, i] = 0.0  end
```

```
      if (Int(dof) & Int(yDof)) != 0; K[d2, i] = 0.0; end
      if (Int(dof) & Int(zDof)) != 0; K[d3, i] = 0.0; end
    end
    if (Int(dof) & Int(xDof)) != 0
      K[d1, d1] = 1.0
      f[d1] = value[1]
    end
    if (Int(dof) & Int(yDof)) != 0
      K[d2, d2] = 1.0
      f[d2] = value[2]
    end
    if (Int(dof) & Int(zDof)) != 0
      K[d3, d3] = 1.0
      f[d3] = value[3]
    end
  end
end;

#=
  This function handles the Neumann BCs by using a linear finite elements
  Im not 100% sure its implimented correctly, but it seems to give the correct
  results when I compared to other libraries
=#
function handleNeumannBCs!(
    constraints,     # Node connectivity of the constrained elements
    pos,         # Nodal coordinates
    value,         # Neumann BC value (size = nNodalDofs x 1)
    f,           # Global RHS
    nNodalDofs)      # Number of nodal DOFs


  nElements = size(constraints)[1]
  fe = zeros(2 * nNodalDofs)
  for e = 1:nElements
    idx = constraints[e]
    elePoints = pos[idx,1:2]
    eleLength = norm(elePoints[2,:] - elePoints[1,:]) # weight * det(J) = eleLength (2 * L/2 = L)

    N = eleLength * 0.5
    elementMap = zeros(Int32, 2 * nNodalDofs)
    for i = 1:2
      for j = 1:nNodalDofs
        elementMap[nNodalDofs * i - (nNodalDofs - j)] = nNodalDofs * idx[i] - (nNodalDofs - j)
        fe[nNodalDofs * i - (nNodalDofs - j)] = value[j] * N
      end
    end

    f[elementMap] += fe
  end
end;

#=
  This function gets the sparcity pattern of the global matrix based on
  the element connectivity and then returns a sparse matrix.
=#
function getSparsityPatternHeat(ncon)
  nElements = size(ncon)[1]
  nDofs = maximum(maximum.(ncon))

  # Create temporary arrays to store which elements are non-zero
  indexI = [Vector{Int64}(undef,0) for _ = 1:nDofs] # Non-zeros in the rows
  indexJ = [Vector{Int64}(undef,0) for _ = 1:nDofs] # Non-zeros in the columns
  for e = 1:nElements
    element = ncon[e]
    for r in element
      for c in element
        indexI[r] = [indexI[r]; r]  # Add the non-zeros to the relevant array
        indexJ[r] = [indexJ[r]; c]  # Add the non-zeros to the relevant array
      end
    end
  end

  indexI = collect(Iterators.flatten(indexI)) # Flatten vectors into 1D arrays for constructing sparse matrix
  indexJ = collect(Iterators.flatten(indexJ)) # Flatten vectors into 1D arrays for constructing sparse matrix
  values = zeros(Float64, length(indexI))   # Create a values array of 0's
  return sparse(indexI, indexJ, values), Vector{Float64}(undef, nDofs)  # Return the sparce matrix and full RHS
end;
function getSparsityPatternSolid(ncon)
  nElements = size(ncon)[1]
  nDofs = 2 * maximum(maximum.(ncon))

  indexI = [Vector{Int64}(undef,0) for _ = 1:nDofs]
  indexJ = [Vector{Int64}(undef,0) for _ = 1:nDofs]
  for e = 1:nElements
    element = ncon[e]
    for r in element
```

```
      for c in element
        indexI[2*r] = [indexI[2*r]; [2*r;2*r]]
        indexJ[2*r] = [indexJ[2*r]; [2*c;2*c-1]]
        indexI[2*r-1] = [indexI[2*r-1]; [2*r-1;2*r-1]]
        indexJ[2*r-1] = [indexJ[2*r-1]; [2*c;2*c-1]]
      end
    end
  end
  indexI = collect(Iterators.flatten(indexI))
  indexJ = collect(Iterators.flatten(indexJ))
  values = zeros(Float64, length(indexI))
  return sparse(indexI, indexJ, values), Vector{Float64}(undef, nDofs)
end;


# This isnt really relevant
function getNeighbours(ncon)
  nNodes = maximum(maximum.(ncon))
  nElements = size(ncon)[1]
  connectedNodes = [Vector{Int64}(undef,0) for _ = 1:nNodes]  # Stores which elements are connected to node i

  for e = 1:nElements
    nodes = ncon[e]
    for node in nodes
      connectedNodes[node] = [connectedNodes[node]; e]
    end
  end

  neighbours = [Vector{Int64}(undef,0) for _ = 1:nElements] # Stores which elements neighbour each element

  for n = 1:nNodes
    for element in connectedNodes[n]
      indices = findall(!=(element), connectedNodes[n])
      for index in indices
        neighbours[element] = [neighbours[element]; connectedNodes[n][index]]
      end
    end
  end

  for e = 1:nElements
    neighbours[e] = unique(neighbours[e])
  end
  return neighbours
end


#=
  This computes the weight factor matrix for filtering the element
  sensitivities.

  This is constructed by finding the distance between each element, and
  storing the distance in a sparse matrix. The filter radius defines
  how far sensititivy is filtered. Only the positive r - distance values
  are stored

  This distances are the distance between element centroids
=#
function getWeightFactorMatrix(ncon, pos, radius::Float64)
  nElements = size(ncon)[1]
  centroids = zeros(Float64, (nElements, 2))

  # Compute element centroids
  for e = 1:nElements
    element = pos[ncon[e],1:2]
    centroid = sum(element, dims=1) ./ 4
    centroids[e,:] = centroid
  end

  # Compute weight factor matrix
  indexI = [Vector{Int64}(undef,0) for _ = 1:nElements]
  indexJ = [Vector{Int64}(undef,0) for _ = 1:nElements]
  values = [Vector{Float64}(undef,0) for _ = 1:nElements]
  Threads.@threads for i = 1:nElements
    for j = 1:nElements
      distance = norm(centroids[i,:] - centroids[j,:])
      if distance < radius
        indexI[i] = [indexI[i]; i]
        indexJ[i] = [indexJ[i]; j]
        values[i] = [values[i]; radius - distance]
      end
    end
  end

  indexI = collect(Iterators.flatten(indexI))
  indexJ = collect(Iterators.flatten(indexJ))
  values = collect(Iterators.flatten(values))
```

```
  return sparse(indexI, indexJ, values)
end;
```

fileio.jl

This file handles all the file input/output, such as importing the mesh and exporting results to .vtu or .pvd files.

```
using Printf

## ----- Export Time Dependant Results ----- ##
function initExport2pvd(filepath, points, ncon, timestep, sol = 0.0, cellData = 0.0)
  pvdfile = open(filepath, "w");
  write(pvdfile, "<?xml version=\"1.0\"?>\n")
  write(pvdfile, "<VTKFile type=\"Collection\" version=\"1.0\">\n");
  write(pvdfile, "\t<Collection>\n")

  folder = collect(eachsplit(filepath,"."))[1];
  relfolder = collect(eachsplit(folder,"/"))[end];
  timestep_string = collect(eachsplit(string(timestep),"."));
  vtkpath = []; relpath = [];
  if size(timestep_string)[1] == 1
    vtkpath = folder * "_results/ts_" * timestep_string[1] * ".vtu";
    relpath = relfolder * "_results/ts_" * timestep_string[1] * ".vtu";
  else
    vtkpath = folder * "_results/ts_" * timestep_string[1] * "_" * timestep_string[2] * ".vtu";
    relpath = relfolder * "_results/ts_" * timestep_string[1] * "_" * timestep_string[2] * ".vtu";
  end

  try
    readdir(folder * "_results");
  catch
    mkdir(folder * "_results");
  end

  dataline = Printf.@sprintf("\t\t<DataSet timestep=\"%.6f\" part=\"0\" file=\"%s\" />\n", timestep, relpath);
  export2vtk(vtkpath, points, ncon, sol, cellData);

  write(pvdfile, dataline);
  write(pvdfile, "\t</Collection>\n");
  write(pvdfile, "</VTKFile>");
  close(pvdfile);
end;

function export2pvd(filepath, points, ncon, timestep, sol = 0.0, cellData = 0.0)
  pvdfile = open(filepath, "a+");
  seekstart(pvdfile);
  lines = readlines(pvdfile);
  close(pvdfile);
  pvdfile = open(filepath, "w");

  nlines = size(lines)[1];
  first = 0; last = 0;
  for i = 1:nlines
    if occursin("DataSet", lines[i])
      first = i;
      break;
    end
  end
  for i = nlines:-1:1
    if occursin("DataSet", lines[i])
      last = i;
      break;
    end
  end

  write(pvdfile, "<?xml version=\"1.0\"?>\n")
  write(pvdfile, "<VTKFile type=\"Collection\" version=\"1.0\">\n");
  write(pvdfile, "\t<Collection>\n")

  if first > 0
    for i = first:last
      lines[i] = lines[i] * "\n"
      write(pvdfile, lines[i])
    end
  end

  folder = collect(eachsplit(filepath,"."))[1];
  relfolder = collect(eachsplit(folder,"/"))[end];
```

```
  timestep_string = collect(eachsplit(string(timestep),"."));
  vtkpath = []; relpath = [];
  if size(timestep_string)[1] == 1
    vtkpath = folder * "_results/ts_" * timestep_string[1] * ".vtu";
    relpath = relfolder * "_results/ts_" * timestep_string[1] * ".vtu";
  else
    vtkpath = folder * "_results/ts_" * timestep_string[1] * "_" * timestep_string[2] * ".vtu";
    relpath = relfolder * "_results/ts_" * timestep_string[1] * "_" * timestep_string[2] * ".vtu";
  end

  try
    readdir(folder * "_results");
  catch
    mkdir(folder * "_results");
  end
  dataline = Printf.@sprintf("\t\t<DataSet timestep=\"%.6f\" part=\"0\" file=\"%s\" />\n", timestep, relpath);
  export2vtk(vtkpath, points, ncon, sol, cellData);

  write(pvdfile, dataline);
  write(pvdfile, "\t</Collection>\n");
  write(pvdfile, "</VTKFile>");
  close(pvdfile);
end;

## ----- Export to VTK function ----- ##
function export2vtk(filepath, points, ncon, sol = 0.0, cellData = 0.0)
  file = open(filepath, "w");
  write(file, "<?xml version=\"1.0\" encoding=\"utf-8\"?>\n");
  write(file, "<VTKFile type=\"UnstructuredGrid\" version=\"1.0\" byte_order=\"LittleEndian\" header_type=\"UInt64\"
          compressor=\"vtkZLibDataCompressor\">\n");
  write(file, "\t<UnstructuredGrid>\n");

  nNodes = size(points)[1];
  nElements = size(ncon)[1];
  write(file, @sprintf("\t\t<Piece NumberOfPoints=\"%i\" NumberOfCells=\"%i\">\n", nNodes, nElements));

  # Write node position data
  write(file, "\t\t\t<Points>\n");
  write(file, "\t\t\t\t<DataArray type=\"Float64\" Name=\"Points\" NumberOfComponents=\"3\" format=\"ascii\">\n");
  for i = 1:nNodes
    if size(points)[2] == 2
      write(file, @sprintf("\t\t\t\t\t%f %f %f\n", points[i,1], points[i,2], 0.0));
    elseif size(points)[2] == 3
      write(file, @sprintf("\t\t\t\t\t%f %f %f\n", points[i,1], points[i,2], points[i,3]));
    end
  end
  write(file, "\t\t\t\t</DataArray>\n");
  write(file, "\t\t\t</Points>\n");


  # Write connectivity data
  write(file, "\t\t\t<Cells>\n");
  write(file, "\t\t\t\t<DataArray type=\"Int64\" Name=\"connectivity\" NumberOfComponents=\"1\" format=\"ascii\">\n");
  for i = 1:nElements
    if length(ncon[i]) == 2
      write(file, @sprintf("\t\t\t\t\t%i %i\n", ncon[i][1]-1, ncon[i][2]-1));
    elseif length(ncon[i]) == 4
      write(file, @sprintf("\t\t\t\t\t%i %i %i %i\n", ncon[i][1]-1, ncon[i][2]-1, ncon[i][3]-1, ncon[i][4]-1));
    elseif length(ncon[i]) == 1
      write(file, @sprintf("\t\t\t\t\t%i\n", ncon[i][1]));
    end
  end
  write(file, "\t\t\t\t</DataArray>\n");

  # Write element offset information
  write(file, "\t\t\t\t<DataArray type=\"Int64\" Name=\"offsets\" NumberOfComponents=\"1\" format=\"ascii\">\n");
  counter = 0
  for i = 1:nElements
    if length(ncon[i]) == 2
      write(file, @sprintf("\t\t\t\t\t%i\n", counter+2));
      counter += 2
    elseif length(ncon[i]) == 4
      write(file, @sprintf("\t\t\t\t\t%i\n", counter+4));
      counter += 4
    elseif length(ncon[i]) == 1
      write(file, @sprintf("\t\t\t\t\t%i\n", counter+1));
      counter += 1
    end
  end
  write(file, "\t\t\t\t</DataArray>\n");

  # Write element type information
  write(file, "\t\t\t\t<DataArray type=\"UInt8\" Name=\"types\" NumberOfComponents=\"1\" format=\"ascii\">\n");
  for i = 1:nElements
    if length(ncon[i]) == 2
      write(file, @sprintf("\t\t\t\t\t%i\n", 3));
```

```
    elseif length(ncon[i]) == 4
      write(file, @sprintf("\t\t\t\t\t%i\n", 9));
    elseif length(ncon[i]) == 1
      write(file, @sprintf("\t\t\t\t\t%i\n", 1));
    end
  end
  write(file, "\t\t\t\t</DataArray>\n");
  write(file, "\t\t\t</Cells>\n");

  # Write solution data
  if length(sol) > 1
    nDofs = Int(length(sol) / nNodes);
    write(file, "\t\t\t<PointData>\n");
    write(file, @sprintf("\t\t\t\t<DataArray type=\"Float64\" Name=\"u\" NumberOfComponents=\"%i\" format=\"ascii\">\n", nDofs
        ));
    write(file, @sprintf("\t\t\t\t\t"));
    for i = 1:nNodes
      for j = 1:nDofs
        write(file, @sprintf("%f ", sol[nDofs * i - (nDofs - j)]))
      end
    end
    write(file, "\n")
    write(file, "\t\t\t\t</DataArray>\n");
    write(file, "\t\t\t</PointData>\n");
  end
  if length(cellData) > 1
    write(file, "\t\t\t<CellData>\n");
    write(file, "\t\t\t\t<DataArray type=\"Float64\" Name=\"Density\" NumberOfComponents=\"1\" format=\"ascii\">\n");
    counter = 1
    for i = 1:nElements
      if length(ncon[i]) == 2
        write(file, @sprintf("\t\t\t\t\t%f\n", 0.0));
      elseif length(ncon[i]) == 1
        write(file, @sprintf("\t\t\t\t\t%f\n", 0.0));
      elseif length(ncon[i]) == 4
        write(file, @sprintf("\t\t\t\t\t%f\n", cellData[counter]));
        counter += 1
      end
    end
    write(file, "\t\t\t\t</DataArray>\n");
    write(file, "\t\t\t</CellData>\n");
  end

  write(file, "\t\t</Piece>\n");
  write(file, "\t</UnstructuredGrid>\n");
  write(file, "</VTKFile>\n");
  close(file);
end;

## ----- Read RelDense from VTL ----- ##
function readReldenseVTK(filepath)
  file = open(filepath);
  nElements = 0;
  reldense = [];
  while !eof(file)
    line = readline(file);
    if contains(line, "NumberOfCells")
      idx = findfirst("NumberOfCells",line)
      substr = line[idx[end]+1:end];
      idx1 = findfirst("\"", substr);
      idx2 = findnext("\"", substr, idx1[1]+1);
      nElements = parse(Int64, substr[idx1[1]+1:idx2[1]-1]);
    end
    if contains(line, "Density") && nElements > 0;
      reldense = zeros(nElements);
      line = readline(file);
      counter = 1;
      while !contains(line, "</DataArray>")
        line = lstrip(line);
        line = rstrip(line);
        if !isempty(line)
          reldense[counter] = parse(Float64, line);
          counter += 1;
        end
        line = readline(file);
      end
    end
  end

  close(file);

  return reldense;
end;

## ----- Mesh import functions ----- ##
function importMsh(filepath)
```

```
file = open(filepath, "r")

# Preallocate arrays and maps
ncon = Vector{Vector{Int64}}([])
pos = []
groups = Dict()
entityMap = Vector{Vector{Int}}([])
elementTags = Vector{Vector{Int}}([])

# Read through file
while true
  line = readline(file)

  # Read mesh format
  if contains(line, "MeshFormat")
    version, type, _ = parseMeshFormat(file)
    if version != 4.1
      @printf "[Error] Version not supported! : %.1f" version
      break
    end
    if type == true
      @printf "[Error] File type not supported! : %i" type
      break
    end

  # Get physical groups
  elseif contains(line, "PhysicalNames")
    groups = parsePhysicalNames(file)

  # Get mesh entities
  elseif contains(line, "Entities")
    entityMap = parseEntities(file)

  # Read node positions
  elseif contains(line, "Nodes")
    pos = parseNodes(file)

  # Read element connectivity
  elseif contains(line, "Elements")
    ncon, elementTags = parseElements(file)

  # If EOF reached
  elseif isempty(line)
    break

  # Ignore other blocks
  else
    @printf "[Error] Block not recognised! : %s" line
  end
end

# Process the groups to get element tags in one array
boundaryElements = Dict()
for key in keys(groups)
  group, dim = groups[key]
  id = []

  # Loop through entity map
  for i = 1:size(entityMap)[1]
    if dim == entityMap[i][1]
      # Loop through groups entity is assigned to
      for j = 3:size(entityMap[i])[1]
        if group == entityMap[i][j]
          push!(id, entityMap[i][2])
        end
      end
    end
  end

  # Loop through elementTags to find
  idx = []
  for i = 1:size(elementTags)[1]
    if dim == elementTags[i][1]
      for j = 1:size(id)[1]
        if id[j] == elementTags[i][2]
          push!(idx, i)
        end
      end
    end
  end

  # Concat all elements to one groups
  elements = []
  for i = 1:size(idx)[1]
    elements = [elements; elementTags[idx[i]][3:end]]
  end
```

```
    boundaryElements[key] = elements
  end

  boundaryNodes = Dict()
  for key in keys(boundaryElements)
    _, dim = groups[key]
    nElements = length(ncon[boundaryElements[key]])
    nconTmp = ncon[boundaryElements[key]]
    nodes = []

    # This is a naive implimentation that doesn't consider the element order
    if dim == 0
      nodes = Vector{Int64}(undef, nElements)
      for i = 1:nElements
        nodes[i] = nconTmp[i][1]
      end
    elseif dim == 1
      nodes = Vector{Int64}(undef, 2*nElements)
      for i = 1:nElements
        nodes[2*i-1] = nconTmp[i][1]
        nodes[2*i] = nconTmp[i][2]
      end
    elseif dim == 2
      nodes = Vector{Int64}(undef, 4*nElements)
      for i = 1:nElements
        nodes[4*i-3] = nconTmp[i][1]
        nodes[4*i-2] = nconTmp[i][2]
        nodes[4*i-1] = nconTmp[i][3]
        nodes[4*i] = nconTmp[i][4]
      end
    end
    boundaryNodes[key] = unique(nodes)
  end

  return ncon, pos, boundaryElements, boundaryNodes
end

function parseMeshFormat(file)
  # Preallocate return values
  version = 0.0
  type = false
  dataSize = 0

  # Parse block
  while true
    line = readline(file)
    if contains(line, '$')
      break;
    end

    parts = split(line)
    version = parse(Float64, parts[1])
    type = parse(Bool, parts[2])
    dataSize = parse(Int64, parts[3])
  end

  return version, type, dataSize
end;

function parsePhysicalNames(file)
  # Preallocate map
  map = Dict()

  # Parse block
  line = readline(file)
  n = parse(Int64, line)
  for i = 1:n
    parts = split(readline(file))
    dim = parse(Int64, parts[1])
    val = parse(Int64, parts[2])
    name = parts[3][2:end-1]
    map[name] = [val dim]
  end
  if !contains(readline(file), '$')
    @printf "[Error] End of block not reached!"
  end

  return map
end;

function parseEntities(file)
  parts = split(readline(file))
  nPoints = parse(Int64, parts[1])
  nCurves = parse(Int64, parts[2])
  nSurfaces = parse(Int64, parts[3])
  nVolumes = parse(Int64, parts[4])
```

```
  map = Vector{Vector{Int}}([])

  # Parse points
  for i = 1:nPoints
    parts = split(readline(file))
    tag = parse(Int64, parts[1])
    nPhysicalTags = parse(Int64, parts[5])
    if nPhysicalTags > 0
      tmpArr = zeros(nPhysicalTags + 2)
      tmpArr[1] = 0
      tmpArr[2] = tag
      for j = 1:nPhysicalTags
        tmpArr[j+2] = parse(Int64, parts[5+j])
      end
      push!(map, tmpArr)
    end
  end

  # Parse Curves
  for i = 1:nCurves
    parts = split(readline(file))
    tag = parse(Int64, parts[1])
    nPhysicalTags = parse(Int64, parts[8])
    if nPhysicalTags > 0
      tmpArr = zeros(nPhysicalTags + 2)
      tmpArr[1] = 1
      tmpArr[2] = tag
      for j = 1:nPhysicalTags
        tmpArr[j+2] = parse(Int64, parts[8+j])
      end
      push!(map, tmpArr)
    end
  end

  # Parse Surfaces
  for i = 1:nSurfaces
    parts = split(readline(file))
    tag = parse(Int64, parts[1])
    nPhysicalTags = parse(Int64, parts[8])
    if nPhysicalTags > 0
      tmpArr = zeros(nPhysicalTags + 2)
      tmpArr[1] = 2
      tmpArr[2] = tag
      for j = 1:nPhysicalTags
        tmpArr[j+2] = parse(Int64, parts[8+j])
      end
      push!(map, tmpArr)
    end
  end

  # Parse volumes
  for i = 1:nVolumes
    @printf "[Warning] Volumes not implimented!"
  end

  if !contains(readline(file), '$')
    @printf "[Error] End of block not reached!"
  end

  return map;
end;

function parseNodes(file)
  parts = split(readline(file))
  nEntities = parse(Int64, parts[1])
  nNodes = parse(Int64, parts[2])

  nodes = Matrix{Float64}(undef, nNodes, 3)
  for e = 1:nEntities
    parts = split(readline(file))
    nBlock = parse(Int64, parts[4])
    tmpArr = Vector{Int64}(undef, nBlock)
    for i = 1:nBlock
      tmpArr[i] = parse(Int64, readline(file))
    end
    for i = 1:nBlock
      pos = split(readline(file))
      for j = 1:3
        nodes[tmpArr[i],j] = parse(Float64, pos[j])
      end
    end
  end

  if !contains(readline(file), '$')
    @printf "[Error] End of block not reached!"
```

```
  end

  return nodes;
end;

function parseElements(file)
  parts = split(readline(file))
  nEntities = parse(Int64, parts[1])
  nElements = parse(Int64, parts[2])

  ncon = Vector{Vector{Int64}}(undef, nElements)
  map = Vector{Vector{Int64}}(undef, nEntities)
  for e = 1:nEntities
    parts = split(readline(file))
    dim = parse(Int64, parts[1])
    tag = parse(Int64, parts[2])
    type = parse(Int64, parts[3])
    nBlock = parse(Int64, parts[4])

    entitiesTemp = Vector{Int64}(undef, nBlock + 2)
    entitiesTemp[1] = dim;
    entitiesTemp[2] = tag;

    nNodePerElement = 0
    if type == 1
      nNodePerElement = 2
    elseif type == 3
      nNodePerElement = 4
    elseif type == 15
      nNodePerElement = 1
    else
      @printf "[Error] Element not supported!"
    end

    nconTemp = zeros(nNodePerElement)
    for i = 1:nBlock
      parts = split(readline(file))
      idx = parse(Int64, parts[1])
      entitiesTemp[i+2] = idx
      for j = 1:nNodePerElement
        nconTemp[j] = parse(Int64, parts[j+1])
      end
      ncon[idx] = nconTemp;
    end

    map[e] = entitiesTemp
  end

  if !contains(readline(file), '$')
    @printf "[Error] End of block not reached!"
  end

  return ncon, map
end;
```