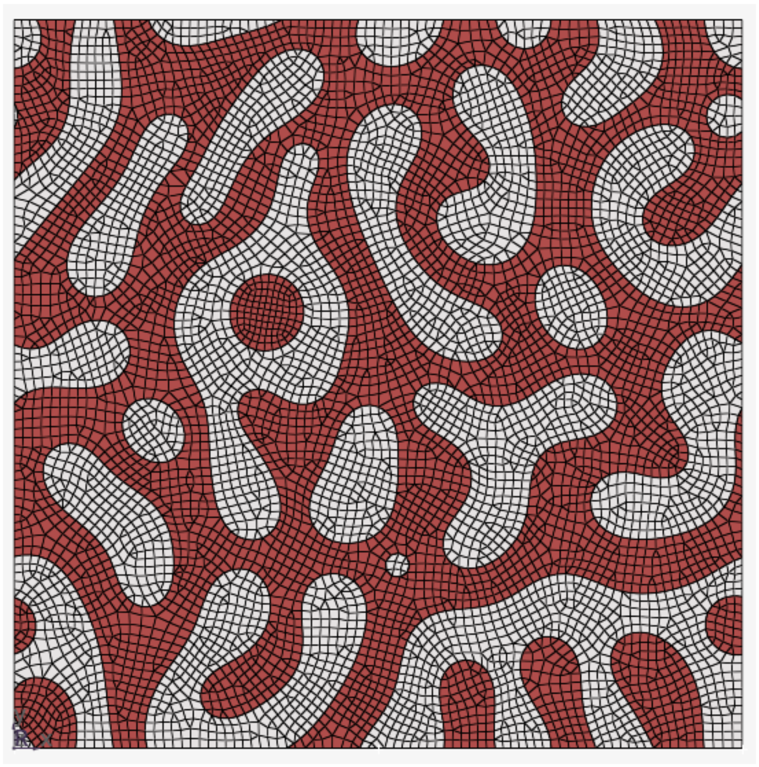
CHALMERS
UNIVERSITY OF TECHNOLOGY



# Deep material networks for lead-free solder alloys

**Master's thesis in Applied mechanics**

**Author:**
Gustav Juhlin Onbeck

**Department of Industrial and Materials Science**
**Chalmers university of technology**
Gothenburg, Sweden, 2023

**Deep material networks for lead-free solder alloys**
Gustav Juhlin Onbeck

Department of Industrial and Materials Science
Chalmers university of technology
Gothenburg, Sweden, 2023

**Cover:**
A finite element discretization of one of the morphologies used for dataset creation
**Sponsoring entity:**
Saab AB
**Saab supervisors:**
Luo Ruoshan
Erik Svenning

# Abstract

In this report the Deep Material Network (DMN) method is applied to a microstructural model for SAC-305, SnAgCu with 3 weight percent silver and 0.5 weight percent copper, generated by a phase field simulation for spinodal decomposition. The DMN method homogenizes a heterogeneous microstructure for multiscale simulation purposes. Five microstructures from the phase-field simulations were generated, and for each micro structure a data set describing the homogenized elastic parameters was calculated. The data sets were generated using finite element simulations for each morphology. The DMN method is modified to use individual weights for each node and a bias, the modifications improved the accuracy of the network and brings the method more inline with conventional neural network structures. The new DMN method was evaluated on data sets for each of the individual microstructures and on combinations of multiple microstructure data sets. DMNs trained on data sets of multiple microstructures resulted in a much greater error than DMNs trained on individual data sets. The DMNs trained on a single microstructure reached a relative error of 2.5 - 3 %, whereas a DMN trained on a multi-microstructure set could at best reach an error of 11.5 %.

**Keywords:**
Machine learning, deep material network, multi-scale simulations, lead-free solder alloys.

**Acknowledgments**
The author wants to acknowledge the support from his two supervisors at Saab, Ruoshan Luo and Erik Svenning for their help using Nastran MSC and Hyperworks during the dataset creation, as well as feedback and general help in the writing process. The author also wants to acknowledge his examiner Fredrik Larsson from Chalmers university of Technology for his insights and helpful meetings about multi-scale simulations and material homogenization, as well as report feedback.

# Contents

**1 Introduction** **1**
- 1.1 Literature study . . . 1
  - 1.1.1 Machine learning for finite element simulations . . . 1
  - 1.1.2 Generative networks . . . 1
  - 1.1.3 Neural network solver . . . 2
  - 1.1.4 Multi-scale simulation . . . 2
- 1.2 Project background . . . 3
- 1.3 Objective . . . 4
- 1.4 Limitations . . . 4

**2 Basic concepts** **5**
- 2.1 Machine learning fundamentals . . . 5
  - 2.1.1 Neural network . . . 6
  - 2.1.2 Forward pass . . . 6
  - 2.1.3 Cost function . . . 6
  - 2.1.4 Backwards propagation and gradient descent . . . 7
- 2.2 Homogenization of elastic parameters . . . 8
  - 2.2.1 Computational homogenization . . . 8
  - 2.2.2 Classical homogenization . . . 9

**3 Creating a dataset** **10**
- 3.1 Microstructure . . . 11
- 3.2 Simulations . . . 13
- 3.3 Validating dataset values . . . 14

**4 Deep material network** **17**
- 4.1 Forward-pass . . . 17
- 4.2 Weights . . . 19
  - 4.2.1 Traditional DMN structure . . . 20
  - 4.2.2 Conventional ML structure . . . 20
- 4.3 Bias . . . 21
- 4.4 Cost function . . . 22

4.5 Back propagation and learn step . . . . . . . . . . 22
4.5.1 Adaptive learning rate . . . . . . . . . . 25

**5 Results** **27**
5.1 DMN trained on single micro structure . . . . . . . . . . 28
5.2 Demonstration of temperature variance . . . . . . . . . . 30

**6 Discussion** **32**
6.1 New network structure . . . . . . . . . . 32
6.1.1 Bias . . . . . . . . . . 33
6.2 Multi-morphology datasets . . . . . . . . . . 34
6.3 Multi-scale simulations . . . . . . . . . . 35

**7 Conclusion and future research** **37**
7.1 Deep material networks for lead-free solder alloys . . . . . . . . . . 37
7.2 Future research . . . . . . . . . . 37

**A Gradients for back propagation** **42**

**B Supplementary results** **44**

**C Python code** **50**
C.1 Phasefield simulations . . . . . . . . . . 50
C.1.1 dxf file script . . . . . . . . . . 52
C.2 Dataset creation . . . . . . . . . . 53
C.2.1 Input phase generator . . . . . . . . . . 53
C.2.2 Phase class . . . . . . . . . . 55
C.2.3 Nastran file module . . . . . . . . . . 65
C.2.4 Main script . . . . . . . . . . 78
C.3 Deep material network . . . . . . . . . . 81
C.3.1 DMN class . . . . . . . . . . 81
C.3.2 Main script . . . . . . . . . . 92
C.3.3 Requirements . . . . . . . . . . 104

# 1
# Introduction

Artificial Intelligence (AI) and Machine Learning (ML) are hot topics at the moment and attempts to utilize it in new areas is the focus of a lot of research. AI can automate and reduce the time it takes to perform menial tasks or make impossibly time consuming tasks feasible. For this project AI/ML is researched in the field of solid mechanics/structural analysis simulations.

## 1.1 Literature study

A literature study was first performed on computational solid mechanics subjects. Three different parts of a solid mechanics simulation were the focus; defining the geometry with a generative network (CAD/mesh); material modeling for multi-scale simulations; solving the nodal displacements using a neural network for fine meshes.

### 1.1.1 Machine learning for finite element simulations

Finite element simulation is a numerical method derived from the discretization of a solid object into a mesh. The mesh defines a finite amount of volumes for which approximations can be applied to, such that the error when the mesh becomes impossibly fine, the approximation vanishes. Performing a simulation utilizing the Finite Element Method (FEM) requires work from an engineer. First the component, the subject of the simulation, needs to be digitally defined using Computer Aided Design (CAD). Next the CAD object is discretized into a mesh. The boundary conditions, loads, and material properties for the component is then set. A multitude of different types of simulations can be performed from this, depending on the material modeling, load conditions, and desired type of result.

### 1.1.2 Generative networks

Generative adversarial networks (GANs) [1] and diffusion models are the hottest topic in AI. These networks are capable of automating task previously believed impossible, to create new information. Generative networks such as ChatGPT or stable diffusion have been at the center of a lot of media attention in resent time. These new generative networks have pushed a lot of new research into this topic.

From the basis of these new generative networks, the question arise whether a generative network could be used for a finite element simulation. In the prep work before a simulation is performed, a digital version of the component needs to be created. Could a generative network not perform this step and drastically reduce the time to perform a FEM simulation? A generative network would then take a text input and give a CAD file as output, similar to how a text-to-image network functions. Some research is done in this field, to create a 3D model from a text prompt [2]. The current generative text-to-image models require hundreds of millions of samples in the datasets [3] [4]. These datasets are usually created by web crawlers, algorithms that take information from websites.

### 1.1.3 Neural network solver

The final step of a finite element simulation is to solve for the nodal displacement of the mesh. When a very fine mesh is used this step can be time consuming. Some research is being done on this topic, looking into the combination of a static FEM solver and a neural network for the very fine parts [5]. The neural network would be trained on a specific material and only applicable to that material.

### 1.1.4 Multi-scale simulation

Multi-scale simulations are a specific type of finite element simulation framework. This framework utilize information from a finer length scale in the process to solve a regular macro scale finite element simulation. Finite element square ($FE^2$) is the direct numerical reference method for multiscale simulations, a fine mesh is then representing the morphology of the microstructure. When a non-linear material is used in the material, newton iterations are performed for each element in the mesh and since each element is represented by a finer mesh, the newton iterations are performed for each of the finer meshes. This is why the method is called $FE^2$, since an entire second dimension is required [6]. Due to the exceptional computational requirements for this method, alternative methods are being researched.

One alternative way which is currently attracting greater interest is the use of AI/ML [7]. Depending on the material considered different approaches can be used to introduce AI/ML. One article research the use of a neural network for a bone structure. On the micro scale the bone material parameters change irregularly due to the stress, the neural network can predict the bone parameters for a macro scale element using the neural network and the macro scale stresses [8].

One new and interesting method has been researched lately and has been the focus of some research, the deep material network (DMN) method [9–12]. The DMN method homogenizes a heterogeneous micro structure to generate a material compliance matrix. The DMN method utilizes a novel machine learning structure which can be trained on linear elastic training data and later applied for a non-linear material modeling case.

## 1.2 Project background

Multi-scale simulation is the simulation framework to incorporate data from a smaller, finer length scale, in the process to solve a problem at a greater length scale. A multi-scale simulation is an extremely time consuming task when applied in the most direct approach. Solving on both the macro and micro or meso scale, requires an incredibly fine mesh or otherwise fine representation of the material to incorporate the fine length scales. In some cases of multi-scale simulations, generalizations can be used on the smaller scales to specifically consider only a portion of the micro or meso scale interactions, for example the finite element - fast Fourier transform (FE-FFT) method [13]. A different approach to multi-scale simulations is to allow a neural network to process the fine length scale interactions. The neural network is trained on data generated either from a high resolution representation of the micro-scale morphology in 2D or 3D, or from experimental data [14]. Depending on the implementation of the neural network, it can be considered a reduced order model (ROM).

One novel approach to implementing machine learning in a multi-scale simulation is to use a deep material network (DMN) [9]. DMNs have received a lot of focus lately as a method to model the homogenization of heterogeneous microstructure in materials. The DMN method has been used to model linear-elasticity, plasticity, hyper-elasticity, and composite materials from linear-elastic training data [10, 15].

In resent time, lead (Pb), has been phased out of every application possible, including as an additive in solder materials. The development of good Pb-free solder materials is very much in demand at the moment and a lot of research focus on accurately modeling the microstructure and its effect on mechanical properties [16–19]. One large group of Pb-free solder materials are Sn-Ag based materials. Tin and silver create an eutectic or hypo-eutectic phase, with an additive like Copper [16]. SACs, tin (Sn), silver (Ag), and copper (Cu) alloys, are one type of such an alloy. SACs can form tin dendrite branches in the microstructure during solidification for various SAC alloys [17, 18], but it can also form separate phases without nucleation called spinodal decomposition [20]. Dendrite branch formation and spinodal decomposition are phenomenons which are possible to simulate using phase-field simulations and therefore a good test case for the DMN method.

The solder-alloy's microstructure varies greatly throughout the material, in addition, different alloy compositions have very different microstructures, and the microstructure can change due to aging. Simply put, modeling solder materials with a single microstructures is not possible. The DMN model has showed great interpolation functionality for microstructures of fiber reinforced composites [10]. The interpolation functionality due to a large variety of microstructures for solder materials, would potentially be very useful. For this project the spinodal decomposition will be focused on and different micro structures generated using Cahn-Hilliard's equation [21]. The python scripts for phase-field simulations, dataset creation, and the deep material network is presented in Appendix C.

## 1.3 Objective

The main purpose of this project is to determine whether a multi morphology dataset is possible to use for the training of a DMN.
More specifically the objectives of this thesis is to:
- Create homogenization datasets using the finite element method for morphologies of the spinodal decomposition phenomenon.
- Train DMN models to homogenize the morphologies using linear elastic training data.
- Investigate the change in training response and final error for different datasets, network structures, and hyper parameter choices.

## 1.4 Limitations

The microstructures are only generated using a phase-field simulation for the spinodal decomposition phenomenon. The phase-field simulations are not meant to perfectly replicate the microstructure of a SAC-305 alloy, rather act as an example of a possible microstructure. Linear elastic, two-dimensional, static FEM simulations are used for the creation of all datasets.

The depth of the network is limited to allow for a reasonable training times. Previous works using DMNs indicated that a 8-layered network was the limit for training a network under 12 hours [9].

# 2

# Basic concepts

**Nomenclature**

**D**: Compliance matrix, inverse of material stiffness matrix.
$\bar{D}$: Homogenized compliance matrix.
$\bar{E}$: Homogenized stiffness matrix
$\bar{\sigma}$: volume averaged stress
$\bar{\epsilon}$: volume averaged strain
Epoch: An evaluation of the full dataset.
$\eta$: A learning rate, determines the step size for a parameter during training.
**w**: A weight, determines the ratio of signal/-compliance matrix from a node when homogenized with another node.
**b**: A bias, An added quantity to the signal from a node.
$\theta$: A rotation angle, rotates the coordinate system for a material node.

## 2.1 Machine learning fundamentals

Machine Learning (ML) is a subset of Artificial Intelligence (AI) algorithms. The simplest definition for what AI is, is that AI is an algorithm, system or similar, performing a task using information provided as input. The task can be anything from classifying or interpreting data, to creating new objects like texts or images. Sometimes the task to be performed can be analytically described using a sequence of functions, with predefined parameters.

When the task the algorithm performs becomes vague and difficult to analytically describe, then machine learning is the go to framework in AI. Machine learning systems are different in that they require a learning phase before use. The learning can be performed in many different ways depending on the AI's use case. One type of machine learning uses a data set of coupled data points as a means of learning. The coupled data points are the input data and the correct output data. The learning is simply to provide the system with the input data from the data set, let the system provide an output, and calculating an error based on the known correct value in the data set. The error is then provided to the system which performs some changes based on the errors to reduce the error during future evaluations. The exact way to perform any of these steps depend on the AI's use case and structure of the machine learning system.

### 2.1.1 Neural network

A neural network is a machine learning system which contains a set of layers. Some or all of the layers are composed of fully connected neurons which gives the network its name. The neurons are modeled after biological neurons in the brain and are therefore sometimes called artificial neurons and artificial neural networks. Neurons sends signals, typically modeled as real numbers or floats and never as higher dimensional entities.

The neural layers process the information by combining the input from its connected child neurons, the neurons in the previous layer closer to the input layer. The combination makes use of a function with a parameter which is related to or is the weight of that neuron. The weight is typically the parameter to weigh the sum of the inputs from the child neurons. The new value summed or otherwise derived from the child neurons is then activated. Activation means to apply an activation function on the signal, typically to deactivate a node when it starts sending negative values, and not changing the information at all if the data is above zero. This specific activation function is called a rectified linear unit or ReLU [22].

### 2.1.2 Forward pass

A neural network works on two main principles, a forward pass, and during training, a backwards propagation. When the network is fully trained the forward-pass of the network is the functionality of the network, to produce an output from input.

The data is fed into the input layer, the shape of the input layer is that of the data set. The information is then passed to the next layer which manipulates the data in some way using the current parameters of that layer. That layer passes it to the next layer using its current parameters and so on. The initial value of the parameters are typically randomly generated but they can be initialized from a pre-trained network using a method called transfer learning [23].

When the information has been passed to the output layer the shape of the information has changed to that of the desired output. If an image is to be classified for example, the output layer shape would be the amount of classifications possible, one node for each class.

### 2.1.3 Cost function

For the network to learn its performance must be quantified using a cost function. The purpose of the cost function is to accurately define the error of the current network.

A cost is provided every time a result is generated from the network. Typically the network will provide widely varying errors depending on the specific input from the data set. To make the cost function give a general indication of the networks current inaccuracy, multiple evaluations are combined and averaged to create a single cost value. The best indication of the networks current performance occurs when the entire data set is evaluated.

To process the entire data set once is called an epoch. The same dataset is processed each epoch and the thing that should change is the network. The order of the data set can be randomly shuffled each epoch to make sure the order of the data points is not impacting the training in some way but the data set remains the same.

During training the cost function typically evaluates a mini-batch of the data set, a smaller portion of it. This speeds up the training time since multiple cost values can be generated during a single epoch.

### 2.1.4 Backwards propagation and gradient descent

For each sample processed in the network a cost is calculated. Backwards propagation is the method to calculate a reasonable change for every parameter in the network by sending the cost backwards through the network using the functions and their current parameters as basis for their individual change.

$$\begin{aligned} f(NN,x) &= f(w_j^0, b_j^0(f(w_j^1, b_j^1(...(x))))) \\ C(y, f(NN,x)) &= C(y, f(w_j^0, b_j^0(f(w_j^1, b_j^1(...(x)))))) \end{aligned} \tag{2.1}$$

Equation (2.1) shows an example of a neural network function and its associated cost function. For this example w is a weight, b a bias, super-script denotes layer and sub-script neuron. y is the correct value from the dataset, if f(NN,x) is equal to y the cost is zero.

$$\frac{\partial C}{\partial w^i} = \frac{\partial C}{\partial a^0} \bullet \frac{\partial a^0}{\partial d^0} \bullet ... \bullet \frac{\partial d^i}{\partial w^i} \tag{2.2}$$

Given that the neural network can be described as a composition of functions, the cost functions gradient with respect to any parameter can be determined using the chain rule. Equation (2.2) shows the chain rule applied to the example from Equation (2.1), each individual gradient is a diagonal matrix with the dimension of the layer size. This is the most efficient way to back-propagate but the gradients can be calculated for individual neurons iteratively as well. The gradients calculate the local error for the network with respect to a layer and its intermediary functions and parameters, a and d correlates to the local results from these intermediary functions be it weights, biases or others. a and d would for a regular NN correspond to the signal from a neuron before and after activation for example. For the DMN method d and a correlates to intermediary material compliance components, after homogenization, after rotation, and after bias.

## 2.2 Homogenization of elastic parameters

The elastic parameters of a material determines the stress-strain relation for the material during loading. When multiple materials are combined, to make an alloy or composite for example, the elastic parameters in the material varies. The material is then heterogeneous. The material heterogeneity poses a problem when the material is used for calculations or finite element simulations. A homogenization of the elastic parameters is necessary, a single set of parameters which determines the effective elastic properties in the material. The resulting homogenized elastic parameters depends both on the quantity of the different phases, its weight percentage, but also on the micro structure, the forms the phases take in the material [24].

$$\bar{\sigma} = \bar{E} : \bar{\epsilon} \tag{2.3}$$

Equation (2.3) is the effective relation of linear elasticity. $\bar{E}$ is the effective material stiffness of the representative volume element (RVE).

### 2.2.1 Computational homogenization

Computational homogenization is the macroscale prediction of elasticity parameters by solving the discretized FEM problem for the sub-scale domain. Computational homogenization is the most precise way to predict the elasticity tensor given enough time and a good representation of the microstructure. A discretization of the microstructure into finite elements is used, each element takes either the material data from the first or second phase.

An important consideration when approximating the elasticity parameters for the RVE is the applied loads at the boundaries for the domain. The boundary conditions used are Dirichlet boundary conditions. The boundaries are displaced linearly in each of the $x$, $y$, and $xy$ directions providing three separate load cases.

$$\begin{aligned} u(x,y)^{\Gamma}_{x} &= x\bar{\epsilon}_x + y\bar{\epsilon}_{xy} \\ u(x,y)^{\Gamma}_{y} &= x\bar{\epsilon}_{xy} + y\bar{\epsilon}_y \end{aligned} \tag{2.4}$$

Three different loading conditions are applied using Equation (2.4). $u(x,y)^{\Gamma}$ is the coordinates along the boundary ($\Gamma$). One case with strictly displacement in the x direction, meaning $\bar{\epsilon}_x$ is non-zero and other strains are zero. One when $\bar{\epsilon}_y$ is non-zero and one when $\bar{\epsilon}_{xy}$ is non-zero. The displacement is solved from a static consideration. $\bar{\epsilon}$ is the macro strain for the micro structure, the volume averaged strain.

### 2.2.2 Classical homogenization

Classical homogenization produce upper and lower bounds for what the elasticity tensor could possibly be for a given material composition based on a guessed or generalized morphology.

The Reuss lower bound, is the assumption that the morphology is structured such that the sub-scale stress is uniform. The stress is for example uniform in one direction if the phases are structured similarly to the DMN building block Figure 4.2. This is also the assumption used to determine the elastic property of a composite across the direction of the fibers.

$$\begin{aligned} \sigma &= \bar{\sigma} \\ \bar{\epsilon} &= \bar{D}\bar{\sigma} \end{aligned} \tag{2.5}$$

Under these assumptions that the stress is uniform, the stress is then the same in both materials

$$\bar{\epsilon} = (D_1 * f)\bar{\sigma} + D_2 * (f - 1)\bar{\sigma} \tag{2.6}$$

$f$ is in Equation (2.6) the volume fraction of the first phase. By comparing with Equation (2.5) the compliance under a Reuss assumption is

$$\bar{D} = D_1 * f + D_2 * (f - 1) \tag{2.7}$$

Similarly the strain can instead be considered uniform. Assuming the strain field is uniform is the Voigt assumption and gives the upper limit for elastic parameters. Again this can be compared to the homogenization of elastic parameters in the fiber direction of a fiber reinforced composite.

$$\begin{aligned} \epsilon &= \bar{\epsilon} \\ \bar{C}\bar{\epsilon} &= \bar{\sigma} \end{aligned} \tag{2.8}$$

Since the strain is the same in both materials the volume averaged stress is

$$\bar{\sigma} = C_1 * f\bar{\epsilon} + C_2 * (f - 1)\bar{\epsilon} \tag{2.9}$$

Using the constitutive equation for the compliance Equation (2.5)

$$\bar{D} = (C_1 * f + C_2 * (f - 1))^{-1} \tag{2.10}$$

# 3

# Creating a dataset

This chapter details the method to generate a dataset to train a deep material network on. To create a dataset that correctly maps the function showed in Figure 3.1, a lot of repetitive work needs to be done. The flowchart to generate the dataset is shown in Figure 3.2

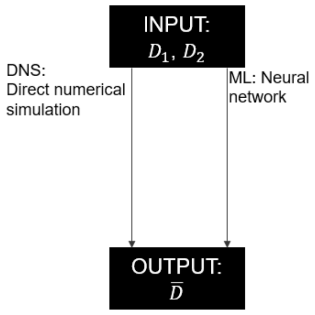


Figure 3.1: Function to approximate

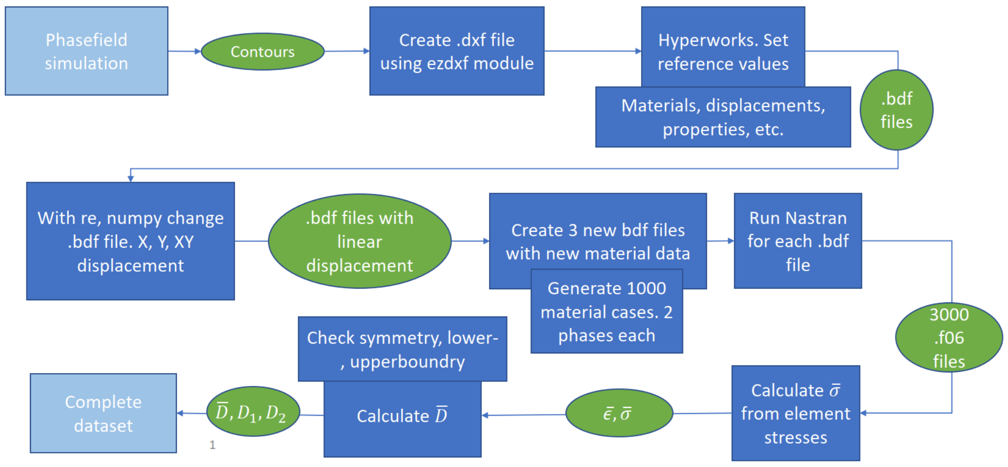

Phasefield simulation
Contours
Create .dxf file using ezdxf module
Hyperworks. Set reference values
Materials, displacements, properties, etc.
.bdf files
With re, numpy change .bdf file. X, Y, XY displacement
.bdf files with linear displacement
Create 3 new bdf files with new material data
Generate 1000 material cases. 2 phases each
Run Nastran for each .bdf file
3000 .f06 files
Calculate $\bar{\sigma}$ from element stresses
$\bar{\epsilon}, \bar{\sigma}$
Calculate $\bar{D}$
Check symmetry, lower-, upperboundry
$\bar{D}, D_1, D_2$
Complete dataset
1

Figure 3.2: Flowchart to generate a dataset to map the function in Figure 3.1

## 3.1 Microstructure

As mentioned in Section 1.1.4 the microstructure of SAC-305 can be modeled using spinodal decomposition. This section details the method to simulate the spinodal decomposition phenomenon and creating a usable CAD-file to perform the subseqent elastic simulations on.

The Cahn-hilliard's model for spinodal decomposition is defined by the total free energy, Equation (3.1) [21]

$$F = \int_V \left[ f(c) + \frac{1}{2}\kappa(\nabla c)^2 \right] dv \tag{3.1}$$

$f(c)$ which governs the diffusion/redistribution of c in Equation (3.1), is the chemical or bulk energy for spinodal decomposition, which is modeled using a double-welled potential.

$$f(c) = Ac^2(1-c)^2 \tag{3.2}$$

Equation (3.2) is a simple model where the concentration (c) equal to 0 or 1 corresponds to one of the equilibrium phases. A in Equation (3.2) controls the magnitude of the barrier between the two equilibrium phases.

Cahn-Hilliard's equation is discretized in two dimensions and numerically solved using a finite difference scheme. Periodic boundary conditions are used, meaning the concentration (c) at opposite ends of a boundary are the same. Let $N_x$ and $N_y$ be the grid points in the x and y directions.

$$\begin{aligned} c_{0,j} &= c_{N_x,j}, \quad c_{N_x+1,j} = c_{1,j} \\ c_{i,0} &= c_{i,N_y}, \quad c_{i,N_y+1} = c_{i,1} \end{aligned} \tag{3.3}$$

The initial concentration determines the volume fraction of the two phases, an initial concentration of 0.5 means an equal amount of both phases is present at every node. A very small random value is added to the initial concentration of every node, this ensures that the concentration is not perfectly uniform.

Taking the functional derivative of Equation (3.1) and (3.2), and using an explicit Euler time integration scheme

$$\frac{c_{ij}^{n+1} - c_{ij}^{n}}{\Delta t} = \nabla^2 M \left( \frac{\delta F_{ij}}{\delta c} \right)^n \tag{3.4}$$

In Equation (3.4) M is the mobility coefficient, $\Delta t$ is the time step, n is the current time step and $\nabla$ is the Laplace operator.

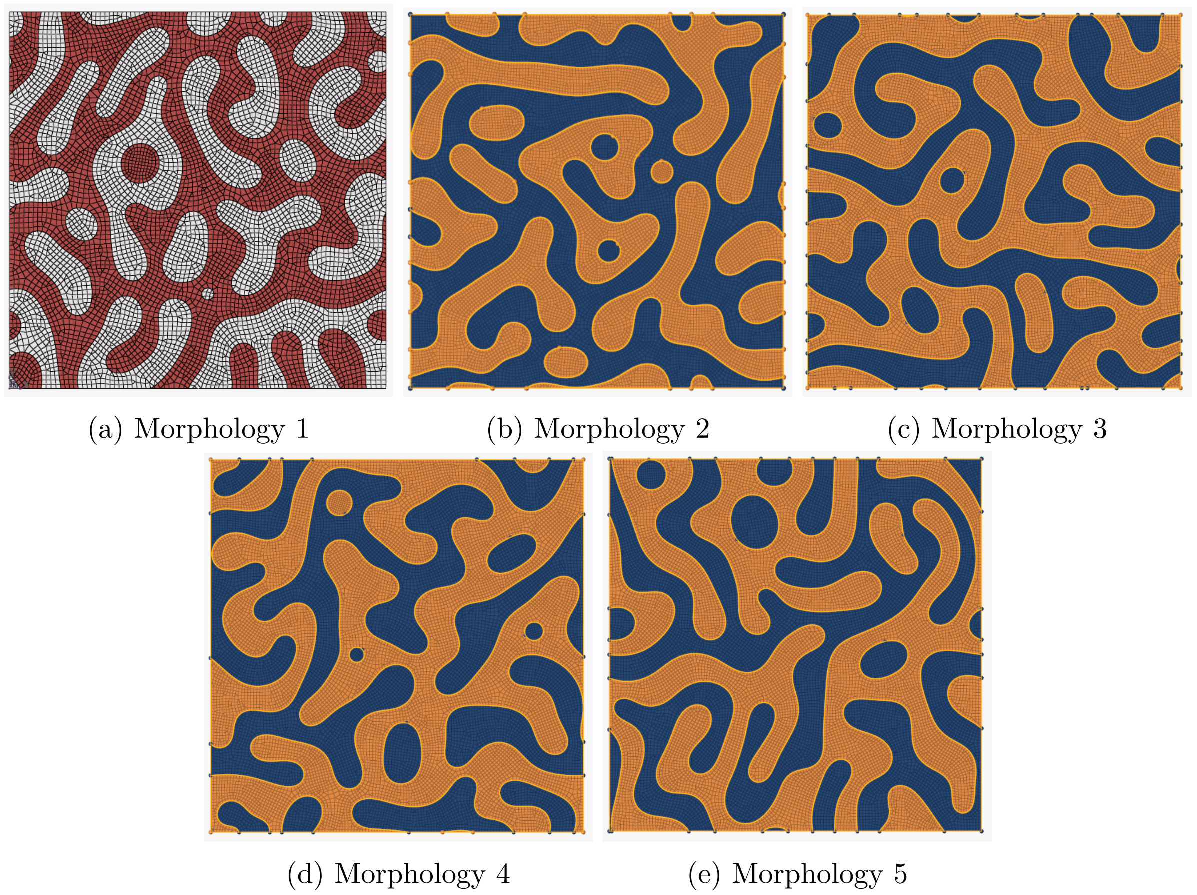
$$\left( \frac{\delta F_{ij}}{\delta c} \right)^n = \mu(c_{ij}^n) - \kappa \nabla^2 c_{ij}^n \tag{3.5}$$

Equation (3.5) is the functional derivative of the free energy, with the functional derivative of the chemical energy (3.2) given by

$$\mu(c_{ij}^n) = A\left(2c_{ij}^n(1-c_{ij}^n)^2 + 2(c_{ij}^n)^2(1-c_{ij}^n)\right) \tag{3.6}$$

The implementation used can be found in Appendix C.1. Constants A, $\kappa$ influence the shape of the solution. The constants were iteratively changed to match the observed resulting microstructures from experiments in [20]. The value for A was 1.5, $\kappa$ was 2, and M was 1.

In Section 1.1.4 the project aims to evaluate the DMN method, a machine learning method, on possible SAC-305 micro structures. Due to the variety and uncertainty associated with the phenomenons responsible for the formation of the micro structure of SAC-305 and other similar alloys, the example micro structures are chosen such that they differ a significant amount. The initial concentration could vary between morphologies to ensure this, but we instead used a 0.5 concentration and a smaller domain. The smaller domain in combination with the 0.5 concentration ensured that the results differed and did not require a too fine discretization during subsequent FEM analysis.


(a) Morphology 1 (b) Morphology 2 (c) Morphology 3

(d) Morphology 4 (e) Morphology 5

Figure 3.3: Spinodal decomposition morphologies

Figures 3.3 are the five morphologies generated from the phase-field simulation. All five morphologies are generated using the same conditions, parameters, and size.

## 3.2 Simulations

The input phases are determined using a Latin hypercube to ensure the least amount of overlap between data points. The compliance matrix of each phase must be positive definite and symmetrical. Equation (3.7) is Sylvester's criterion [25] applied to an orthotropic compliance matrix, where D is the two-dimensional compliance matrix.

$$\begin{aligned} D_{11} &> 0 \\ D_{22} &> 0 \\ D_{33} &> 0 \\ D_{11} * D_{22} - D_{12}^2 &> 0 \end{aligned} \tag{3.7}$$

D is a 3 by 3 matrix, the indices for this purpose start at 1.

The input phase material parameters are then picked with the script located in Appendix C.2.1.

From the previous section a microstructure representation has been generated which can be inserted into a pre-processing software to prepare for the simulation, Figure 3.2 shows all the steps necessary. The phase filed simulation generates contour paths, x and y pairs for a line path object. Next the Ezdxf module is used to create a dxf file, a sort of CAD file. The path objects are used to create polylines, polyhedral line objects in the dxf format. A CAD representation of the morphology has been made in the dxf format, but a step file format is necessary in Hyperworks. Autocad 360 is used to open the dxf file and convert it to a step file. Next the step file can be opened in Hyperworks, constant displacements are set at the boundaries since linearly changing displacement conditions are not supported. Linear displacement conditions are set using a python script, see Appendix C.2.3 specifically the functions called linear_displacement and linear_shear_displacement. Reference material and properties are also set for the two phases so that the bdf files can be easily changed using a python script. The values does not matter since the values from the Latin hypercube will be used. It is important that the material type MAT8 is used and that the PSHELL property type is used.

A reference bdf file has now been created with three loading conditions, linear x displacement, linear y displacement, and linear shear displacement. The next step is to insert the material data the Latin hypercube generates and run the simulations for the loading cases. For the first morphology 1000 input phase pairs were generated, for each pair the input material data is inserted by creating a new bdf file using the reference as a basis. For a material pair three new bdf files are generated for the three loading cases. For each loading case, a solution is generated in terms of a displacement field. The finite element discretization also yields stress and strain fields using the constant given orthotropic material data for each element.

In Section 2.2 the concept that a homogenized representation of the elastic parameters of a heterogeneous material can be described using the volume averaged stresses and strains. From the finite element simulations the volume averaged strain is given from the linear displacement conditions at the boundaries. The volume averaged stress is calculated by summing the elemental stresses multiplied by each elements individual area, divided by the total area of the RVE. Combining the solutions from the three load cases the result is 9 equations with 9 unknowns, the 9 components of the compliance matrix, (neglecting matrix symmetry).

## 3.3 Validating dataset values

The first test for the dataset generation procedure was to make certain that the homogenization of two materials with the same elastic parameters resulted in the same elastic parameters, this was the case. The homogenized material parameters calculated were also checked against the classical homogenization methods in Section 2.2.2. The procedure creates 9 parameters in the compliance matrix, however, due to the symmetry of the compliance matrix only 6 parameters should be independent. A symmetry check is therefore performed to make sure the parameters that should be the same only differ by an inconsequential amount.

Figures 3.4 and 3.5 shows the homogenized $\bar{E}_1$ and $\bar{E}_2$ values as a function of the input Voigt and Reuss analytically homogenized values. $\bar{E}$ is the homogenized stiffness, and $\bar{E}_1$ and $\bar{E}_2$ are the first and second diagonal terms. Figures 3.4 and 3.5 are only for the second and third morphologies, the remaining morphologies are included in Appendix B.

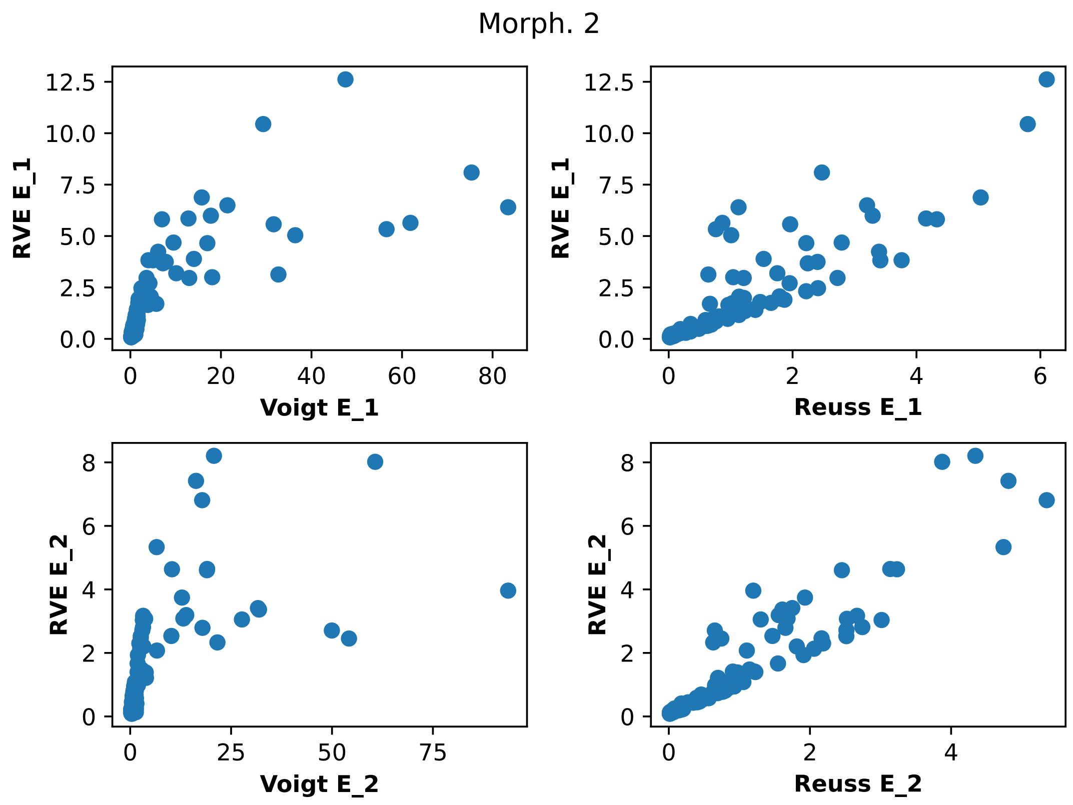


Figure 3.4: Scatter plot of first and second diagonal terms of $\bar{E}$ as a function of Voigt and Reuss bounds for the morphology 2 dataset

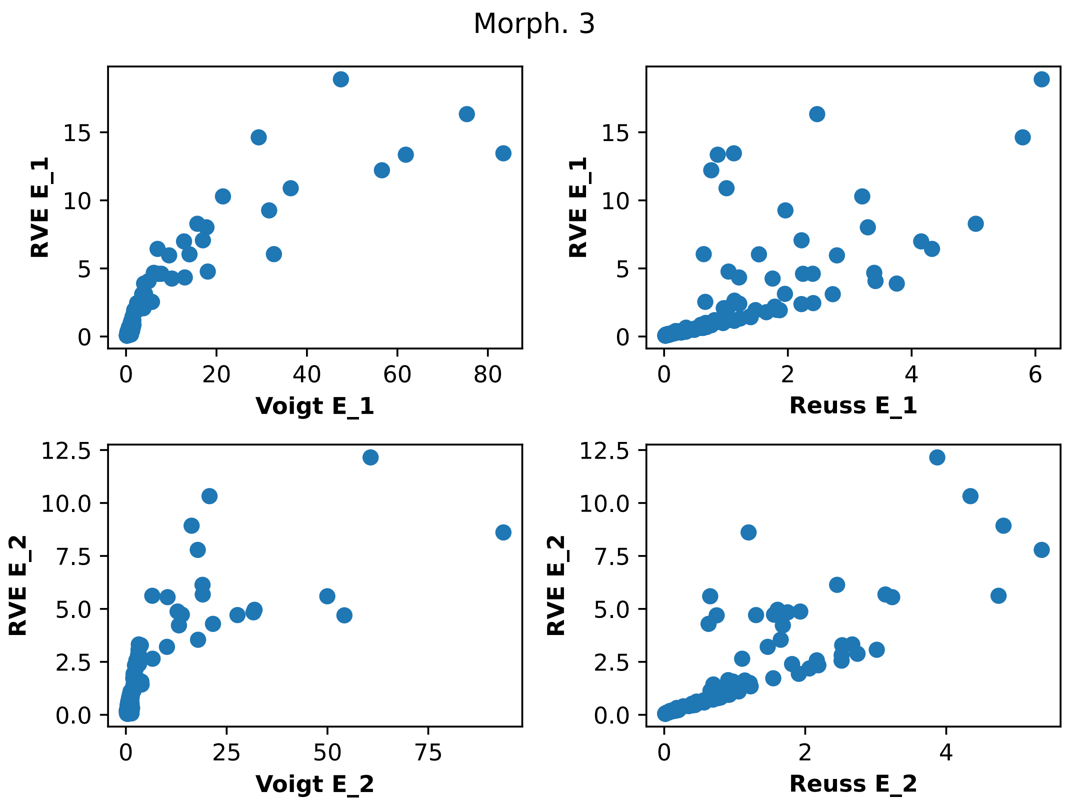


Figure 3.5: Scatter plot of first and second diagonal terms of $\bar{E}$ as a function of Voigt and Reuss bounds for the morphology 3 dataset

The scatter plots shows the correlation between FEM homogenized elasticity module and the Reuss and Voigt upper and lower bounds. The second morphology shows a stronger correlation to the Reuss bounds than the Voigt values. The third morphology shows on the other hand a clearer correlation to the Voigt values, for both $\bar{E}_1$ and $\bar{E}_2$.

# 4

# Deep material network

The DMN method is a type of neural network designed for the specific task of homogenizing heterogeneous materials. The DMN method is based on conventional machine learning techniques described in Section 2.1.

## 4.1 Forward-pass

The DMN's functionality is to produce a compliance matrix representative of the whole microstructure domain given two compliance matrices, one for each phase. The DMN is constructed from (N+1) layers, the first layer, the input layer, contains $2(N+1)$ nodes. The next layer is connected with the first layer. Each nodes gets an input from two nodes of the previous layer, and the next layer gets 2 inputs from the previous layer and so on. The last layer is a single node, the output layer. See Figure 4.1

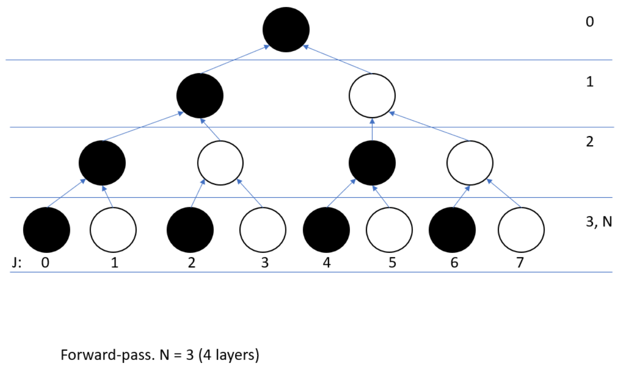


Figure 4.1: 4-layer deep material network

The layer structure is similar to that of a binary tree, each layer has half the amount of nodes as the previous one.

Consider a node from the second layer which takes its inputs from the input layer. Each node in the second layer has two child nodes, one fore each phase. Half of the input nodes contains the compliance of the first phase and half contain the compliance of the second phase.

The parent node and the two child nodes from the previous layer makes a branch. This branch is the fundamental building block of the network. For the second layer the child nodes are in the input layer, for subsequent layers, the input from the child nodes is the combined compliance from previous child nodes. The combination procedure contains a homogenization function, a rotation function, and a bias function. The homogenization function uses a volume fraction derived from a weight, the rotation function uses a rotation angle, and the bias function a bias. All values are initially guessed and later improved upon during the training sequence. Each node has an individual bias, weight, and rotation angle.

The DMN-homogenization utilize an analytical method derived from the DMN building bloc, Figure 4.2.

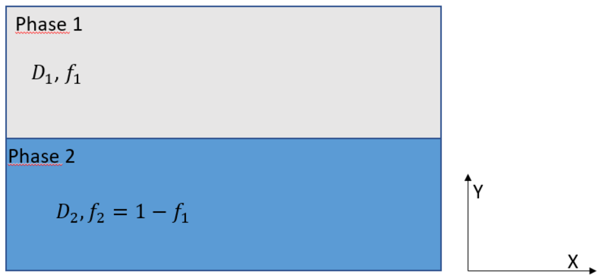


Figure 4.2: DMN building block

The homogenization function is derived from 2D plane strain conditions. The stress-strain relation for the entire morphology can be expressed as

$$\bar{\epsilon} = \bar{D}\bar{\sigma} \tag{4.1}$$

for materials 1 and 2 the same expression holds

$$\begin{aligned} \epsilon^1 &= D^1\sigma^1 \\ \epsilon^2 &= D^2\sigma^2 \end{aligned} \tag{4.2}$$

For two phases in the DMN-block morphology, the stress and strain in the material can be expressed as follows

$$\sigma_y^1 = \sigma_y^2, \quad \sigma_{xy}^1 = \sigma_{xy}^2, \quad \epsilon_x^1 = \epsilon_x^2 \tag{4.3}$$

The equilibrium and kinematic assumptions are the same used for deriving the Reuss and Voigt bounds, Section 2.2.2.

The morphology compliance $\bar{D}$ is derived from two operations, a homogenization and a rotation. $\bar{D}^r$ is the compliance after homogenization but before rotation can be derived from the known kinematic and equilibrium conditions, Equations (4.1 - 4.3).

$$\bar{D}^r = f(D^1, D^2, f_1) \tag{4.4}$$

$$\begin{aligned}
\bar{D}^r_{11} &= \frac{1}{\Gamma}(D^1_{11}D^2_{11}) \\
\bar{D}^r_{12} &= \frac{1}{\Gamma}(f_1 D^1_{12}D^2_{11} + f_2 D^2_{12}D^1_{11}) \\
\bar{D}^r_{13} &= \frac{1}{\Gamma}(f_1 D^1_{13}D^2_{11} + f_2 D^2_{13}D^1_{11}) \\
\bar{D}^r_{22} &= f_1 D^1_{22} + f_2 D^2_{22} - \frac{1}{\Gamma} f_1 f_2 (D^1_{12} - D^2_{12})^2 \\
\bar{D}^r_{23} &= f_1 D^1_{23} + f_2 D^2_{23} - \frac{1}{\Gamma} f_1 f_2 (D^1_{13} - D^2_{13})(D^1_{12} - D^2_{12}) \\
\bar{D}^r_{33} &= f_1 D^1_{33} + f_2 D^2_{33} - \frac{1}{\Gamma} f_1 f_2 (D^1_{13} - D^2_{13})^2
\end{aligned} \tag{4.5}$$

Where

$$\Gamma = f_1 D^2_{11} + f_2 D^1_{11}, \quad f_2 = 1 - f_1 \tag{4.6}$$

The rotation function is simply the matrix multiplication of a rotation matrix. And the rotation matrix is

$$R(\theta) = \begin{vmatrix} cos^2(\theta) & sin^2(\theta) & \sqrt{2}sin(\theta)cos(\theta) \\ sin^2(\theta) & cos^2(\theta) & -\sqrt{2}sin(\theta)cos(\theta) \\ -\sqrt{2}sin(\theta)cos(\theta) & \sqrt{2}sin(\theta)cos(\theta) & cos^2(\theta) - sin^2(\theta) \end{vmatrix} \tag{4.7}$$

The compliance before a bias is added is then

$$\bar{D} = g(\bar{D}^r, \theta) = R(-\theta)\bar{D}^r R(\theta) \tag{4.8}$$

After a bias has been added the compliance is

$$D_{node} = \bar{D} * (1 + bias) \tag{4.9}$$

## 4.2 Weights

The homogenization function f, Equation (4.5), uses the parameter $f_1$. $f_1$ is the volume fraction of the first material of the two materials which are homogenized in the building block. $f_1$ is the parameter in each building block which should be optimized. However, a volume fraction is only indicative of the full homogenization when only 2 materials are homogenized, when two nodes combine too a third parent node. The machine learning system should be generalized to accommodate for the homogenization of any amount of phases. For this purpose we borrow the concept of weights from traditional machine learning [22].

Weights are given for each child node instead of each branch like the volume fraction $f_1$. The volume fraction $f_1$ can be described using the weights as

$$f_1 = \frac{w_1}{w_1 + w_2} \tag{4.10}$$

### 4.2.1 Traditional DMN structure

The DMN structure used by Liu Z. [10] only contains individual weights in the nodes in the input layer, weights in subsequent layers are functions of the input layer weights. The weights of a node not in the input layer is the sum of its child nodes in the previous layer and the weights of those child nodes are the sum of their child nodes if they are not themselves input nodes.

$$w_i^j = w_{i+1}^{j*2} + w_{i+1}^{j*2+1} \tag{4.11}$$

The traditional DMN structure applies the ReLU activation function on the parameters in the input layer before the parameter is used to determine the volume fraction in the homogenization function. The result from ReLU is the weight, before ReLU the parameter is called z. In the traditional DMN structure an addition to the cost function is implemented to constrain the z parameters and improve training. The cost function is introduced in Section 4.5.

The cost function addition is formulated as

$$L(z) = \left( \sum_j Re(z_N^j) - \xi \right)^2 \tag{4.12}$$

*Re* is the ReLU activation function, $\xi$ is a hyper parameter and z are the activations. Index N denotes the input layer and j denote the node in the input layer.

When the sum of the activations equal the hyper parameter the cost is zero. The hyper parameter is therefore most suitable set to the average of the initial activations multiplied by the amount of activation parameters, which due to network structure equals $0.5 * 2^{(N-1)}$.

The cost function additive $L(z)$ is multiplied with a parameter $\lambda$ to control the impact of the function. If $\lambda$ is to large the constraint on the z activations will be over constrained and training will be slow.

### 4.2.2 Conventional ML structure

The DMN method is composed of nodes and not neurons. This distinction might seem irrelevant but is actually very important. In a neural network, neurons are the fundamental building block and contain individual weights and biases. The structure of a neural network is not restricted to such that the same amount of nodes or neurons combine at each layer. A neuron in the conventional sense, should only take a float value, a real number. The DMN method process a 3 by 3 compliance matrix or 6 individual components of the compliance matrix. To retain the connection with the physical representation of homogenization the individual components can not be split up into 6 neurons and treated individually. A single weight is necessary for a node containing six float values.

A DMN method which more closely aligns with conventional neural networks is designed. Our method uses individual weights, rotation angles, and biases for each node. Our new structure retains a closer connection with preexisting knowledge from neural networks [22] and should therefore be easier to understand and reuse methods for.

The new structure is simply to remove the formulation to determine the weights for nodes in non-input layers using Equation 4.11, and allow the weights in all layers to be individually determined. The cost function addition Equation 4.12 is also removed. The addition was a necessity due to the use of inherited weights now removed.

## 4.3 Bias

In addition to a weight governing a function combining the signals from a node's connected child nodes, a bias is also typically used in a neural network. In a conventional neural network composed of neurons sending real numbers, a bias is an added term on the activation in the form of another real number. The bias is updated separately from the node's weight but during the same back propagation step. For a network comprised of nodes sending a compliance matrix a conventional bias can not work. Previous works on the DMN method forgo the inclusion of a bias altogether, never even mentioning its possible use. In conventional neural networks a bias is a necessity to ensure the network can minimize the cost function. In our DMN structure which utilize individual weights and rotation angles for each node, a bias has also been added. The bias should work with a physical reference in mind, comparative to the addition or subtraction of a compliance matrix. Our method stiffens the material or weakens the material with the bias. If the bias of a node is zero no change occurs, when the bias is positive a percentage increase is added to the compliance matrix weakening the material. For example, a bias of 0.1 increase the compliance of a node by 10 %, a bias of -0.1 decreases the compliance by 10 % instead.

$$D_{node} = \bar{D} * (1 + bias) \tag{4.13}$$

$\bar{D}$ is the rotated and homogenized compliance of a node, $D_{node}$ is the compliance sent to the next layer.

Combining all the parameters and the structure of the network described earlier, the forward-pass can be shown for a part of a network, Figure 4.3.

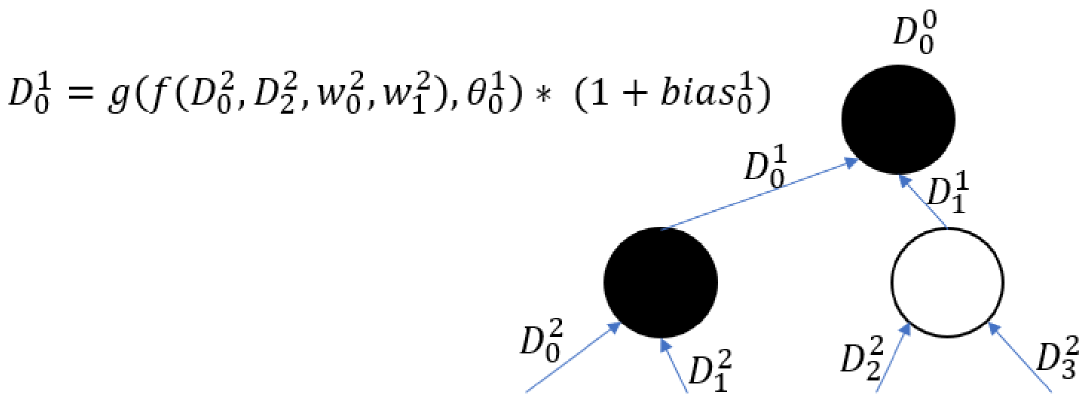

$D_0^1 = g(f(D_0^2, D_2^2, w_0^2, w_1^2), \theta_0^1) * (1 + bias_0^1)$
$D_0^0$
$D_0^1$
$D_1^1$
$D_0^2$
$D_1^2$
$D_2^2$
$D_3^2$

Figure 4.3: Part of network during forward-pass

## 4.4 Cost function

The cost function determines how the error of the network should be quantified. A typical cost function for machine learning is mean squared error (MSE).

$$C = \frac{1}{N}\sum_{s}\frac{||D_s^{dns} - \bar{D}_s||^2}{||D_s^{dns}||^2} \tag{4.14}$$

The sample amount (N) is a parameter that can be altered. Using the entire data set to produce a single error is a good stable practice but it's slow. A common approach in machine learning is to divide the dataset into mini-batches, which allows for multiple training steps during a single epoch. This accelerates training time but can lead to divergence and even slower training since the network is not evaluated on the whole network at any training step.

During each epoch the data set is randomly shuffled and mini-batches are created by dividing the data set into equally large mini-batches. The mini-batch size during most training was in the region of 1/8 - 1/4 of the total size of the dataset. Multiple datasets with different sizes have been used during this work, the important part is to make sure the mini-batch size is divisible with the total dataset.

## 4.5 Back propagation and learn step

In Section 2.1 the concept of back propagation and the stochastic gradient descent method is introduced. The purpose of the method is to determine the gradient of the cost function with respect to the parameters in the network that should be updated. The neural network can be likened to a single massive function that takes one set of inputs and produce a set of outputs.

For the purpose of a forward pass, the input is the parameters of the two material phases and the output is the parameters of the resulting compliance matrix. For a learn step, the input is instead the parameters which controls the function and the output is the cost or error. If we consider the neural network as a single function we can use the chain rule to determine the gradient of the resulting cost function with respect to any individual parameter in the network. Due to the structure of a neural network, where each layer is connected only to the layer above and below, this procedure of deriving the gradient for an individual parameter is performed by iteratively moving backwards from the output layer where the cost is generated.

The gradient is determined iteratively for each step in the process with respect to the temporary information the step generates. The last step in the forward-pass generates the output, the predicted compliance matrix for the RVE. The first step in the back-propagation is to derive the output with respect to the cost function.

For the purposes of the forward-pass there is no distinction between a compliance matrix and a vector containing the 6 individual parameters. For this step when the gradient is determined as a result of the intermediary information its important to make this distinction of 3 by 3 compliance matrix and 6 by 1 vector of individual components. For a node, the compliance matrix after homogenization is $\bar{D}^r$, when put in the the perspective of the whole network the 6 individual components are $a_i^j$. The compliance matrix after rotation is $\bar{D}$ and its individual components are $d_i^j$. After the bias is added the compliance matrix is $D_{node}$ and its individual components are $h_i^j$. Sub-script is as a rule the index of the layer in the network, and the super-script is either the node at that layer or the components for a node at that layer. For a faster back-propagation step, the components should be structured in 6N by 1 vectors at each layer, with N as the amount of nodes at that layer, this would make the subsequent steps shown much faster since matrix multiplications for each layer could be performed instead of for each individual node.

The cost function is based on the mean square of errors, a common ML cost function. The Frobenius norm of a matrix is the square root of the sum of squares of its components.

$$||D||^2 = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{m} b_{ij}^2}^{\,2} \rightarrow$$
$$||D^{dns} - \bar{D}||^2 = \sum_{i=1}^{m}\sum_{j=1}^{m}(d_{ij}^{dns} - \bar{d}_{ij})^2 \tag{4.15}$$

The gradients with respect to the components $\bar{d}_0^j$, components j at layer 0, is then

$$\frac{\partial C}{\partial \bar{d}_0^j} = 2(\bar{d}_0^j - d^{dns}) \tag{4.16}$$

Equation (4.16) shows the gradient at the output node, using the chain rule from calculus as basis the gradient is propagated backwards through the entire network. The DMN back propagation is the same for our network structure and previous DMN structures [9].

$$\begin{aligned}\delta_i^n &= \frac{\partial C}{\partial d_i^n} \\ \alpha_i^n &= \frac{\partial C}{\partial a_i^n} \\ \beta_i^n &= \frac{\partial C}{\partial h_i^n}\end{aligned} \tag{4.17}$$

$\delta$ and $\alpha$ are local errors for a node, these are later used to update the weights, rotation angles, and biases. $\delta$ and $\alpha$ for a node are calculated backwards from local errors of the previous errors and the local gradients.

$$\alpha_{i-1}^k = \delta_{i-1}^j \frac{d_{i-1}^j}{\partial a_{i-1}^k} \tag{4.18}$$

$$\delta_{i-1}^n = \beta_{i-1}^k \frac{\partial h_{i-1}^k}{\partial d_{i-1}^n} \tag{4.19}$$

$$\beta_i^n = \alpha_{i-1}^n \frac{\partial a_{i-1}^k}{\partial h_i^n} \tag{4.20}$$

$$\frac{\partial d_{i-1}^j}{\partial a_{i-1}^k} = R_{ik}(-\theta) R_{lj}(\theta) \tag{4.21}$$

$\frac{\partial a_{i-1}^k}{\partial h_i^n}$ is included in the Appendix A. $\frac{\partial h_{i-1}^k}{\partial d_{i-1}^n}$ is the gradient of the compliance matrix after the bias has been added with respect to the compliance before the bias, Equation (4.13).

$$\frac{\partial a_{i-1}^k}{\partial h_i^n} = \begin{vmatrix} 1+b_i^j & 1+b_i^j & 1+b_i^j \\ 1+b_i^j & 1+b_i^j & 1+b_i^j \\ 1+b_i^j & 1+b_i^j & 1+b_i^j \end{vmatrix} \tag{4.22}$$

The weights in a node dictate the volume fraction used in the homogenization in its connected parent node. The update rule for a weight depends on the local error of its parent node and the gradient of the weight with respect to the volume fraction.

$$\frac{\partial C}{\partial w_i^j} = \alpha_{i-1}^k \frac{\partial a_{i-1}^k}{\partial f_1} \frac{f_1}{w_i^j} \tag{4.23}$$

In Equation (4.23) the first term is the local error of the parent node, and second is the gradient of the local compliance with respect to the volume fraction of the weight of the child node which is updated. The third term is the gradient of the volume fraction with respect to the weight. $\frac{\partial a_{i-1}^k}{\partial f_1}$ is included in Appendix A

The formulation of the weight parameter is shown in Section 4.2. The gradient of a volume fraction and a weight is

$$\frac{\partial f_1}{\partial w_1} = \frac{w_2}{(w_1+w_2)^2} \quad \frac{\partial f_1}{\partial w_2} = -\frac{w_1}{(w_1+w_2)^2} \tag{4.24}$$

The local compliance matrix and gradients are for the purpose of back propagation a 6 by 1 vector of the individual components, this is to make sure the non diagonal components are not summed twice and over represented.

In addition to the weights rotation angles and biases are used in the network and needs to updated as well. The rotation angle gradient is first calculated

$$\frac{\partial C}{\partial \theta_i^k} = \delta_i^j \frac{\partial d_i^j}{\partial \theta_i^k} \tag{4.25}$$

Beta is the local error before rotation but after the bias has been applied as shown earlier. The local gradient of the compliance matrix with respect to the rotation angle is

$$\frac{\partial \bar{D}}{\partial \theta} = -R'(-\theta)\bar{D}^r R(\theta) + R(-\theta)\bar{D}^r R'(\theta) \tag{4.26}$$

R is the rotation matrix from Equation (4.7) and R' is its derivative with respect to $\theta$.

$$R'(\theta) = \begin{vmatrix} -sin(2\theta) & sin(2\theta) & \sqrt{2}cos(2\theta) \\ sin(2\theta) & -sin(2\theta) & -\sqrt{2}cos(2\theta) \\ -\sqrt{2}cos(2\theta) & \sqrt{2}cos(2\theta) & -2sin(2\theta) \end{vmatrix} \tag{4.27}$$

The gradient of a bias with respect to the output cost is calculated the same as for the rotation function.

$$\frac{\partial C}{\partial b_i^k} = \beta_i^j \frac{\partial h_j^k}{\partial b_i^k} \tag{4.28}$$

$\frac{\partial h_j^k}{\partial b_i^k}$ or in the format used when showing the bias function, Equation (4.13) $\frac{\partial D_{node}}{\partial b_i^k}$ is the gradient of the components after a bias has been added with respect to that bias. This gradient is clearly the components before the bias is added, $d_i^j$ or $\bar{D}$.

### 4.5.1 Adaptive learning rate

An adaptive learning rate method determines a suitable learning rate for each individual parameter, during each training step to ensure stable training. A common type of adaptive learning rate methods modify a baseline learning rate $\eta$ with the current gradient of the cost function with respect to the parameter to change. If the gradient is very large the learning rate should be reduced, if the gradient is very small, the learning rate can be increased to facilitate faster training.

The adaptive learning rate tested is RMSprop [26], a common learning rate method for stochastic mini-batch machine learning.

$$\begin{aligned} c^n &= c^{n-1} * \gamma + (1-\gamma) * \left(\frac{\partial C}{\partial w_j^i}\right)^2 \\ \eta &= \frac{\eta_0}{\sqrt{c} + \epsilon} \end{aligned} \tag{4.29}$$

The c parameter is stored between each learning step, when a new gradient is calculated the c parameter is updated using weighed averaging. n is here a time step. If the new gradient is larger than the previous c the new c will increase. If the new gradient is smaller, the c value will decrease. $\eta$ is the learning step size used to update its corresponding weight $w_j^i$, $\eta_0$ is some reference learning rate which is modified for its weight.

A different adaptive learning technique is called momentum [22]. The new gradient for time step n contains the previous gradient multiplied by a decay rate gamma. The decay rate can be any value from [0-1). However a decay rate close to one will result in a very large momentum and poor learning, overshooting occurs easily and nodes will be deactivated to early.

$$\Delta w_j^i(n) = \eta * \frac{\partial C}{\partial w_j^i} + \gamma * \Delta w_j^i(n-1) \tag{4.30}$$

If the error at a node is constant the gradient increases continuously. If $\gamma$ is below one an upper limit exists that the gradient approaches. In Equation (4.30) $\eta$ is the learning rate. $\Delta w_j^i$ is the change to parameter w at layer j in node i. $\delta w_j^i$ (n-1) indicates the change during the previous update.

# 5

# Results

During training the weights, rotation angles, and biases are presented visually. Each subfigure includes a single layer with the input layer starting at the top. The cost and error are displayed at the bottom of the figure. The cost function, Equation (4.14) is the function the network attempts to minimize. The average error is

$$E = \frac{1}{N} \sum_s \frac{||D_s^{dns} - \bar{D}_s||}{||D_s^{dns}||} \tag{5.1}$$

In Equation (5.1) index $s$ is a sample in a dataset. $\bar{D}$ is the generated RVE compliance from the input values for the network and $D^{dns}$ is the correct output value contained in the dataset. $N$ is the dataset size, meaning E corresponds to the average error when the network evaluates a dataset. $||D||$ is the matrix norm or specifically the Frobenius norm of matrix $D$ mentioned in Section 4.15.

The error function shows the normalized average error of the network evaluated on a validation dataset. A validation dataset is a dataset derived in the same way as the training dataset but kept separate from the training dataset. The network process the data from the validation dataset but should not update the weights, rotation angles, or biases based on that error. This dataset separation ensures that the error is reflective of the networks actual use rather than when its applied to the specific information of the training dataset.

| DMN \ Dataset | Morph. 1 | Morph. 2 | Morph. 3 | Morph. 4 | Morph. 5 | Morphs. 1-5 | Morphs. 3-5 | Morphs. 2&5 |
|---|---|---|---|---|---|---|---|---|
| Morph. 1 | 0.033 | 0.127 | 0.270 | 0.244 | 0.199 | 0.170 | 0.149 | - |
| Morph. 2 | 0.139 | 0.025 | 0.230 | 0.221 | 0.202 | 0.164 | 0.145 | - |
| Morph. 3 | 0.288 | 0.218 | 0.031 | 0.073 | 0.141 | 0.154 | 0.188 | - |
| Morph. 4 | 0.269 | 0.220 | 0.076 | 0.030 | 0.101 | 0.140 | 0.201 | - |
| Morph. 5 | 0.229 | 0.206 | 0.150 | 0.110 | 0.029 | 0.143 | 0.214 | - |
| Morphs. 1-5 | 0.158 | 0.124 | 0.136 | 0.117 | 0.112 | 0.115 | 0.138 | 0.105 |
| Morphs. 3-5 | 0.130 | 0.104 | 0.152 | 0.142 | 0.144 | 0.123 | 0.121 | 0.113 |
| Morphs. 2&5 | - | - | - | - | - | - | - | 0.091 |
| Voigt | 0.443 | 0.461 | 0.457 | 0.463 | 0.469 | 0.452 | 0.461 | 0.457 |
| Reuss | 1.299 | 1.230 | 1.282 | 1.226 | 1.170 | 1.253 | 1.394 | 1.259 |

Table 5.1: Average error for DMNs evaluated on different data sets

In Table 5.1 the DMNs are trained on the dataset with the same name, Morph. 1 is trained on dataset Morph. 1, Morphs. 1-5 is trained on dataset Morphs. 1-5, etc. For single morphology datasets 1 trough 5, the first five rows, the error is significantly lower for the dataset the DMN has been trained on. The error for a single morphology DMN is consistently in the 0.025-0.03 range when evaluated on its corresponding dataset. The best evaluation of a DMN on a dataset it was not trained on is DMN 4 evaluated on a dataset for morphology 3 with a 0.076 error.

Morph. 4 DMN error 0.0304 corresponds to a cost of 0.000572. The cost was initially 0.2 - 0.1 before training commenced for all DMNs, this means the cost has been reduced by a factor 200-400 for the single morphology datasets.

Voigt and Reuss analytical results are calculated using the equations in Section 2.2.2. Both analytical methods results in a new orthotropic material with zero non-diagonal terms $D_{1,3}$ and $D_{2,3}$. It should be noted that the Voigt and Reuss assumptions give the upper and lower bounds for the eigenvalues.

## 5.1 DMN trained on single micro structure

A DMN was first trained on a single morphology with a dataset containing input material compliance, the first and second phase, and the resulting homogenized compliance. The first and second phase values are selected using a Latin hypercube method described in Section 3.2. The python script performing this is included in Appendix C.2.1
During training we showcase the change to weights, rotation angles, and biases alongside the current error and cost. Figure 5.1 shows 1000 epochs during training of a DMN on the morph. 1 dataset.

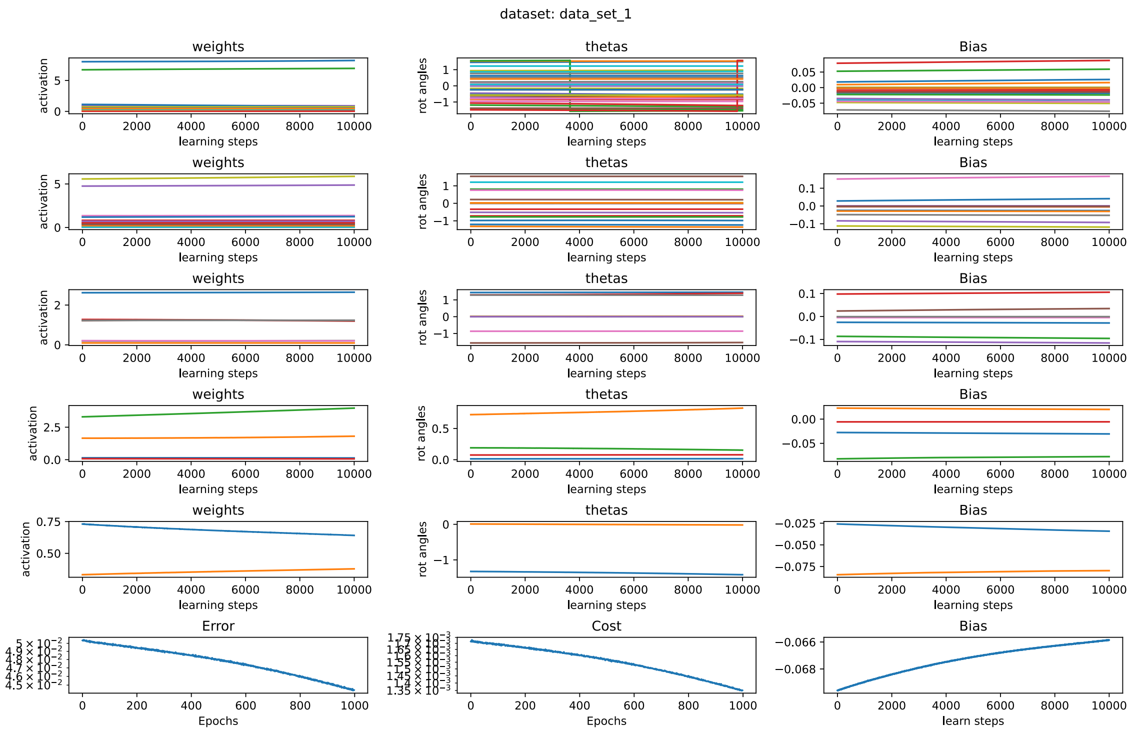


Figure 5.1: Graphic displayed during training. Shows the weights, rotation angles, and biases of all layers as well as the cost and error

Figure 5.2 shows the validation error for the DMN trained on the morphology 3 dataset. The x axis shows training time in 100 epochs and logarithmic scale, and the y axis shows average error, Equation (5.1).

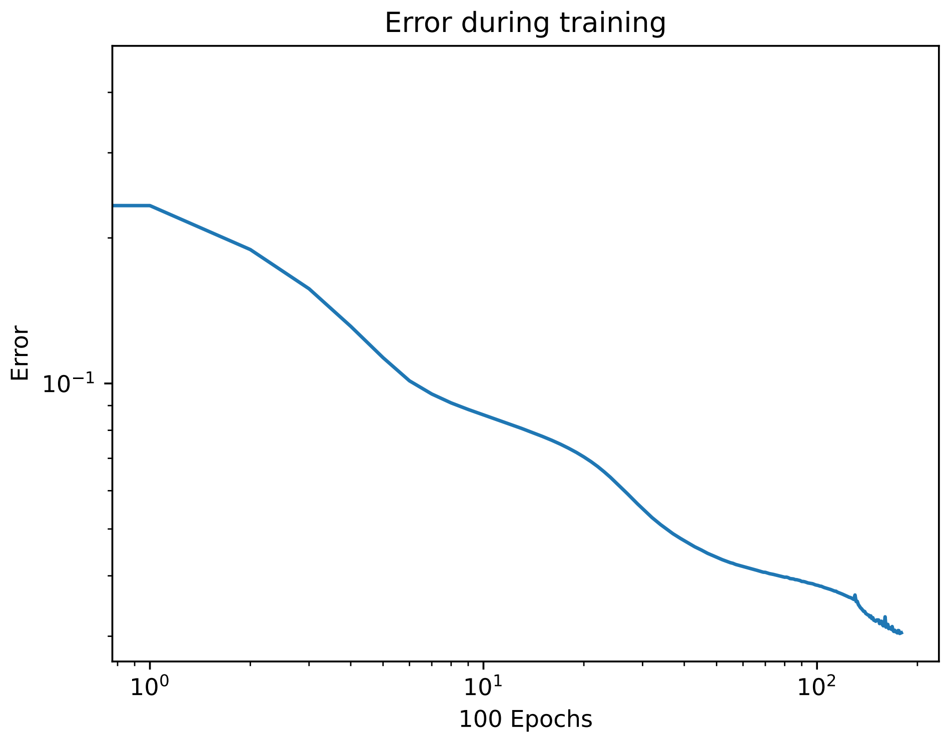


Figure 5.2: Validation error for morphology 3 DMN during training

Three training sequences were performed for this DMN totaling 18000 epochs of training. Each training stint of 6000 epochs took about 12 hours, the full training for this network structure required about 36 hours to complete. A deeper network requires fewer epochs to converge to a similar error but each epoch takes longer to complete, in the end a similar amount of run time is required to complete training for networks in the 6-9 layer range. Networks become much slower to train with more layers and converge to worse results with fewer layers. Final versions for DMNs 1 and 2 used 6-layers, DMN 3 used 7-layers, DMNs 4 and 5 used 8-layers.

## 5.2 Demonstration of temperature variance

A trained DMN can be used to easily generate homogenized material parameters. One way to showcase this use pragmatically is to apply it to a case with some temperature varying properties.

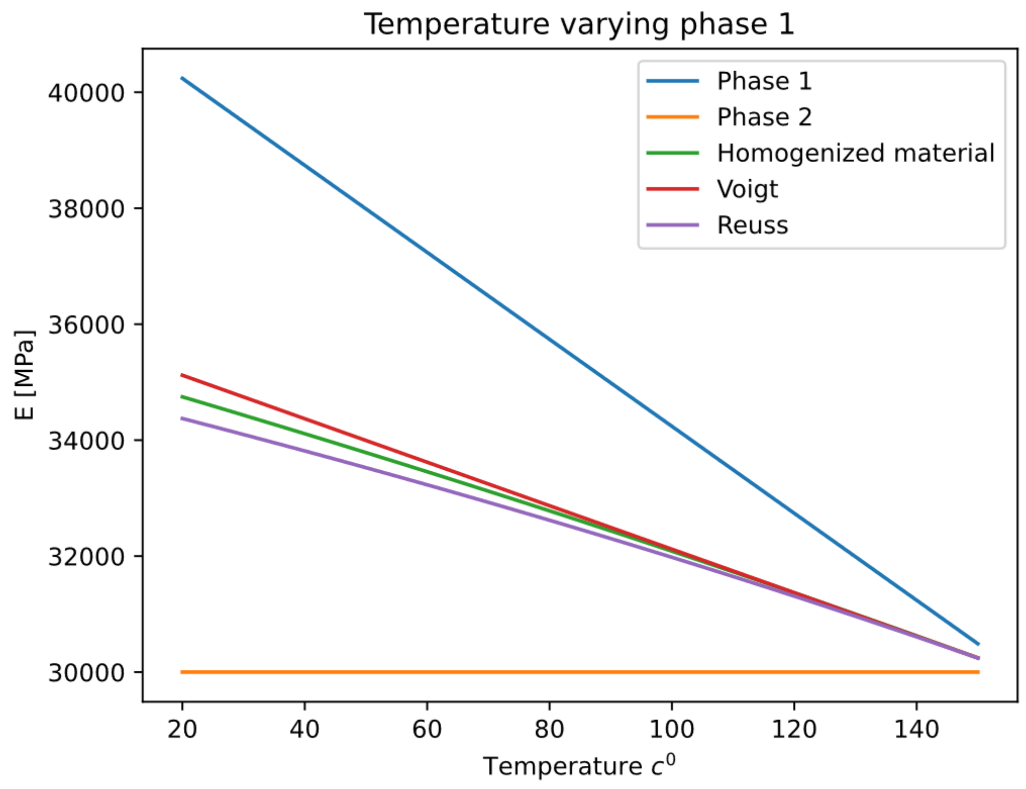


Figure 5.3: Demo of temperature varying first phase for morphology 1

In Figure 5.3 the first and second phase are for this demonstration isotropic. The first and second phase elasticity module is

$$\begin{aligned} Phase_1 : E_1 = E_2 = 6.2225 * 10^4 - 75 * (T + 273.15), \nu = 0.35 \\ Phase_2 : E_1 = E_2 = 3 * 10^4, \nu = 0.35 \end{aligned} \tag{5.2}$$

The first phase elasticity parameters are the same as for SAC-305. In Equation (5.2) T is the temperature in °Celsius. The second phase's elasticity module is chosen as constant and lower than the first phase's elasticity module. The Reuss and Voigt analytical homogenized values are showcased as well. The DMN homogenized values are always in between the Reuss and Voigt values.

# 6

# Discussion

This chapter includes a discussion of the DMN method based on the results in Section 5.

## 6.1 New network structure

A new network structure was introduced in Section 4. Our new method retains a closer connection to neural networks with individual weights and biases. The reason for a new network structure was that the DMN method described in [9] was difficult to interpret.

$$\frac{\partial C_0}{\partial z_N^j} = \left( \alpha_{i-1}^l \frac{\partial a_{i-1}^l}{\partial f_i^m} \frac{\partial f_i^m}{\partial w_N^j} \right) \circ RE'(z_N^j) \tag{6.1}$$

Equation (6.1) is the update rule for the old DMN structure. Two things are unclear in this equation and could be interpreted in different ways. The local error $\alpha$ and $\delta$ propagates through the network to the input layer, i-1 can be interpreted as the layer index of the last layer before the input or as a summation index indicating that every layer contributes to the output gradient.

Section 4.5.1 mentions the use of learning rate methods during training. In total four different approached were used in some capacity during training on the different datasets. The first and simplest approach is a constant $\eta$, this method is the one used for all of the converged results in Table 5.1. The learning rate for the weights and the rotation angle was 0.1 for the first 1000-3000 epochs and would be increased to 0.3 after a full training stint. The bias learning rate was also constant and at a much smaller value of 0.001.

RMSProp was used in early iterations of the network but was problematic to use. This method relies on a lot of hyperparameters and choices. First the parameters $\gamma$ and $\eta_0$ which determines decay rate and the magnitude respectively. Then there is $\epsilon$ which is important if c approaches zero, if $\epsilon$ is too small $\eta$ might explode. Lastly the initial value for c must be considered. If it is zero the initial learning rate will be very large. The complexity of this method made it to much work and a possible source of errors and was thus not used for the final training runs. The third approach was a simple inclusion of momentum. This method was easier to use but the addition of another hyper parameter for no apparent benefit made the method redundant, the method worked best when the momentum term was close to zero anyway.

The new network structure could sometimes be observed diverging at the last layer, closest to the output layer, which mean that the volume fraction approached 1 or 0. This can be observed in Figure B.6 in the appendix. This means that a network with one less layer could produce the same results. This phenomenon is very unwanted since it reduces the amount of useful parameters by half by a single parameter. This phenomenon might be the reason why the traditional structure was used by Liu Z. [9], although this is never mentioned. The traditional structure can not have this problem since the nodes can only be turned off at the input layer. This problem might be reduced using the hypothesized learning rate method using the layer index as a parameter, then the training could be initialized at the earlier layers and postponed at the later layers, ensuring that a deactivation at those layers does not occur early at least.

### 6.1.1 Bias

The inclusion of a bias in the network was meant to make the DMN as closely as possible match a neural network. In Section 4 the differences between a DMN and a NN is discussed briefly. A NN is composed of neurons and a DMN is not, a NN only process floats in any fully connected layer. There is typically never any benefit to process multi dimensional data in its original shape. The first step in a NN is always to reshape the input matrix into an N by 1 vector where each of the N neurons get a single real number. The reason is simple really, why would you try to solve a problem with a few node with multi dimensional data when it is possible to make the problem much easier by splitting the data up unto more neurons. The DMN is one of if not the only example where a multi dimensional format is necessary. If the data would be split up it would no longer represent a material in the intermediary steps of the hidden layers. For the purposes of predicting an output compliance this might not be a big deal, the benefit of simplifying the data structure would mean 100s if not 1000s of more layers could be used without increasing the training time. The problem is that the network would only be able to predict homogenization on linear elasticity. A neural network would not be a reduced-order model in the sense that it would not be able to predict material homogenization for non-linear models like plasticity and hyper elasticity.

With this in mind, the ways to include a bias are limited. The bias should after being added produce a compliance matrix. The bias should be linearly independent of the previous functions used in a node, the homogenization and rotation functions. The compliance matrix the bias produce should be positive definite, fulfilling Sylvester's criterion Equation (3.7).

$$\begin{aligned} D_{11}(1-b) &> 0 \\ D_{22}(1-b) &> 0 \\ D_{33}(1-b) &> 0 \\ D_{11}D_{22}(1-b)^2 - D_{12}^2(1-b)^2 &> 0 \end{aligned} \tag{6.2}$$

The first three conditions are fulfilled if b > -1 and Sylvester's criterion was fulfilled prior to the bias being added. The last condition can be formulated as

$$(D_{11}D_{22} - D_{12}^2)(1-b)^2 > 0 \tag{6.3}$$

it is clear that Equation (6.3) is again fulfilled assuming Sylvester's criterion was fulfilled before.

## 6.2 Multi-morphology datasets

Multiple datasets were made that included more than one morphology. It became clear very early on that a data set including more than one morphology would result in a worse error and cost, independent of learning rates or network structure. The hope was that the difference would not be too significant and the error comparable. As can be observed in Table 5.1 the data sets with multiple morphologies were at best converging at errors three times that of the networks with a single morphology. The DMNs trained on datasets with multiple morphologies were also much more difficult to train and dependent on good hyper parameters and initial values.

The datasets which included multiple morphologies did not include multiple data point with the same input, this should make it easier for the network to train on multiple morphologies at once. The intention was that the network would learn something abstract about the concept of spinodal decomposition when multiple morphologies were used in a single dataset. A network that can capture this phenomenon could be incredibly useful in multiscale simulation on alloys or for other applications.

The datasets on morphologies 2 to 5 used the same input, a maximum of 25% of each dataset could therefore be combined from each, otherwise the same inputs would be used more than once. A dataset containing multiple morphologies should perhaps include more data points to function better. It is also possible that including more morphologies could improve the end result of a DMN trained on the combined dataset. With more morphologies the dataset would better represent the full phenomenon.

## 6.3 Multi-scale simulations

In Section 1.1.4 it is mentioned that the DMN method can be used to model non-linear material models from a DMN trained on linear elastic training data. The incredible usefulness of this phenomenon should be reiterated. When the DMN models linear elasticity for a component, a single use of the DMN generates the material stiffness parameters for the entire component. If the material structure, the morphology, varies in the component multiple DMNs are used but still only once. The DMN is then used in a pre-processing capacity. For non-linear material models, plasticity for example, the history of the loading influences the current elasticity parameters. The loading history will most likely vary in the component meaning a single use of the DMN is not possible. In fact, the DMN needs to be used for each element individually and for each time step. This is still assuming the material morphology is the same in the entire component.

The DMN method can be compared to the direct numerical way of homogenizing plasticity. The DNS way of modeling non-linear homogenization uses FEM for the morphology, similar to the creation of the dataset. For an element to determine the elasticity parameters, Newton iterations are performed on the second layer FEM discretization until convergence. This is whats known as finite element method squared, since it involves entire additional dimension. This is a necessity even when the material micro structure is assumed the same through out the component [6]. With the DMN method the Newton iterations can instead be applied to the DMN until convergence [9].

$$\begin{aligned} \Delta\epsilon_i^k &= D_i^k \Delta\sigma_i^k + \delta\epsilon_i^k \\ \delta\epsilon_i^k &= q(D_{i+1}^{2k-1}, D_{i+1}^{2k}, \delta\epsilon_{i+1}^{2k-1}, \delta\epsilon_{i+1}^{2k}) \end{aligned} \tag{6.4}$$

In Equation (6.4) $\delta\epsilon_i^k$ is the residual strain in node k at layer i. q is the function combining the residual strains from the previous layer, corresponds to the homogenization function f in Equation (4.5). q is based on the equilibrium and kinematic constrains that govern the DMN building block, Figure 4.2. Similarly when the compliance matrix is rotated or a bias is added the residual strain is processed in i similar fashion. This shows the necessity that any intermediary step in the network must correspond to an actual material compliance matrix.

The use of a computational homogenization method, regardless of type, require a different set of material parameters compared to a simpler FEM simulation. The material parameters of the phases involved can sometimes be more difficult to find or be disputed in research. As was mentioned in Section 1.1.4 lead-free solder alloys and specifically SACs have been the focus of research for some time. This includes applications of multi-scale modeling on such alloys. Maleki, et. al [27] performed multi-scale simulation on SAC-405, an alloy with slightly more silver than SAC-305. Maleki, et. al modeled the intermetallic Ag3Sn with isotropic linear elasticity with an elasticity modulus of 90 GPa. The elasticity modulus of Ag3Sn when considered isotropic has been reported in the range 75 to 115 GPa [27]. Ag3Sn can be considered isotropic if a large enough mass is considered so that the anisotropy of the crystalline structure is nullified.

The machine learning method is developed for the task of performing a homogenization for a micro structure faster. The exact time it takes for a DNS homogenization using FEM depends on the mesh refinement, the codes efficiency, computer hardware, and some problem specific assumption such as linear elasticity, isotropicity or anisotropicity. For the creation of the datasets, which describe homogenization for a morphology using Nastran MSC and a mesh with 8000-9000 elements, 2D, linear elasticity, plane strain, orthotropic input phases, the time it took was about 30 seconds. Those 30 seconds include solving three separate loading conditions, relocating result files, reading result files using regular expressions, and lastly performing the calculations to get a compliance matrix. In comparison when the DMN is used to generate a material compliance matrix, the time it takes is in the 50-60 millisecond range. It took 50-60 milliseconds for an 8-layered network for two-phase microstructures. If the representation of the micro structure requires a finer mesh, or three dimensions the DNS approach will skyrocket in time required to homogenize the material phases, whereas the DMN method after training will still only require 10s of milliseconds to fulfill an iteration.

# 7
# Conclusion and future research

This chapter concludes the rapport on deep material networks for lead-free solder alloys and discusses the necessary future work before DMNs and multi-scale simulations can be applied in an industrial setting.

## 7.1 Deep material networks for lead-free solder alloys

The problem of the uncertainty of the microstructure remains a problem. The multi-morphology datasets intended to mitigate this problem by providing a network capable of modeling multiple different microstructures. Liu, Z. the original creator of the DMN method solves this uncertainty problem using a different approach, using multiple DMNs with individually datasets which combines to map the microstructure space. The microstructure-guided deep material network [12] is applied to short fiber reinforced polymer composites, they train some DMNs on the extreme case of aligned fibers with the lowest assumed volume fraction, some with highest assumed volume fraction, some with randomly oriented and lowest volume fraction, and so on. The DMNs each model an extreme case that could exist in the material. The DMNs are then combined to create a single DMN which represents the entire domain.

The use of adaptive learning methods could be further improved upon. One hypothetical learning method was considered for this network. Since the network has the shape of a binary tree, it was considered that learning should occur first for nodes closer to the input layer and slower for nodes closer to the output layer. A method utilizing the layer index could solve the problem mentioned in Section 6.1 where one of the two nodes in the second to last layer are deactivated early, effectively halving the amount of useful parameters.

## 7.2 Future research

The deep material network and multi-scale simulations as a whole require a lot of additional work if it is to be adopted for industrial use. Programmatically slotting a DMN into a FEM framework is not a big problem, previous work by Liu Z. [9] have already combined LS-

DYNA with a DMN. The additional work comes from the material parameters and micro structure representations and resulting datasets before a DMN can be trained. Building a library of DMNs would be a necessary first step before the method could be used with any time efficiency, which will require a lot of work and research using information that is not typically used. Currently when a FEM simulation is performed for SAC-305 a homogeneous and isotropic elasticity modulus is used, with some temperature dependency and if plasticity is considered it's again considered homogeneous. For a multi-scale simulation the elastic parameters must be given for each individual phase instead, something that can be difficult to find, difficult to experimentally determine, or be debated in research as mentioned for Ag3Sn [27].

Multi-principal element alloys (MPEAs) are alloys where a single element does not make up the majority of an alloy, instead the weight percentage is more evenly split among constituents. Research into MPEAs evaluates hundreds of thousands of possible alloys to try and find new useful materials. Current research uses the simplest analytical solution to homogenize the constituents and estimate the resulting elasticity parameters [28]. A DMN model can be trained for a set of morphologies that is believed to be representative of a specific combination of constituents. The results shown in Table 5.1 indicate that a Voigt or Reuss based analytical solution is less accurate even when compared to DMNs evaluated on datasets they were not trained for. This project has only evaluated morphologies where the weight percentage is 50-50 of two constituents. This is one of the most extreme cases and gives very varied structures even when the constants dictating the phase-field simulations remains the same, as can be seen in Figure 3.3.

# Bibliography

[1] F. Fang, Z. Li, F. Luo, and C. Xiao, “Discriminator modification in gan for text-to-image generation.” *2022 IEEE International Conference on Multimedia and Expo (ICME), Multimedia and Expo (ICME), 2022 IEEE International Conference on*, pp. 1 – 6, 2022.

[2] V. Liu, J. Vermeulen, G. Fitzmaurice, and J. Matejka, “3dall-e: Integrating text-to-image ai in 3d design workflows.” 2022.

[3] Wikipedia contributors, “Stable diffusion — Wikipedia, the free encyclopedia,” 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Stable_Diffusion&oldid=1170803489 [Online; accessed 17-August-2023].

[4] ——, “Dall-e — Wikipedia, the free encyclopedia,” 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=DALL-E&oldid=1168785459 [Online; accessed 17-August-2023].

[5] M. Yin, G. Karniadakis, E. Zhang, and Y. Yu, “Interfacing finite elements with deep neural operators for fast multiscale modeling of mechanics problems.” *Computer Methods in Applied Mechanics and Engineering*, vol. 402, 2022.

[6] J. Yvonnet, *Computational Homogenization of Heterogeneous Materials with Finite Elements. [electronic resource]. Chapter 9*, ser. Solid Mechanics and Its Applications: 258. Springer International Publishing, 2019.

[7] D. Bishara, Y. Xie, W. K. Liu, and S. Li, “A state-of-the-art review on machine learning-based multiscale modeling, simulation, homogenization and design of materials.” *Archives of Computational Methods in Engineering*, vol. 30, no. 1, p. 191, 2023.

[8] R. Hambli, H. Katerchi, and C.-L. Benhamou, “Multiscale methodology for bone remodelling simulation using coupled finite element and neural network computation.” *Biomechanics and Modeling in Mechanobiology*, vol. 10, no. 1, pp. 133–145 – 145, 2011.

[9] Z. Liu, C. T. Wu, and M. Koishi, “A deep material network for multiscale topology learning and accelerated nonlinear modeling of heterogeneous materials,” *Comput. Methods Appl. Mech. Engrg.*, no. 345, pp. 1138–1168, 2019.

[10] Z. Liu, H. Wei, T. Huang, and C. T. Wu, "Intelligent multiscale simulation based on process-guided composite database," 2020. [Online]. Available: https://arxiv.org/abs/2003.09491

[11] S. Gajek, M. Schneider, and T. Böhlke, "On the micromechanics of deep material networks." *Journal of the Mechanics and Physics of Solids*, vol. 142, 2020.

[12] T. Huang, Z. Liu, C. Wu, and W. Chen, "Microstructure-guided deep material network for rapid nonlinear material modeling and uncertainty quantification." *Computer Methods in Applied Mechanics and Engineering*, vol. 398, 2022.

[13] J. Kochmann, S. Wulfinghoff, L. Ehle, J. Mayer, B. Svendsen, and S. Reese, "Efficient and accurate two-scale fe-fft-based prediction of the effective material behavior of elasto-viscoplastic polycrystals," *Computational Mechanics*, vol. 61, no. 6, pp. 751–764, Jun 2018.

[14] J. Shen and X. Zhou, "Least squares support vector machine for constitutive modeling of clay," *International Journal of Engineering*, vol. 28, no. 11, pp. 1571–1578, 2015, url: https://www.ije.ir/article$_7$2612*.html*.

[15] T. Huang, Z. Liu, C. Wu, and W. Chen, "Microstructure-guided deep material network for rapid nonlinear material modeling and uncertainty quantification," *Computer Methods in Applied Mechanics and Engineering*, vol. 398, pp. 115–197, 2022, doi: https://doi.org/10.1016/j.cma.2022.115197.

[16] G. Cuddalorepatta and A. Dasgupta, "Multi-scale modeling of the viscoplastic response of as-fabricated microscale pb-free sn3.0ag0.5cu solder interconnects," *Acta Materialia*, vol. 58, no. 18, pp. 5989–6001, 2010, doi: https://doi.org/10.1016/j.actamat.2010.07.016.

[17] G. Zeng, S. Liu, S. McDonald, Q. Gu, Z. Zheng, H. Yasuda, and K. Nogita, "Investigation on the solidifcation and phase transformation in pb-free solders using in situ synchrotron radiography and difraction: A review," *Acta Metallurgica Sinica*, vol. 35, pp. 49–66, 2021.

[18] G. Zeng, M. Callaghan, S. McDonald, H. Yasuda, and K. Nogita, "In situ studies revealing dendrite and eutectic growth during the solidification of sn-0.7cu-0.5ag pb-free solder alloy," *Journal of alloys and compounds*, no. 797, pp. 804–810, 2019.

[19] L. Zhang, J.-g. Han, C.-w. He, and Y.-h. Guo, "Reliability behavior of lead-free solder joints in electronic components," *Journal of Materials Science: Materials in Electronics*, vol. 24, no. 1, pp. 172–190, Jan 2013.

[20] S. Jia, L. Huaxin, S. Shaofu, H. Juntao, T. Chunyu, Z. Lingyan, and B. Hailong, "Pattern formation by spinodal decomposition in ternary lead-free sn-ag-cu solder alloy." *Metals*, vol. 12, no. 1640, 2022.

[21] S. B. Biner, *Programming Phase-Field Modeling.* Springer International Publishing, 2017.

[22] S. Theodoridis, *Machine Learning : A Bayesian and Optimization Perspective.*, ser. .NET Developers Series. Elsevier Science Technology, 2015.

[23] K. P. Murphy, *Machine Learning : A Probabilistic Perspective.*, ser. Adaptive Computation and Machine Learning Ser. MIT Press, 2012.

[24] J. Yvonnet, *Computational Homogenization of Heterogeneous Materials with Finite Elements. [electronic resource]. Chapter 4*, ser. Solid Mechanics and Its Applications: 258. Springer International Publishing, 2019.

[25] G. T. Gilbert, "Positive definite matrices and sylvester's criterion." *The American Mathematical Monthly*, vol. 98, no. 1, p. 44, 1991.

[26] Y. Yu, L. Zhang, L. Chen, and Z. Qin, "Adversarial samples generation based on rmsprop." *2021 IEEE 6th International Conference on Signal and Image Processing (ICSIP), Signal and Image Processing (ICSIP), 2021 IEEE 6th International Conference on*, pp. 1134 – 1138, 2021.

[27] M. Maleki, J. Cugnoni, and J. Botsis, "Multi-scale modeling of elasto-plastic response of snagcu lead-free solder alloys at different ageing conditions: Effect of microstructure evolution, particle size effects and interfacial failure." *Materials Science Engineering A*, vol. 661, pp. 132 – 144, 2016.

[28] O. N. Senkov, J. D. Miller, D. B. Miracle, and C. Woodward, "Accelerated exploration of multi-principal element alloys with solid solution phases," *Nature Communications*, vol. 6, no. 1, p. 6529, Mar 2015, doi: 10.1038/ncomms7529.

# A

# Gradients for back propagation

$D^1$ is the compliance of the first child node, $D^2$ is the compliance of the second child node, both include their biases.

$$\frac{\partial D^r}{\partial D^1_{11}} = \frac{1}{\Gamma} \begin{vmatrix} \frac{f_1 D^2_{11} D^2_{11}}{\Gamma} & f_2(D^2_{12} - D^r_{12}) & f_2(D^2_{13} - D^r_{13}) \\ & \frac{f_1 (f_2)^2 (D^1_{12} - D^2_{12})^2}{\Gamma} & \frac{f_1 (f_2)^2 (D^1_{13} - D^2_{13})(D^1_{12} - D^2_{12})}{\Gamma} \\ \text{Symmetry} & & \frac{f_1 (f_2)^2 (D^1_{13} - D^2_{13})^2}{\Gamma} \end{vmatrix} \tag{A.1}$$

$$\frac{\partial D^r}{\partial D^1_{12}} = \begin{vmatrix} 0 & \frac{f_1 D^2_{11}}{\Gamma} & 0 \\ & \frac{-2 f_1 f_2 (D^1_{12} - D^2_{12})^2}{\Gamma} & \frac{-f_1 f_2 (D^1_{13} - D^2_{13})}{\Gamma} \\ \text{Symmetry} & & 0 \end{vmatrix} \tag{A.2}$$

$$\frac{\partial D^r}{\partial D^1_{13}} = \begin{vmatrix} 0 & 0 & \frac{f_1 D^2_{11}}{\Gamma} \\ & 0 & \frac{-f_1 f_2 (D^1_{12} - D^2_{12})}{\Gamma} \\ \text{Symmetry} & & \frac{-2 f_1 f_2 (D^1_{13} - D^2_{13})}{\Gamma} \end{vmatrix} \tag{A.3}$$

$$\frac{\partial D^r}{\partial D^1_{22}} = \begin{vmatrix} 0 & 0 & 0 \\ & f_1 & 0 \\ \text{Symmetry} & & 0 \end{vmatrix}, \frac{\partial D^r}{\partial D^1_{23}} = \begin{vmatrix} 0 & 0 & 0 \\ & 0 & f_1 \\ \text{Symmetry} & & 0 \end{vmatrix}, \frac{\partial D^r}{\partial D^1_{33}} = \begin{vmatrix} 0 & 0 & 0 \\ & 0 & 0 \\ \text{Symmetry} & & f_1 \end{vmatrix} \tag{A.4}$$

$$\Gamma = f_1 D^2_{11} + f_2 D^1_{11}, \quad f_2 = 1 - f_1 \tag{A.5}$$

$\frac{\partial D^r}{\partial D^2}$ is the same but with the phase index switched, $D^1 \leftrightarrow D^2$ and $f_1 \leftrightarrow f_2$.

$\frac{\partial \bar{D}^r_{ij}}{\partial f_1}$:

$$\begin{aligned}
\frac{\partial \bar{D}^r_{11}}{\partial f_1} &= \frac{1}{\Gamma}(D^1_{11} - D^2_{11})\bar{D}^r_{11} \\
\frac{\partial \bar{D}^r_{12}}{\partial f_1} &= \frac{1}{\Gamma}((D^1_{12} - D^2_{11})\bar{D}^r_{12} + D^1_{12}D^2_{11} - D^2_{12}D^1_{11}) \\
\frac{\partial \bar{D}^r_{13}}{\partial f_1} &= \frac{1}{\Gamma}((D^1_{12} - D^2_{11})\bar{D}^r_{13} + D^1_{13}D^2_{11} - D^2_{13}D^1_{11}) \\
\frac{\partial \bar{D}^r_{22}}{\partial f_1} &= D^1_{22} - D^2_{22} + \frac{1}{\Gamma^2}(f_1^2 D^2_{11} - f_2 D^1_{11})(D^1_{12} - D^2_{12})^2 \\
\frac{\partial \bar{D}^r_{23}}{\partial f_1} &= D^1_{23} - D^2_{23} + \frac{1}{\Gamma^2}(f_1^2 D^2_{11} - f_2 D^1_{11})(D^1_{13} - D^2_{13})(D^1_{12} - D^2_{12}) \\
\frac{\partial \bar{D}^r_{33}}{\partial f_1} &= D^1_{33} - D^2_{33} + \frac{1}{\Gamma^2}(f_1^2 D^2_{11} - f_2 D^1_{11})(D^1_{13} - D^2_{13})^2
\end{aligned} \tag{A.6}$$

# B

# Supplementary results

Shows the datasets of FEM homogenized elasticity parameters $\bar{E}_1$ and $\bar{E}_2$ as functions of the analytically homogenized elasticity parameters using the Reuss and Voigt assumptions.

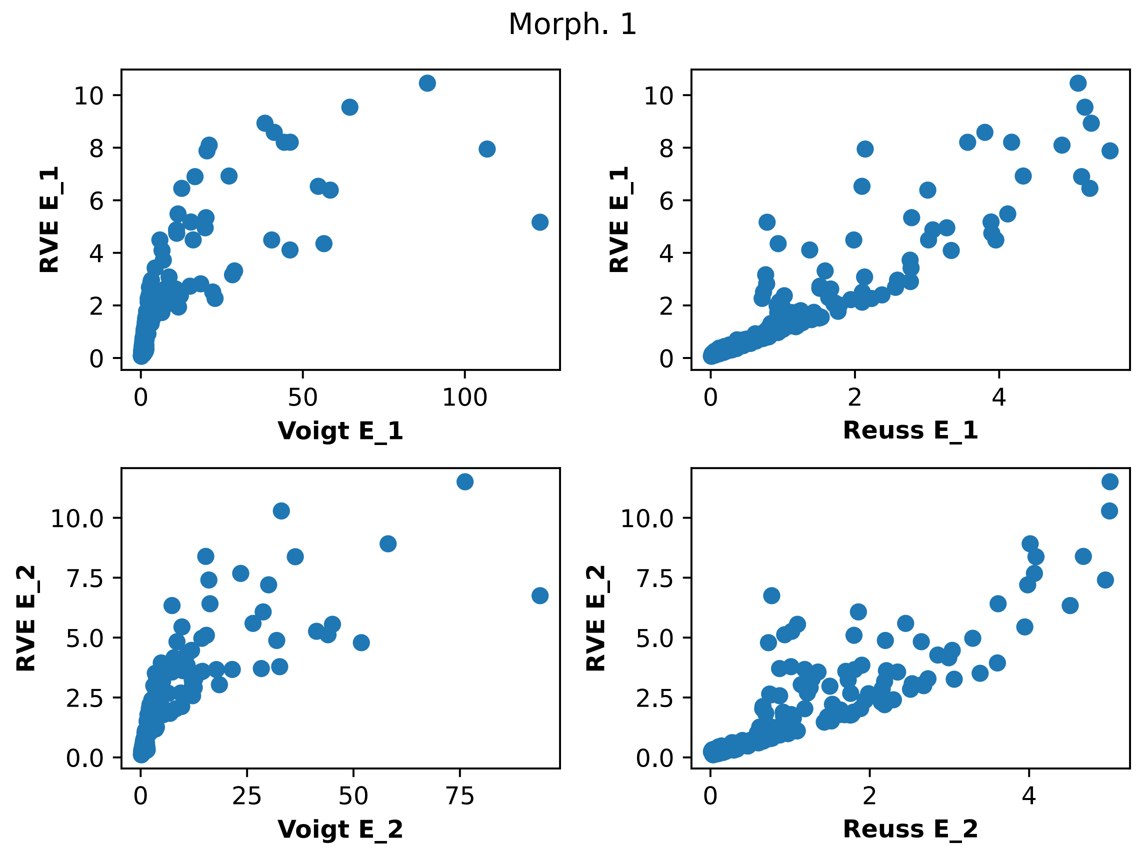


Figure B.1: Scatter plot of $\bar{E}$ values as a function of Reuss and Voigt values for Morphology 1

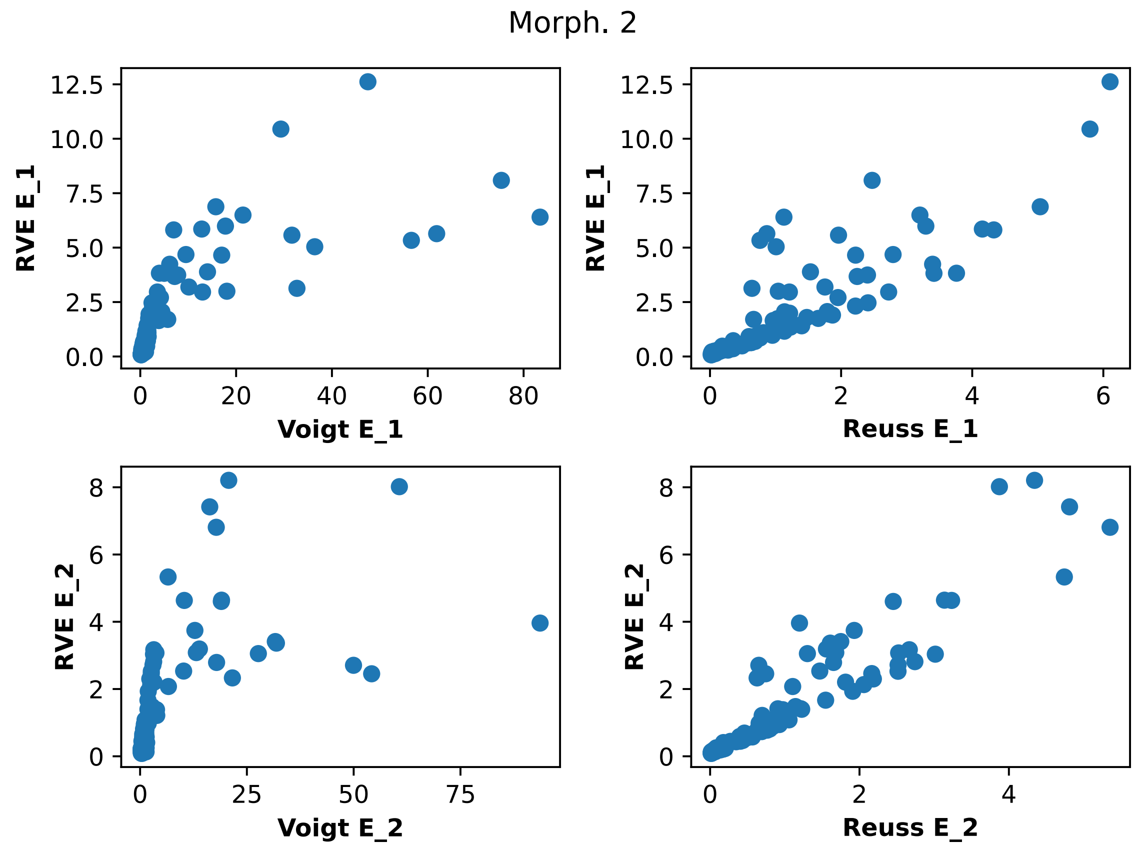


Figure B.2: Scatter plot of $\bar{E}$ values as a function of Reuss and Voigt values for Morphology 2

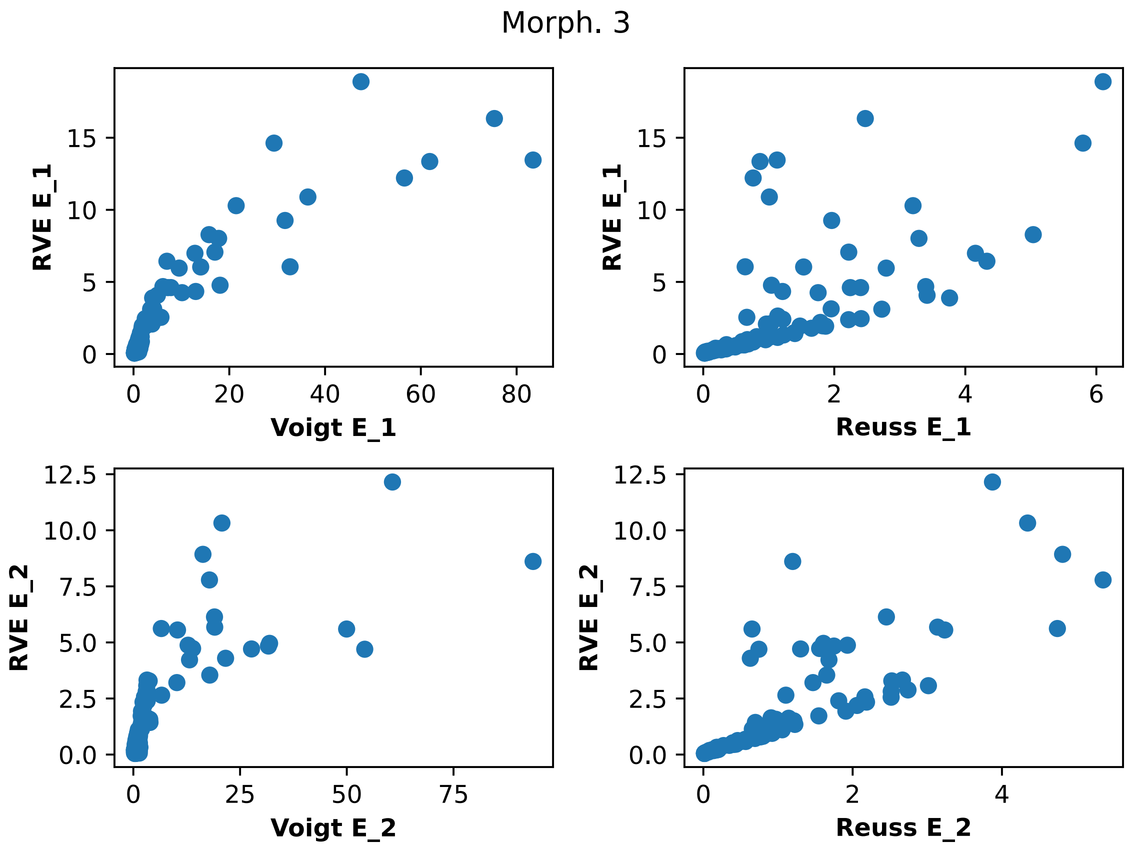


Figure B.3: Scatter plot of $\bar{E}$ values as a function of Reuss and Voigt values for Morphology 3

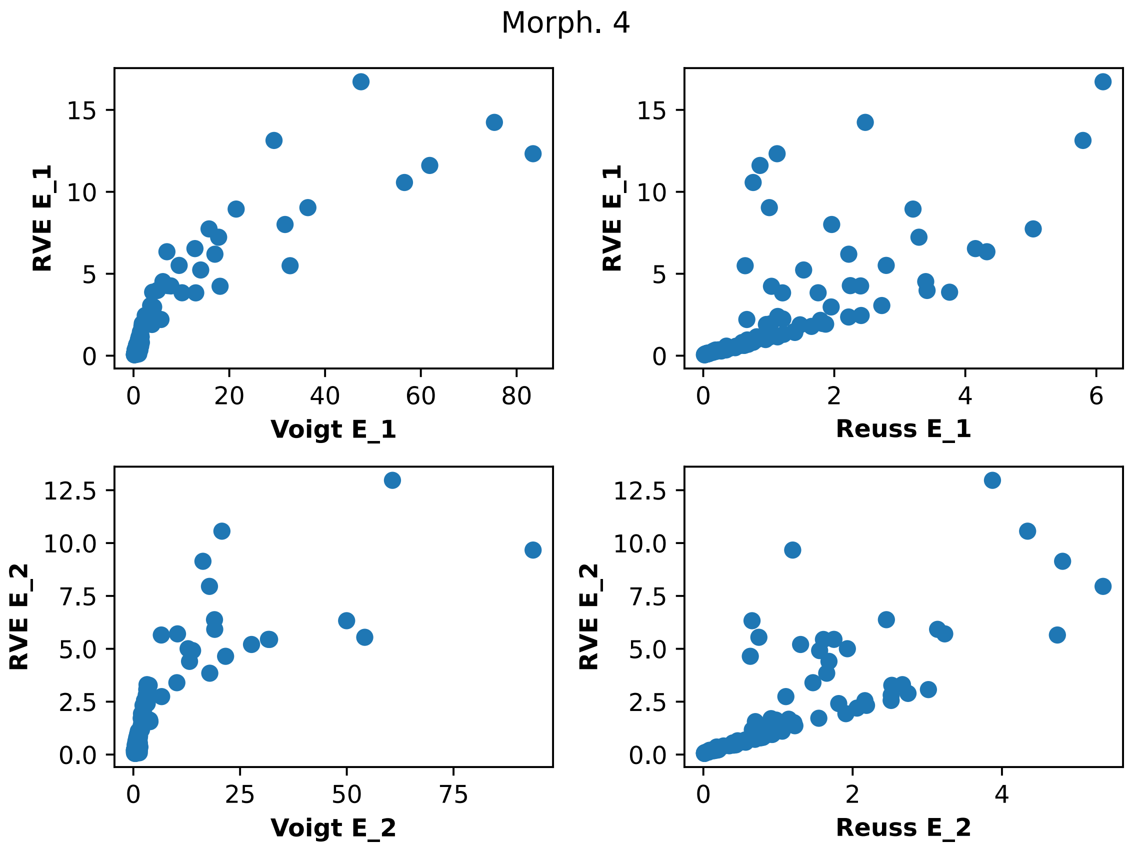


Figure B.4: Scatter plot of $\bar{E}$ values as a function of Reuss and Voigt values for Morphology 4

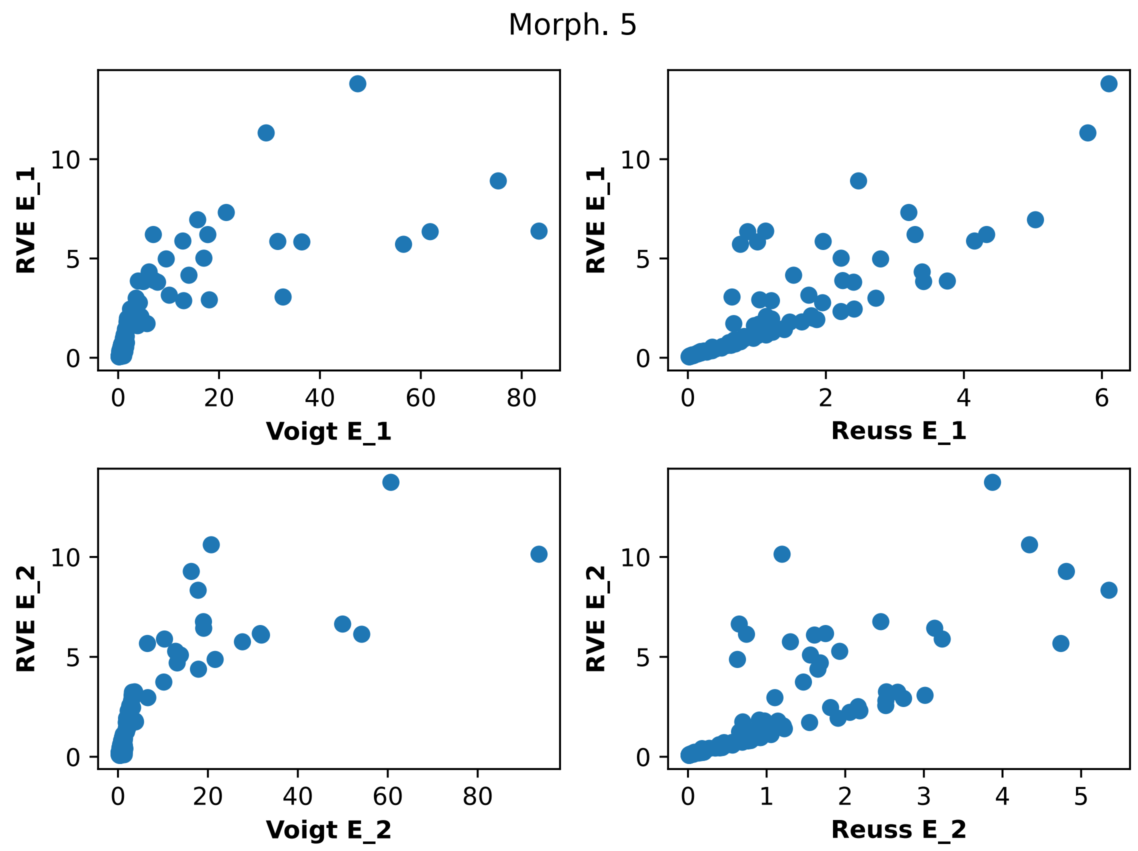


Figure B.5: Scatter plot of $\bar{E}$ values as a function of Reuss and Voigt values for Morphology 5

Training stint for a combined dataset, shows the weights diverging at the last layer before the output layer. Shows the initial cost better as well.

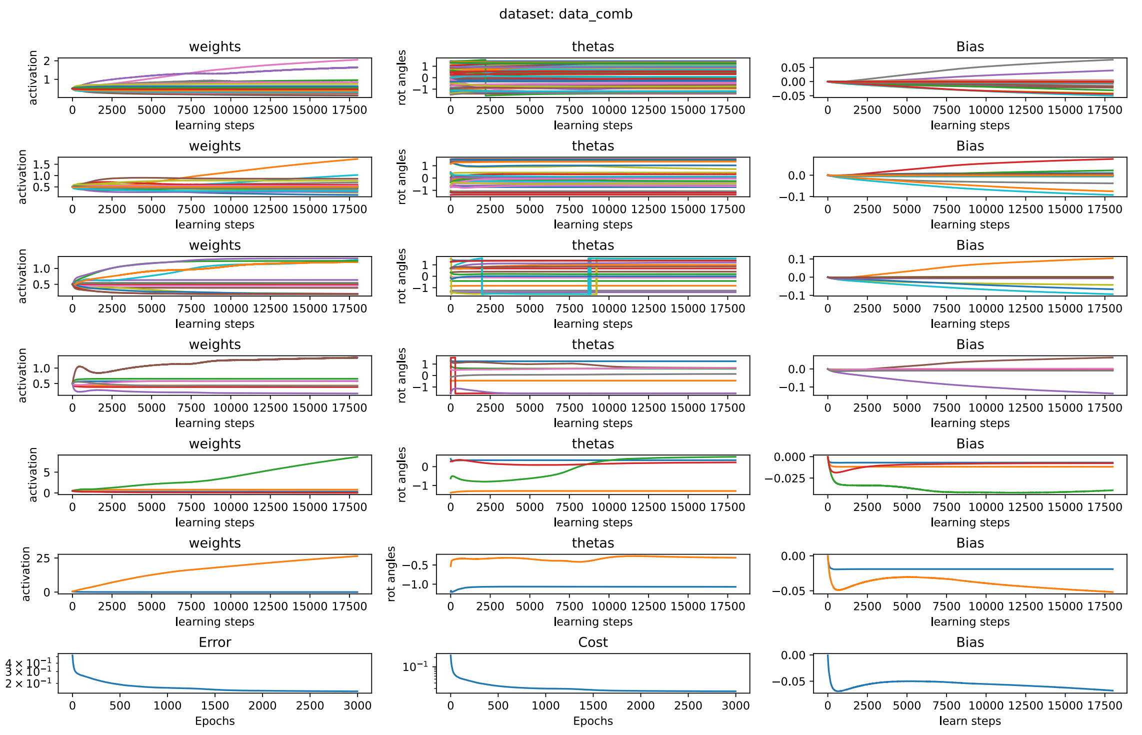


Figure B.6: Graphic displayed during training. Shows the weights, rotation angles and biases of all layers as well as the cost and error

# C

# Python code

This chapter includes the code used for this project.

## C.1 Phasefield simulations

This section includes the simulation codes implementing the pystencils module to simulate spinodal decomposition using the Cahn-Hilliard equation. [21] pages 26-27 includes a simpler implementation in matlab.

```
from pystencils.session import *
import shapefile
import pandas as pd
import fiona
import ezdxf
import numpy as np


def init(value=0.5, noise=0.02):
    for b in dh.iterate():
        b['c'].fill(value)
        np.add(b['c'], noise*np.random.rand(*b['c'].shape), out=b['c'])


def timeloop(steps=120000):
    c_sync = dh.synchronization_function(['c'])
    _sync = dh.synchronization_function(['mu'])
    for t in range(steps):
        c_sync()
        dh.run_kernel(_kernel)
        _sync()
        dh.run_kernel(c_kernel)
    return dh.gather_array('c')
```

```
dh = ps.create_data_handling(domain_size=(256, 256), periodicity=True)
_field = dh.add_array('mu', latex_name='')
c_field = dh.add_array('c')

, A = sp.symbols(" A")

c = c_field.center
 = _field.center


def f(c):
    return A * c**2 * (1-c)**2

bulk_free_energy_density = f(c)
grad_sq = sum(ps.fd.diff(c, i)**2 for i in range(dh.dim))
interfacial_free_energy_density = /2 * grad_sq

free_energy_density = bulk_free_energy_density +
↪  interfacial_free_energy_density
plt.figure(figsize=(7,4))
plt.sympy_function(bulk_free_energy_density.subs(A, 0.8), (-0.2, 1.2))
plt.xlabel("c")
plt.title("Bulk free energy")

ps.fd.functional_derivative(free_energy_density, c)

discretize = ps.fd.Discretization2ndOrder(dx=1, dt=0.01)

_update_eq = ps.fd.functional_derivative(free_energy_density, c)
_update_eq = ps.fd.expand_diff_linear(_update_eq, constants=[])  # pull
↪  constant  in front of the derivatives
_update_eq_discretized = discretize(_update_eq)
print(_update_eq_discretized)

_kernel = ps.create_kernel([ps.Assignment(_field.center,
↪  _update_eq_discretized.subs(A, 1.5).subs(, 2))]).compile()
M = sp.Symbol("M")
cahn_hilliard = ps.fd.transient(c) - ps.fd.diffusion(, M)
print(cahn_hilliard)
c_update = discretize(cahn_hilliard)
c_kernel = ps.create_kernel([ps.Assignment(c_field.center,c_update.subs(M,
↪  1))]).compile()
```

```
init()
if 'is_test_run' in globals():
    timeloop(10)
    result = None
else:
    field = timeloop()
ident = "A_1-5_k2_1-2MP-s_4"
f_min, f_max = np.min(field), np.max(field)
field = (field - f_min) / (f_max - f_min)
x = np.linspace(0,255,256)
[X,Y] = np.meshgrid(x,x)
cs = plt.contour(field,levels=[0.5])
plt.savefig(ident+'contour_plot.png')
a = cs.collections[0].get_paths()
f = open("myfile_4.txt", 'w')


for i,cnt in enumerate(a):
    points = cnt.vertices.tolist()
    for point in points:
        f.write(str(point)+"#")
    f.write("\n")
f.close()
# msp.add_lwpolyline(points)}}
#doc.saveas("Hatch_test.dxf")


#plt.show()

#ani = ps.plot.scalar_field_animation(timeloop, rescale=True, frames=600)

#ani.save('animation_A15_k2.avi')
```

### C.1.1 dxf file script

This small script converts the contour paths from the phase field simulation to lwpolylines and makes a dxf file.

```
import ezdxf


def file_reader(filename):
    f = open(filename, 'r')
    Lines = f.readlines()
    line = []
    for Line in Lines:
        nums = Line.split("#")
        points = []
        for num in nums:
            if num == "\n":
                line.append([points])
                break
            point = num.replace("[", '')
            point = point.replace("]",'')
            point = point.split(",")
            points.append([float(point[0]), float(point[1])])
    f.close()
    return line


lines = fr.file_reader("myfile_4.txt")
# hatch requires the DXF R2000 or later
doc = ezdxf.new("R2000")
msp = doc.modelspace()
msp.add_lwpolyline([(0,0), (0,255), (255,255), (255,0), (0,0)],)
for i, line in enumerate(lines):
    points = line[0]
    msp.add_lwpolyline(points)
doc.saveas("Morph_5.dxf")
```

## C.2 Dataset creation

This section includes the python scripts for generating a dataset using Nastran MSC.

### C.2.1 Input phase generator

Script to generate the input phases using Latin hypercube sampling. Module is used in the main script.

```
import numpy as np


def hyper_cube(n, d):
    rng = np.random.default_rng()
    samples = rng.uniform(size=(n, d))
    perms = np.tile(np.arange(1, n + 1), (d, 1))
    for i in range(d):
        rng.shuffle(perms[i, :])
    perms = perms.T
    samples = (perms - samples) / n
    return samples


def comp_mats(sample):
    E_1 = np.zeros((3,3))
    E_2 = np.zeros((3,3))

    uni_11 = sample[0]*2 - 1
    uni_11_2 = sample[1]*2 - 1
    uni_44 = sample[2]*8 - 4

    uni_nu_1 = sample[3]*0.4 + 0.3
    uni_nu_2 = sample[4]*0.4 + 0.3

    uni_G_1 = sample[5]*0.25 + 0.25
    uni_G_2 = sample[6]*0.25 + 0.25

    E_1[0,0] = np.sqrt(1/10**(uni_11))
    E_1[1,1] = 1/E_1[0,0]
    E_2[1,1] = np.sqrt((10**(uni_11_2))*(10**uni_44))
    E_2[0,0] = (10**uni_44)/E_2[1,1]

    nu_1 = uni_nu_1*np.sqrt(E_1[1,1]/E_1[0,0])
    nu_2 = uni_nu_2*np.sqrt(E_2[1,1]/E_2[0,0])

    G_1 = uni_G_1*np.sqrt(E_1[1,1]*E_1[0,0])

    G_2 = uni_G_2 * np.sqrt(E_2[1, 1] * E_2[0, 0])

    D_1 = np.zeros((3,3))
    D_1[0,0] = 1/E_1[0,0]
    D_1[1,1] = 1/E_1[1,1]
    D_1[0,1] = -nu_1/E_1[1,1]
    D_1[1,0] = D_1[0,1]
    D_1[2,2] = 1/(2*G_1)

    D_2 = np.zeros((3,3))
    D_2[0,0] = 1/E_2[0,0]
    D_2[1,1] = 1/E_2[1,1]
    D_2[0,1] = -nu_2/E_2[1,1]
    D_2[1,0] = D_2[0,1]
    D_2[2,2] = 1/(2*G_2)

    return D_1, D_2


def phaseinator():
    n = 1000
    d = 7
    hyperCube = hyper_cube(n,d)
    D_1 = []
    D_2 = []

    for i, samples in enumerate(hyperCube):
        D_1_temp, D_2_temp = comp_mats(samples)
        D_1.append(D_1_temp)
        D_2.append(D_2_temp)

    return D_1, D_2
```

### C.2.2 Phase class

Module contains 2 classes. Phase is a representation of an input phase, micro is a representation of a 2-phase heterogeneous micro structure.

```
import re
import numpy as np
import sympy as sm
from sympy.solvers import solve
import subprocess
import scipy
from scipy import optimize
import time
import os


class PhaseMat:
    """
    Object representing a phase of a material
    """
    def __init__(self, D):
        E_1 = 1/D[0,0]
        E_2 = 1/D[1,1]
        nu_12 = -D[1,0]*E_1
        G_12 = 1/(2*D[2,2])

        self.E_1 = list("%.6f" % E_1)
        while len(self.E_1) > 8:
            self.E_1.pop(-1)
        self.E_1 = "".join(self.E_1)

        self.E_2 = list("%.6f" % E_2)
        while len(self.E_2) > 8:
            self.E_2.pop(-1)
        self.E_2 = "".join(self.E_2)

        self.nu_12 = list("%.6f" % nu_12)
        while len(self.nu_12) > 8:
            self.nu_12.pop(-1)
        self.nu_12 = "".join(self.nu_12)

        self.G_12 = list("%.6f" % G_12)
        while len(self.G_12) > 8:
            self.G_12.pop(-1)
        self.G_12 = "".join(self.G_12)

        self.D = D
        self.C = np.ndarray((1, 4))
        self.C[0] = [E_1, E_2, nu_12, G_12]

    def __str__(self):
        string = [" "]
        for i in range(3):
            for j in range(3):
                string.append(str(self.D[i, j]))
                string.append(" ")
        return "".join(string)


class Micro:
    def __init__(self, phase_1, phase_2, test_nr, grid_data, e_area):
        self.strain = np.zeros((1,3))
        self.strain[0] = [0.04, 0.04, 0.04*np.sqrt(2)]
        self.vol_frac = 0.5
        self.grid_data = grid_data
        self.e_area = e_area
        self.n_el = len(e_area)  # number of elements
        # 3 start files, one for each load case
        self.start_file_x = "new_orig_x_10.2"
        self.start_file_y = "new_orig_y_10.2"
        self.start_file_xy = "new_orig_xy_10.2"
        self.phase_1 = phase_1
        self.phase_2 = phase_2
        self.test_nr = test_nr
        self.change_material(self.start_file_x)
        self.change_material(self.start_file_y)
        self.change_material(self.start_file_xy)
        self.run_nastran()  # generate f06 file for each load case

    def __str__(self):
        """Order: D00, D01, D02, D10, D11, D12, D20, D21, D22"""
        mic_string = [" "]
        for i in range(3):
            for j in range(3):
                mic_string.append(str(self.D[i, j]))
                mic_string.append(" ")

        return "".join(mic_string)

    def calc_stresses(self, el_nodes):
        stress_x = self.ele_stress(self.start_file_x)
        stress_y = self.ele_stress(self.start_file_y)
        stress_xy = self.ele_stress(self.start_file_xy)

        self.stress_x = self.calc_stress(stress_x)
        self.stress_y = self.calc_stress(stress_y)
        self.stress_xy = self.calc_stress(stress_xy)

        # self.stress_x = self.sort_forces(self.start_file_x)
        # self.stress_y = self.sort_forces(self.start_file_y)
        # self.stress_xy = self.sort_forces(self.start_file_xy)
        self.D = self.calc_comp_mat()
        C = np.zeros((1,4))
        C[0,0] = 1/self.D[0,0]
        C[0,1] = 1/self.D[1,1]
        C[0,2] = -C[0,0]*self.D[1,0]
        C[0,3] = 1/(2*self.D[2,2])
        self.C = C

    def change_material(self, start_file):
        # for orthotropic materials
        with open(f'nastran_input/{start_file}.bdf', 'r') as file:
            # read a list of lines into data
            data = file.readlines()
        next_line = False
        for i, line in enumerate(data):
            if next_line == "Mat_1":
                data[i] = (
                    f'MAT8           7{self.phase_1.E_1}{self.phase_1.E_2}{⌋
                    ↪  self.phase_1.nu_12}{self.phase_1.G_12}1.0     1.0
                    ↪          \n')
            elif next_line == "Mat_2":
                data[i] = (
                    f'MAT8           8{self.phase_2.E_1}{self.phase_2.E_2}{⌋
                    ↪  self.phase_2.nu_12}{self.phase_2.G_12}1.0     1.0
                    ↪          \n')
            if re.findall("HWCOLOR MAT                   7       9", line):
            ↪    # Find pattern that starts with "pts_time:"
                next_line = "Mat_1"
            elif re.findall("HWCOLOR MAT                   8       8",
            ↪   line):
                next_line = "Mat_2"
            else:
                next_line = False
        with open(f'nastran_output/{start_file}_{self.test_nr}.bdf',
        ↪   mode='w') as file:
            file.writelines(data)

    def run_nastran(self):
        """
        run the generated .bdf files in nastran MSC
        Move generated .f06 file to \nastran_sol
        """
        # call os with start file x and self.test_nr
        p1 = subprocess.call(['C:\\Program
        ↪   Files\\MSC.Software\\MSC_Nastran\\2021.3\\bin\\nastranw.exe',
                                 f'C:\\Users\\u086939\\PycharmProjects\\pythonP⌋
                                 ↪   roject\\nastran_output\\{self.start_file_x⌋
                                 ↪   }_{self.test_nr}.bdf'])
        # call os with start file y and self.test_nr
        p2 = subprocess.call(['C:\\Program
        ↪  Files\\MSC.Software\\MSC_Nastran\\2021.3\\bin\\nastranw.exe',
                                f'C:\\Users\\u086939\\PycharmProjects\\pythonP⌋
                                ↪  roject\\nastran_output\\{self.start_file_y⌋
                                ↪  }_{self.test_nr}.bdf'])

        # call os with start file xy and self.test_nr
        p3 = subprocess.call(['C:\\Program
        ↪  Files\\MSC.Software\\MSC_Nastran\\2021.3\\bin\\nastranw.exe',
                                f'C:\\Users\\u086939\\PycharmProjects\\pythonP⌋
                                ↪  roject\\nastran_output\\{self.start_file_x⌋
                                ↪  y}_{self.test_nr}.bdf'])

    def move_f06_files(self):
        os.replace(f"C:\\Users\\u086939\\PycharmProjects\\pythonProject\\{s⌋
        ↪  elf.start_file_x}_{self.test_nr}.f06",
                   f"C:\\Users\\u086939\\PycharmProjects\\pythonProject\\na⌋
                   ↪  stran_sol\\{self.start_file_x}_{self.test_nr}.f06")
        os.replace(f"C:\\Users\\u086939\\PycharmProjects\\pythonProject\\{s⌋
        ↪  elf.start_file_y}_{self.test_nr}.f06",
                   f"C:\\Users\\u086939\\PycharmProjects\\pythonProject\\na⌋
                   ↪  stran_sol\\{self.start_file_y}_{self.test_nr}.f06")
        os.replace(f"C:\\Users\\u086939\\PycharmProjects\\pythonProject\\{s⌋
        ↪  elf.start_file_xy}_{self.test_nr}.f06",
                   f"C:\\Users\\u086939\\PycharmProjects\\pythonProject\\na⌋
                   ↪  stran_sol\\{self.start_file_xy}_{self.test_nr}.f06")

    def calc_stress(self, e_stress):
        tot_area = 255 * 255
        tot_stress = np.zeros((1,3))
        for e_s in e_stress:
            el_area = self.e_area[np.where(self.e_area[:, 0] == e_s[0]), 1]
            for i in range(3):
                tot_stress[0,i] += el_area * e_s[i + 1]
        avg_stress = tot_stress/tot_area
        avg_stress[0, 2] *= np.sqrt(2)
        return avg_stress

    def read_reac_force(self, start_file):
        with open(f'nastran_sol/{start_file}_{self.test_nr}.f06', 'r') as
        ↪  file:
            in_data = file.readlines()
        disp_flag = False
        out_data = []
        in_data_iter = iter(in_data)
        for line in in_data_iter:
            if line[0] == '1':
                disp_flag = False
            if re.findall(" \*\*\*", line):
                disp_flag = False
            if disp_flag:
                out_data.append(line)
            if re.findall(
                    "                                F O R C E S   O F   S I
                    ↪  N G L E - P O I N T   C O N S T R A I N T",
                    line):
                disp_flag = True
                next(in_data_iter)
                next(in_data_iter)
        data = np.ndarray((342, 7))
        for i, s in enumerate(out_data):
            s_out = re.findall("[-+]?[.]?[\d]+(?:,\d\d\d)*[\.]?\d*(?:[eE][-⌋
            ↪  +]?\d+)?",
            ↪  s)
            data[i] = [float(val) for val in s_out]
        return data

    def sort_forces(self, start_file):
        reac_data = self.read_reac_force(start_file)
        top_side = np.zeros((1, 2))
        bot_side = np.zeros((1, 2))
        left_side = np.zeros((1, 2))
        right_side = np.zeros((1, 2))
        for i, node in enumerate(reac_data[:, 0]):
            reac_x = reac_data[i, 1]
            reac_y = reac_data[i, 2]
            grid_value = self.grid_data[np.where(self.grid_data[:, 0] ==
            ↪  node)]
            if grid_value[0, 1] == 0:
                if grid_value[0, 2] == 0:
                    bot_side[0, 1] += reac_y
                    left_side[0, 0] += reac_x
                    # corner point
                elif grid_value[0, 2] == 255:
```

```
                left_side[0, 0] += reac_x
                top_side[0, 1] += reac_y
                # corner point
            else:
                left_side[0, 0] += reac_x
                left_side[0, 1] += reac_y
        elif grid_value[0, 1] == 255:
            if grid_value[0, 2] == 0:
                right_side[0, 0] += reac_x
                bot_side[0, 1] += reac_y
                # corner point
            elif grid_value[0, 2] == 255:
                right_side[0, 0] += reac_x
                top_side[0, 1] += reac_y
                # corner point
            else:
                right_side[0, 0] += reac_x
                right_side[0, 1] += reac_y
        elif grid_value[0, 2] == 255:
            top_side[0, 0] += reac_x
            top_side[0, 1] += reac_y
        elif grid_value[0, 2] == 0:
            bot_side[0, 0] += reac_x
            bot_side[0, 1] += reac_y
        else:
            print("grid_value not on boundary")
    stress_x = np.abs((left_side[0, 0] * -1 + right_side[0, 0]) / 2)/255
    stress_y = np.abs((bot_side[0, 1] * -1 + top_side[0, 1]) / 2)/255
    stress_xy = (np.abs((np.abs(bot_side[0, 0]) + np.abs(top_side[0,
    ↪   0]) + np.abs(left_side[0, 1]) + np.abs(right_side[0, 1])) /
    ↪   4))/255
    stress = np.zeros((1, 3))
    stress[0] = [stress_x, stress_y, stress_xy]
    return stress

def ele_stress(self, start_file):
    with open(f'nastran_sol\\{start_file}_{self.test_nr}.f06', 'r') as
    ↪   file:
        in_data = file.readlines()
    disp_flag = False
    out_data = []
    in_data_iter = iter(in_data)
    for line in in_data_iter:
        if line[0] == '1':
            disp_flag = False
        if re.findall(" \*\*\*", line):
            disp_flag = False
        if disp_flag:
            out_data.append(line)
            next(in_data_iter)  # skip duplicate
        if re.findall(
                "  ELEMENT      FIBER                STRESSES IN
                ↪  MATERIAL COORD SYSTEM              PRINCIPAL
                ↪  STRESSES",
                line):
            disp_flag = True
            next(in_data_iter)
    data = np.ndarray((self.n_el, 4))
    for i, s in enumerate(out_data):
        s_out = re.findall("[-+]?[.]?[\d]+(?:,\d\d\d)*[\.]?\d*(?:[eE][-
        ↪  +]?\d+)?",
        ↪  s)
        nums = [float(val) for val in s_out]
        data[i] = [nums[1], nums[3], nums[4], nums[5]]
    return data

def rotate_stress_field(self, stress_data, el_nodes):
    grid_data = self.grid_data
    new_stress_vec = np.zeros((len(el_nodes), 4))
    for i, el_stress in enumerate(stress_data):
        nodes = el_nodes[np.where(el_nodes[:, 0] == el_stress[0])]
        xy_node_1 = grid_data[np.where(grid_data[:, 0] == nodes[0, 1])]
        xy_node_2 = grid_data[np.where(grid_data[:, 0] == nodes[0, 2])]
        x_hat = np.zeros((1, 2))
        x_hat[0, 0] = xy_node_2[0, 1] - xy_node_1[0, 1]
        x_hat[0, 1] = xy_node_2[0, 2] - xy_node_1[0, 2]
        x_hat = x_hat / np.linalg.norm(x_hat)
        rot_mat = np.zeros((2, 2))
        rot_mat[0, 0] = x_hat[0, 0]
        rot_mat[0, 1] = -x_hat[0, 1]
        rot_mat[1, 1] = x_hat[0, 0]
        rot_mat[1, 0] = x_hat[0, 1]
        stress_mat = np.zeros((2, 2))
        stress_mat[0, 0] = el_stress[1]
        stress_mat[0, 1] = el_stress[3]
        stress_mat[1, 0] = el_stress[3]
        stress_mat[1, 1] = el_stress[2]
        new_stress_mat = np.matmul(np.matmul(rot_mat, stress_mat),
         ↪  np.transpose(rot_mat))
        new_stress_vec[i, :] = [nodes[0, 0], new_stress_mat[0, 0],
         ↪  new_stress_mat[1, 1], new_stress_mat[0, 1]]
    temp = np.max(new_stress_vec[:,1])
    temp2 = np.max(new_stress_vec[:, 2])
    return new_stress_vec

def calc_comp_mat(self):
    D = sm.symarray("D", (3,3))
    eqs = []
    eqs.extend(np.matmul(D, self.stress_x[0, :]) - [self.strain[0,0],
     ↪  0, 0])
    eqs.extend(np.matmul(D, self.stress_y[0, :]) - [0,
     ↪  self.strain[0,1], 0])
    eqs.extend(np.matmul(D, self.stress_xy[0, :]) - [0, 0,
     ↪  self.strain[0,2]])
    sol = solve(eqs)
    vals = [val for val in sol.values()]
    D = np.zeros((3, 3))
    D[0,0] = vals[0]
    D[0, 1] = vals[1]
    D[0, 2] = vals[2]
    D[1, 0] = vals[3]
    D[1, 1] = vals[4]
    D[1, 2] = vals[5]
    D[2, 0] = vals[6]
    D[2, 1] = vals[7]
    D[2, 2] = vals[8]

    D_2 = np.zeros((3, 3))
    D_2[0,0] =
        ↪  self.strain[0,0]/(self.stress_x[0,0]*(1-(self.stress_x[0,1]*sel ⌋
        ↪  f.stress_y[0,0])/(self.stress_x[0,0]*self.stress_y[0,1])))
    D_2[0,1] = -D_2[0,0]*self.stress_y[0,0]/self.stress_y[0,1]
    D_2[1,0] = D_2[0,1]
    D_2[1,1] = self.strain[0,1]/self.stress_y[0,1] -
     ↪  D[0,1]*self.stress_y[0,0]/self.stress_y[0,1]
    D_2[2,2] = self.strain[0,2]/self.stress_xy[0,2]
    return D

def elast_bounds(self):
    """
    calculate lower Reuss value and upper Voigt value for the
↪   representative elasticity parameters
    """

    D_voigt = np.zeros((3, 3))
    D_voigt[0, 0] = 1 / (self.vol_frac / self.phase_1.D[0, 0] + (1 -
    ↪  self.vol_frac) / self.phase_2.D[0, 0])
    D_voigt[0, 1] = 1 / (self.vol_frac / self.phase_1.D[0, 1] + (1 -
    ↪  self.vol_frac) / self.phase_2.D[0, 1])
    D_voigt[1, 0] = D_voigt[0,1]
    D_voigt[1, 1] = 1 / (self.vol_frac / self.phase_1.D[1, 1] + (1 -
    ↪  self.vol_frac) / self.phase_2.D[1, 1])
    D_voigt[2, 2] = 1 / (self.vol_frac / self.phase_1.D[2, 2] + (1 -
    ↪  self.vol_frac) / self.phase_2.D[2, 2])

    C_voigt = np.zeros((1, 4))
    C_voigt[0, 0] = 1 / D_voigt[0, 0]
    C_voigt[0, 1] = 1 / D_voigt[1, 1]
    C_voigt[0, 3] = 1 / (2*D_voigt[2, 2])
    C_voigt[0, 2] = -C_voigt[0, 0] * D_voigt[1, 0]

    D_reuss = np.zeros((3,3))
    D_reuss[0, 0] = self.vol_frac * self.phase_1.D[0, 0] + (1 -
    ↪  self.vol_frac) * self.phase_2.D[0, 0]
    D_reuss[0, 1] = self.vol_frac * self.phase_1.D[0, 1] + (1 -
    ↪  self.vol_frac) * self.phase_2.D[0, 1]
    D_reuss[1, 0] = D_reuss[0, 1]
    D_reuss[1, 1] = self.vol_frac * self.phase_1.D[1, 1] + (1 -
    ↪  self.vol_frac) * self.phase_2.D[1, 1]
    D_reuss[2, 2] = self.vol_frac*self.phase_1.D[2, 2] +
    ↪  (1-self.vol_frac)*self.phase_2.D[2, 2]

    C_reuss = np.zeros((1, 4))
    C_reuss[0, 0] = 1 / D_reuss[0, 0]
    C_reuss[0, 1] = 1 / D_reuss[1, 1]
    C_reuss[0, 3] = 1 / (2*D_reuss[2, 2])
    C_reuss[0, 2] = -C_reuss[0, 0] * D_reuss[1, 0]

    self.D_voigt = D_voigt
    self.C_voigt = C_voigt
    self.D_reuss = D_reuss
    self.C_reuss = C_reuss
```

```
    def check_elast_bounds(self):
        D_temp = self.D.copy()
        D_temp[0,2] = 0
        D_temp[2,0] = 0
        D_temp[1,2] = 0
        D_temp[2,1] = 0
        temp = np.abs(D_temp) - np.abs(self.D_voigt)
        if (np.abs(D_temp) - np.abs(self.D_voigt)).all() >= 0:
            self.bound_check = True
        else:
            self.bound_check = False
        if (np.abs(D_temp) - np.abs(self.D_reuss)).all() <= 0:
            self.bound_check = True
        else:
            self.bound_check = False

    def check_symm(self):
        for i in range(3):
            for j in range(3):
                if np.abs(self.D[i, j] - self.D[j, i]) < 5E-4:
                    self.sym_check = True
                else:
                    self.sym_check = False
                    return
```

### C.2.3 Nastran file module

Module containing functions for the pre-processing of the .bdf files used by Nastran MSC.

```
import numpy as np
import re
import Phase


def change_material(phase_1, phase_2, test_nr):
    with open(f'nastran_input/micro_struc_hm_stress_out.bdf', 'r') as file:
        # read a list of lines into data
        data = file.readlines()
    next_line = False
    for i, line in enumerate(data):
        if next_line == "Mat_1":
            data[i] = (f'MAT8           9{phase_1.E_1}50.0    0.35    40.0
             ↪    1.0     1.0\n')
            print(1)
        elif next_line == "Mat_2":
            data[i] = (f'MAT1           {phase_2.E}           {phase_2.nu}
             ↪  \n')
            print(2)

        if re.findall("\$HWCOLOR MAT                   9       4", line):
         ↪  # Find pattern that starts with "pts_time:"
            next_line = "Mat_1"
        elif re.findall("\$HWCOLOR MAT                  10       5", line):
            next_line = "Mat_2"
        else:
            next_line = False
    with open(f'nastran_scripts/nastran_input/micro_struc_hm_tension{test_n⌋
     ↪  r}.bdf', mode='w') as
     ↪  file:
        file.writelines(data)


def read_el_stress(file_name, n_el):
    with open(f'nastran_input/{file_name}.f06', 'r') as file:
        in_data = file.readlines()
    disp_flag = False
    out_data = []
    in_data_iter = iter(in_data)
    for line in in_data_iter:
        if line[0] == '1':
            disp_flag = False
        if re.findall(" \*\*\*", line):
            disp_flag = False
        if disp_flag:
            out_data.append(line)
            next(in_data_iter)  # skip duplicate
        if re.findall("  ELEMENT      FIBER               STRESSES IN
         ↪  ELEMENT COORD SYSTEM             PRINCIPAL STRESSES", line):
            disp_flag = True
            next(in_data_iter)
    data = np.ndarray((n_el, 4))
    for i, s in enumerate(out_data):
        s_out = re.findall("[-+]?[.]?[\d]+(?:,\d\d\d)*[\.]?\d*(?:[eE][-+]?\⌟
        ↪  d+)?",
        ↪  s)
        nums = [float(val) for val in s_out]
        data[i] = [nums[1], nums[3], nums[4], nums[5]]
    return data


def read_el_strain(file_name, n_el):
    with open(f'nastran_output/{file_name}.f06', 'r') as file:
        in_data = file.readlines()
    disp_flag = False
    out_data = []
    in_data_iter = iter(in_data)
    for line in in_data_iter:
        if line[0] == '1':
            disp_flag = False
        if re.findall(" \*\*\*", line):
            disp_flag = False
        if disp_flag:
            out_data.append(line)
            next(in_data_iter)  # skip duplicate
        if re.findall("  ELEMENT      STRAIN               STRAINS IN
        ↪  ELEMENT COORD SYSTEM             PRINCIPAL  STRAINS", line):
            disp_flag = True
            next(in_data_iter)
    data = np.ndarray((n_el, 4))
    for i, s in enumerate(out_data):
        s_out = re.findall("[-+]?[.]?[\d]+(?:,\d\d\d)*[\.]?\d*(?:[eE][-+]?\⌟
        ↪  d+)?",
        ↪  s)
        nums = [float(val) for val in s_out]
        data[i] = [nums[1], nums[3], nums[4], nums[5]]
    return data


def read_grid_data(file_name):
    """
    :param filename:
    :return: point ID, x, y
    """
    #
    with open(f'nastran_input/{file_name}.bdf', 'r') as file:
        in_data = file.readlines()
    disp_flag = False
    out_data = np.ndarray((16508, 3))
    in_data_iter = iter(in_data)
    i = 0
    for line in in_data_iter:
        if re.findall("\$\$", line):
            disp_flag = False
        if disp_flag:

            gp = float(line[4:16])
            temp1 = line[24:24+8]
            if re.findall("\\.\d+-\d+", temp1):
                temp1 = 0
            s1 = float(temp1)

            temp2 = line[24 + 8:24 + 16]
            if re.findall("\\.\d+-\d+", temp2):
                temp2 = 0
            s2 = float(temp2)
            out_data[i, 0] = gp
            out_data[i, 1] = s1
            out_data[i, 2] = s2
            i += 1

        if re.findall("\$\$  GRID Data", line):
            disp_flag = True
            next(in_data_iter)
    return out_data


def linear_shear_displacement(file_name, grid_data, max_disp):
    """
    creates linear pure shear strain condition
    :param file_name: name of file in nastran_input folder to edit
    :param grid_data: node, x, and y values
    :return: writes a new bdf file with linearly increasing load
    """
    with open(f'nastran_input/{file_name}.bdf', 'r') as file:
        in_data = file.readlines()
    for i, line in enumerate(in_data):
        if re.findall("SPCD            ", line):
            direc = int(line[24:32])
            node = int(line[16:24])
            ind = np.where(grid_data[:, 0] == node)
            if direc == 1:
                val = grid_data[ind, 2]

            elif direc == 2:
                val = grid_data[ind, 1]
            else:
                print("fail")
            spcd = max_disp * val[0, 0] / 255000
            spcd_string = "%.5f" % spcd
            spcd_string = list(spcd_string)
            while len(spcd_string) > 7:
                spcd_string.pop(-1)
            spcd_string.insert(0, " ")
            spcd_string = "".join(spcd_string)
            node_string = list(str(node))
            while len(node_string) < 8:
                node_string.insert(0, " ")
            node_string = "".join(node_string)
            in_data[i] = f"SPCD           2{node_string}
             ↪  {direc}{spcd_string}\n"
    with open(f'nastran_input/{file_name}_{max_disp}.bdf', mode='w') as
     ↪  file:
        file.writelines(in_data)


def linear_displacement(file_name, grid_data, direc, max_disp):
    """

    :param file_name: name of file in nastran_input folder to edit
    :param grid_data: node, x, and y values
    :return: writes a new bdf file with linearly increasing load
    """
    with open(f'nastran_input/{file_name}.bdf', 'r') as file:
        in_data = file.readlines()
    for i, line in enumerate(in_data):
        if re.findall("SPCD           ", line):
            node = int(line[16:24])
            ind = np.where(grid_data[:, 0] == node)
            x = grid_data[ind, 1]
            y = grid_data[ind, 2]
            if direc == "x":
                direc_val = 1
                spcd = float(max_disp * x[0, 0] / 255000)
            elif direc == "y":
                direc_val = 2
                spcd = float(max_disp * y[0, 0] / 255000)
            else:
                print("no load direction stated")
                return
            spcd_string = "%.7f" % spcd
            spcd_string = list(spcd_string)
            while len(spcd_string) > 7:
                spcd_string.pop(-1)
            spcd_string.insert(0, " ")
            spcd_string = "".join(spcd_string)
            node_string = list(str(node))
            while len(node_string) < 8:
                node_string.insert(0, " ")
            node_string = "".join(node_string)
            in_data[i] = f"SPCD           2{node_string}
            ↪  {direc_val}{spcd_string}\n"
    with open(f'nastran_input/{file_name}_{max_disp}.bdf', mode='w') as
    ↪  file:
        file.writelines(in_data)


def change_load():
    with open(f'nastran_input/micro_struc_hm_stress_out.bdf', 'r') as file:
        # read a list of lines into data
        data = file.readlines()
    next_line = False
    for i, line in enumerate(data):
        if next_line:
            data[i] = ("+              1     1.0     0.0     0.0\n")

        if re.findall("PLOAD4         2", line):  # Find pattern
            next_line = True
        else:
            next_line = False
    with open(f'nastran_scripts/nastran_input/micro_struc_hm_tension.bdf',
    ↪  mode='w') as file:
        file.writelines(data)
```

```
def read_reac_force(file_name):
    with open(f'nastran_input/{file_name}.f06', 'r') as file:
        in_data = file.readlines()
    disp_flag = False
    out_data = []
    in_data_iter = iter(in_data)
    for line in in_data_iter:
        if line[0] == '1':
            disp_flag = False
        if re.findall(" \*\*\*", line):
            disp_flag = False
        if disp_flag:
            out_data.append(line)
        if re.findall("                              F O R C E S   O F   S
        ↪   I N G L E - P O I N T   C O N S T R A I N T", line):
            disp_flag = True
            next(in_data_iter)
            next(in_data_iter)
    data = np.ndarray((342, 7))
    for i, s in enumerate(out_data):
        s_out = re.findall("[-+]?[.]?[\d]+(?:,\d\d\d)*[\.]?\d*(?:[eE][-+]?\
        ↪   d+)?",
        ↪   s)
        data[i] = [float(val) for val in s_out]
    return data


def sort_forces(reac_data, grid_data):
    top_side = np.zeros((1, 2))
    bot_side = np.zeros((1, 2))
    left_side = np.zeros((1, 2))
    right_side = np.zeros((1, 2))
    for i, node in enumerate(reac_data[:, 0]):
        reac_x = reac_data[i, 1]
        reac_y = reac_data[i, 2]
        grid_value = grid_data[np.where(grid_data[:, 0] == node)]
        if grid_value[0, 1] == 0:
            if grid_value[0, 2] == 0:
                bot_side[0, 1] += reac_y
                left_side[0, 0] += reac_x
                # corner point
            elif grid_value[0, 2] == 255:
                left_side[0, 0] += reac_x
                top_side[0, 1] += reac_y
                # corner point
            else:
                left_side[0, 0] += reac_x
                left_side[0, 1] += reac_y
        elif grid_value[0, 1] == 255:
            if grid_value[0, 2] == 0:
                right_side[0, 0] += reac_x
                bot_side[0, 1] += reac_y
                # corner point
            elif grid_value[0, 2] == 255:
                right_side[0, 0] += reac_x
                top_side[0, 1] += reac_y
                # corner point
            else:
                right_side[0, 0] += reac_x
                right_side[0, 1] += reac_y
        elif grid_value[0, 2] == 255:
            top_side[0, 0] += reac_x
            top_side[0, 1] += reac_y
        elif grid_value[0, 2] == 0:
            bot_side[0, 0] += reac_x
            bot_side[0, 1] += reac_y
        else:
            print("grid_value not on boundary")
    return left_side, right_side, top_side, bot_side


def read_disp(file_name):
    with open(f'nastran_scripts/nastran_input/{file_name}.f06', 'r') as
     ↪  file:
        in_data = file.readlines()
    disp_flag = False
    out_data = []
    in_data_iter = iter(in_data)
    for line in in_data_iter:
        if line[0] == '1':
            disp_flag = False
        if re.findall(" \*\*\*", line):
            disp_flag = False
        if disp_flag:
            out_data.append(line)
        if re.findall("                                          D I S P
         ↪  L A C E M E N T   V E C T O R", line):
            disp_flag = True
            next(in_data_iter)
            next(in_data_iter)
    data = np.ndarray((86, 7))
    for i, s in enumerate(out_data):
        s_out = re.findall("[-+]?[.]?[\d]+(?:,\d\d\d)*[\.]?\d*(?:[eE][-+]?\⌟
         ↪  d+)?",
         ↪  s)
        data[i] = [float(val) for val in s_out]
    return data


def quad_element_nodes(file_name):
    """
    create element node matrix, each row contains element index and 4
↪   corresponding node index
    :param file_name:
    :return:
    """
    #
    with open(f'nastran_input/{file_name}.bdf', 'r') as file:
        in_data = file.readlines()
    data_flag = False
    out_data = []
    in_data_iter = iter(in_data)
    for line in in_data_iter:
        if re.findall("\$\$", line):
            data_flag = False
        if data_flag:
            if re.findall("\$", line):
                continue
            out_data.append(line)
        if re.findall("\$\$  CQUAD4 Elements", line):
            data_flag = True
            next(in_data_iter)
            next(in_data_iter)
    e_node = np.ndarray((len(out_data), 5))
    for i, s in enumerate(out_data):
        s_out = re.findall("[-+]?[.]?[\d]+(?:,\d\d\d)*[\.]?\d*(?:[eE][-+]?\⌟
         ↪  d+)?",
         ↪  s)
        data = [float(val) for val in s_out]
        e_node[i] = [data[1], data[3], data[4], data[5], data[6]]
    return e_node


def tri_element_nodes(file_name):
    """
    create element node matrix, each row contains element index and 3
↪   corresponding node index
    :param file_name:
    :return:
    """
    with open(f'nastran_input/{file_name}.bdf', 'r') as file:
        in_data = file.readlines()
    data_flag = False
    out_data = []
    in_data_iter = iter(in_data)
    for line in in_data_iter:
        if re.findall("\$\$", line):
            data_flag = False
        if data_flag:
            if re.findall("\$", line):
                continue
            out_data.append(line)
        if re.findall("\$\$  CTRIA3 Data", line):
            data_flag = True
            next(in_data_iter)
            next(in_data_iter)
    e_node = np.ndarray((len(out_data), 4))
    for i, s in enumerate(out_data):
        s_out = re.findall("[-+]?[.]?[\d]+(?:,\d\d\d)*[\.]?\d*(?:[eE][-+]?\⌋
         ↪  d+)?",
         ↪  s)
        data = [float(val) for val in s_out]
        e_node[i] = [data[1], data[3], data[4], data[5]]
    return e_node


def tri_element_area(grid_data, e_node):
    """
    Calcualte area of triangular elements
    :param grid_data: node x, and y data
    :param e_node: matrix with, element index, and 3 node index
    :return: e_area: element index, element area
    """
    e_area = np.ndarray((len(e_node), 2))
    for i, ele in enumerate(e_node):
        node_1 = np.where(grid_data[:, 0] == ele[1])
        node_2 = np.where(grid_data[:, 0] == ele[2])
        node_3 = np.where(grid_data[:, 0] == ele[3])
        M = np.ndarray((3,3))
        M[0] = [grid_data[node_1, 1], grid_data[node_1, 2], 1]
        M[1] = [grid_data[node_2, 1], grid_data[node_2, 2], 1]
        M[2] = [grid_data[node_3, 1], grid_data[node_3, 2], 1]
        area = 0.5*np.linalg.det(M)
        e_area[i] = [ele[0], area]
    return e_area


def vol_avg_stress(e_area, e_stress):
    tot_area = 255 * 255
    tot_stress = [0,0,0]
    for e_s in e_stress:
        el_area = e_area[np.where(e_area[:, 0] == e_s[0]), 1]
        for i in range(3):
            tot_stress[i] += el_area*e_s[i+1]
    avg_stress = [s/tot_area for s in tot_stress]
    return avg_stress


def vol_avg_strain(e_area, e_strain):
    tot_area = 255 * 255
    tot_strain = [0, 0, 0]
    for e_s in e_strain:
        el_area = e_area[np.where(e_area[:, 0] == e_s[0]), 1]
        for i in range(3):
            tot_strain[i] += el_area*e_s[i+1]
    avg_strain = [s/tot_area for s in tot_strain]
    return avg_strain


def quad_element_area(grid_data, e_node):
    """
    Calcualte area of triangular elements
    :param grid_data: node x, and y data
    :param e_node: matrix with, element index, and 3 node index
    :return: e_area: element index, element area
    """
    e_area = np.ndarray((len(e_node), 2))
    for i, ele in enumerate(e_node):
        node_1 = np.where(grid_data[:, 0] == ele[1])
        node_2 = np.where(grid_data[:, 0] == ele[2])
        node_3 = np.where(grid_data[:, 0] == ele[3])
        node_4 = np.where(grid_data[:, 0] == ele[4])

        M_1 = np.ndarray((2, 2))
        M_2 = np.ndarray((2, 2))
        M_3 = np.ndarray((2, 2))
        M_4 = np.ndarray((2, 2))

        M_1[0] = [grid_data[node_1, 1], grid_data[node_2, 1]]
        M_1[1] = [grid_data[node_1, 2], grid_data[node_2, 2]]

        M_2[0] = [grid_data[node_2, 1], grid_data[node_3, 1]]
        M_2[1] = [grid_data[node_2, 2], grid_data[node_3, 2]]

        M_3[0] = [grid_data[node_3, 1], grid_data[node_4, 1]]
        M_3[1] = [grid_data[node_3, 2], grid_data[node_4, 2]]

        M_4[0] = [grid_data[node_4, 1], grid_data[node_1, 1]]
        M_4[1] = [grid_data[node_4, 2], grid_data[node_1, 2]]
        area = 0.5*(np.linalg.det(M_1) + np.linalg.det(M_2) +
         ↪  np.linalg.det(M_3) + np.linalg.det(M_4))
        e_area[i] = [ele[0], area]
    return e_area


def rotate_stress_field(stress_data, grid_data, el_nodes):
    new_stress_vec = np.zeros((len(el_nodes), 4))
    for i, el_stress in enumerate(stress_data):
        nodes = el_nodes[np.where(el_nodes[:, 0] == el_stress[0])]
        temp = np.where(grid_data[:, 0] == nodes[0, 1])
        xy_node_1 = grid_data[np.where(grid_data[:, 0] == nodes[0, 1])]
        xy_node_2 = grid_data[np.where(grid_data[:, 0] == nodes[0, 2])]
        x_hat = np.zeros((1, 2))
        x_hat[0, 0] = xy_node_2[0,0] - xy_node_1[0,0]
        x_hat[0, 1] = xy_node_2[0,1] - xy_node_1[0,1]
        x_hat = x_hat/np.linalg.norm(x_hat)
        x = np.ndarray((1,2))
        x[0] = [1,0]
        rot_mat = np.zeros((2,2))
        temp = rot_mat[0,0]
        rot_mat[0,0] = np.vdot(x,x_hat)
        rot_mat[0,1] = -np.linalg.norm(np.cross(x,x_hat))
        rot_mat[1,1] = np.vdot(x,x_hat)
        rot_mat[1,0] = np.linalg.norm(np.cross(x,x_hat))
        stress_mat = np.zeros((2,2))
        stress_mat[0,0] = el_stress[1]
        stress_mat[0,1] = el_stress[3]
        stress_mat[1,0] = el_stress[3]
        stress_mat[1,1] = el_stress[2]
        new_stress_mat = np.matmul(np.matmul(rot_mat, stress_mat),
         ↪  np.transpose(rot_mat))
        new_stress_vec[i,:] = [nodes[0,0], new_stress_mat[0,0],
         ↪  new_stress_mat[1,1], new_stress_mat[0,1]]
    return new_stress_vec
# E_1 = 100.001
# nu_1 = 0.3501
#
# E_2 = 200.001
# nu_2 = 0.3901
# file_name = "micro_struc_hm_current_test_new_load"
#
# grid_data = read_grid_data(file_name)
# e_node_tri = tri_element_nodes(file_name)
# e_area_tri = tri_element_area(grid_data, e_node_tri)
# e_node_quad = quad_element_nodes(file_name)
# e_area_quad = quad_element_area(grid_data, e_node_quad)

#linear_displacement(file_name, data, "y")
#reac_data = read_reac_force(file_name)
#grid_bound = np.zeros((len(reac_data), 3))
#i = 0
# determine grid nodes
#for node in grid_data:
#    if node[0] in reac_data[:, 0]:
#        grid_bound[i] = node
#        i += 1
#(x_reac, y_reac, x_tick, y_tick) = normal_force(reac_data, grid_bound)
#E_22 = [x_reac, y_reac, 0]
```

```
#output_files = os.system(f'C:\Program^
↪  Files\MSC.Software\MSC_Nastran\\2021.3\\bin\\nastranw.exe
↪  nastran_input\\{file_name}.bdf')

#D_22 = calculate_compliance(file_name)

#change_material(phase_1, phase_2, 1)
#change_load()
```

## C.2.4 Main script

Running this script creates a complete dataset.

```
import numpy as np
import time
import Phase
import nastran_file_generator as nas_gen
import phase_comp_generator as ph_gen
import re


def write_dataset(arr):
    """
    Writes a dataset file. test_nr, phase_1, Phase_2, micro
    :param arr: micro obj. array.
    """
    data = []
    for i, micro in enumerate(arr):
        data.append(f"{str(micro.test_nr)}
        ↪  {str(micro.phase_1)}{str(micro.phase_2)}{str(micro)}\n")
    with open(f'data/data_set_5.txt', mode='w') as file:
        file.writelines(data)

def string_func(D):
    string = [" "]
    for i in range(3):
        for j in range(3):
            string.append(str(D[i, j]))
            string.append(" ")
    return "".join(string)


def write_phases(D_1,D_2):
    data = []
    for i, D1 in enumerate(D_1):
        data.append(f"{string_func(D1)}{string_func(D_2[i])}\n")
    with open(f'data/phase_data.txt', mode='w') as file:
        file.writelines(data)

def extract_phases(phase_file):
    with open(f'data/{phase_file}.txt', 'r') as file:
        # read a list of lines into data
        data = file.readlines()
    out_data = []
    for i, line in enumerate(data):
        s_out = re.findall("[-+]?[.]?[\d]+(?:,\d\d\d)*[\.]?\d*(?:[eE][-+]?\
        ↪  d+)?",
        ↪  line)
        out_data.append([float(val) for val in s_out])
    return out_data


start_time = time.time()
orig_file_name = "morph_5_x"
# generate material space with latin hypercube
# (D_1, D_2) = ph_gen.phaseinator()
# write_phases(D_1,D_2)
phase_data = extract_phases("phase_data")
# calculate pre-proc data from bdf original file
grid_data = nas_gen.read_grid_data(orig_file_name)  # node position data
e_node_tri = nas_gen.tri_element_nodes(orig_file_name)
e_area_tri = nas_gen.tri_element_area(grid_data, e_node_tri)  # area for
↪  each triangular element
e_node_quad = nas_gen.quad_element_nodes(orig_file_name)
e_area_quad = nas_gen.quad_element_area(grid_data, e_node_quad)  # area for
↪  each quadrilateral element
e_area = np.append(e_area_quad, e_area_tri, 0)
nel = len(e_area)
e_node_2 = np.append(e_node_quad[:, 0:3], e_node_tri[:, 0:3], 0)
# from a material sample (2 phases) generate a Micro object

# ---- only required once----- #
#nas_gen.linear_displacement("morph_2_x", grid_data, "x", 10200.0)
#nas_gen.linear_displacement("morph_2_y", grid_data, "y", 10200.0)
#nas_gen.linear_shear_displacement("morph_2_xy", grid_data,  10200.0)
```

```
test_nr = 0
tests = 100
micro_arr = []
for i in range(tests):
    k = 0
    D1 = np.zeros((3,3))
    D2 = np.zeros((3,3))
    for j in range(3):
        for m in range(3):
            D1[j,m] = phase_data[i][k]
            D2[j,m] = phase_data[i][k+9]
            k += 1
    phase_1 = Phase.PhaseMat(D1)
    phase_2 = Phase.PhaseMat(D2)
    micro_arr.append(Phase.Micro(phase_1, phase_2, test_nr, grid_data,
    ↪  e_area))
    micro_arr[i].elast_bounds()
    test_nr += 1
    time.sleep(10)
time.sleep(25)

for i, micro in enumerate(micro_arr):
    micro.move_f06_files()

for micro in micro_arr:
    micro.calc_stresses(e_node_2)
    micro.check_elast_bounds()
    micro.check_symm()
    if micro.bound_check:
        print("Passed Voigt-Reuss check")
    else:
        print("Failed")
    if micro.sym_check:
        print("Passed symmetry check")
    else:
        print("Failed")


# # Micro object changes material data and generates new bdf files
# # Micro object runs bdf files in Nastran
# # Micro object interperates f06 file, gets (area, and element stresses)
write_dataset(micro_arr)
```

```
stop_time = time.time()
run_time = stop_time - start_time  # seconds
print(run_time)

print(1)
```

## C.3 Deep material network

### C.3.1 DMN class

This module includes 2 classes a branch which represents a parent node with 2 child nodes. The branch handles the node specific calculations and its stored data. The second class is a network class, it incorporates the network spanning data and calculations.

```
import numpy as np


class Branch:
    """
    represents a branch in the deep-material network.
    The branch is the connection between two child nodes and one parent
↪   node
    """
    def __init__(self, child_1, child_2, theta, inp, w):
        self.bias = 0
        self.bias_eta = 0.004  # stnd: 0.001
        self.bias_del = []
        self.comp_correct = np.array([[1, 1, 1], [0, 1, 1], [0, 0, 1]])
        self.inp = inp
        self.alpha = np.zeros((3, 3))
        self.dC_dW = []
        self.dC_dTheta = []
        self.c_theta = 0.01  # RMSprop parameter
        self.c_z = 0.01  # RMSprop parameter
        self.dC_dW_prev = 0
        self.dC_dtheta_prev = 0
        self.learn_w = 0.01
        self.learn_theta = 0.02
        self.eta_z = 0.3  # stnd: 0.1
        self.eta_theta = 0.3  # stnd:0.1 learning rates
        self.ch_1 = child_1
        self.ch_2 = child_2
        self.theta = theta
        self.D_r = np.zeros((3, 3))  # output compliance before rotation
        self.D_bar = np.zeros((3, 3))  # output compliance after rotation
        self.delta = np.zeros((3, 3))
        self.w = w
        if not inp:
            self.f_1 = self.ch_1.w/(self.ch_1.w + self.ch_2.w)
            self.f_2 = 1 - self.f_1
        else:
            self.f_1 = 1
            self.f_2 = 0

    def homogen(self):
        if (self.ch_1.w + self.ch_2.w) == 0:
            # should never occur
            self.f_1 = 0
        else:
            self.f_1 = self.ch_1.w / (self.ch_1.w + self.ch_2.w)
        self.D_1 = self.ch_1.D_bar*(1 + self.ch_1.bias)
        self.D_2 = self.ch_2.D_bar*(1 + self.ch_2.bias)

        self.f_2 = 1 - self.f_1
        gamma = self.f_1*self.D_2[0, 0] + self.f_2*self.D_1[0, 0]
        self.D_r[0, 0] = (self.D_1[0, 0]*self.D_2[0, 0])/gamma
        self.D_r[0, 1] = (self.f_1*self.D_1[0, 1]*self.D_2[0, 0] +
         ↪  self.f_2*self.D_1[0, 0]*self.D_2[0, 1])/gamma
        self.D_r[1, 0] = self.D_r[0, 1]
        self.D_r[0, 2] = 1/gamma*(self.f_1*self.D_1[0, 2]*self.D_2[0, 0] +
         ↪  self.f_2 * self.D_1[0, 0]*self.D_2[0, 2])
        self.D_r[2, 0] = self.D_r[0, 2]
        self.D_r[1, 1] = self.f_1*self.D_1[1, 1] + self.f_2*self.D_2[1, 1]
         ↪  - self.f_1*self.f_2*((self.D_1[0,1] - self.D_2[0,1])**2)/gamma
        self.D_r[1, 2] = self.f_1*self.D_1[1, 2] + self.f_2*self.D_2[1, 2]
         ↪  - self.f_1*self.f_2*(self.D_1[0,2] -
         ↪  self.D_2[0,2])*(self.D_1[0,1] - self.D_2[0,1])/gamma
        self.D_r[2, 1] = self.D_r[1, 2]
        self.D_r[2, 2] = self.f_1*self.D_1[2, 2] + self.f_2*self.D_2[2, 2]
         ↪  - self.f_1*self.f_2*((self.D_1[0,2] - self.D_2[0,2])**2)/gamma

        self.gamma = gamma

    def rot_mat(self, theta):
        R = np.zeros((3, 3))
        R[0, :] = [np.cos(theta)**2, np.sin(theta)**2,
         ↪  np.sqrt(2)*np.sin(theta)*np.cos(theta)]
        R[1, :] = [np.sin(theta) ** 2, np.cos(theta) ** 2, -np.sqrt(2) *
         ↪  np.sin(theta) * np.cos(theta)]
        R[2, :] = [-np.sqrt(2) * np.sin(theta) * np.cos(theta), np.sqrt(2)
         ↪  * np.sin(theta) * np.cos(theta), np.cos(theta)**2 -
         ↪  np.sin(theta)**2]
        return R

    def rot_mat_prime(self, theta):
        R = np.zeros((3, 3))
        R[0, :] = [-np.sin(2*theta), np.sin(2*theta),
         ↪  np.sqrt(2)*np.cos(2*theta)]
        R[1, :] = [np.sin(2*theta), -np.sin(2*theta),
         ↪  -np.sqrt(2)*np.cos(2*theta)]
        R[2, :] = [-np.sqrt(2) * np.cos(theta*2), np.sqrt(2) *
         ↪  np.cos(theta*2), -2*np.sin(2*theta)]
        return R

    def rotate_comp(self):
        D_bar = np.matmul(np.matmul(self.rot_mat(-self.theta), self.D_r),
         ↪  self.rot_mat(self.theta))
        self.D_bar = D_bar

    def gradients(self):
        # derivative of D comps with respect to D_r
        D_d_Dr = np.zeros((3, 3, 3, 3))
        R_1 = self.rot_mat(-self.theta)
        R_2 = self.rot_mat(self.theta)
        for i in range(3):
            for j in range(3):
                for k in range(3):
                    for l in range(3):
                        D_d_Dr[i,j,k,l] = R_1[i,k]*R_2[l,j]
        self.D_d_Dr = D_d_Dr

        # derivatives of D_r comps with respect to first child D1

        def Dr_d_D_func(D_1,D_2,f_1,f_2):
            Dr_d_D = np.zeros((3, 3, 3, 3))
            temp = np.zeros((3,))
            temp[:] = [f_1 * D_2[0, 0] ** 2 / self.gamma, f_2 *
             ↪  (-self.D_r[0, 1] + D_2[0, 1]),
                       f_2 * (-self.D_r[0, 2] + D_2[0, 2])]
            Dr_d_D[:, 0, 0, 0] = 1 / self.gamma * temp
            temp[:] = [f_2 * (-self.D_r[0, 1] + D_2[0, 1]),
                       f_1 * f_2 ** 2 * (D_1[0, 1] - D_2[0, 1]) ** 2 /
                        ↪  self.gamma,
                       f_1 * f_2 ** 2 * (D_1[0, 2] - D_2[0, 2]) * (
                               D_1[0, 1] - D_2[0, 1]) / self.gamma]
            Dr_d_D[:, 1, 0, 0] = 1 / self.gamma * temp
            temp[:] = [f_2 * (-self.D_r[0, 2] + D_2[0, 2]),
                       f_1 * (f_2 ** 2) * (D_1[0, 2] - D_2[0, 2]) * (
                               D_1[0, 1] - D_2[0, 1]) / self.gamma,
                       f_1 * (f_2 ** 2) * ((D_1[0, 2] - D_2[0, 2]) ** 2) /
                        ↪  self.gamma]
            Dr_d_D[:, 2, 0, 0] = 1 / self.gamma * temp

            Dr_d_D[:, 0, 0, 1] = [0, f_1 * D_2[0, 0] / self.gamma, 0]
            Dr_d_D[:, 1, 0, 1] = [f_1 * D_2[0, 0] / self.gamma,
                                   -2 * f_1 * f_2 * (D_1[0, 1] - D_2[0, 1])
                                    ↪  / self.gamma,
                                   -f_1 * f_2 * (D_1[0, 2] - D_2[0, 2]) /
                                    ↪  self.gamma]
            Dr_d_D[:, 2, 0, 1] = [0, -f_1 * f_2 * (D_1[0, 2] - D_2[0, 2]) /
             ↪  self.gamma, 0]

            Dr_d_D[:, :, 1, 0] = Dr_d_D[:, :, 0, 1]

            Dr_d_D[:, 0, 0, 2] = [0, 0, f_1 * D_2[0, 0] / self.gamma]
            Dr_d_D[:, 1, 0, 2] = [0, 0, -f_1 * f_2 * (D_1[0, 1] - D_2[0,
             ↪  1]) / self.gamma]
            Dr_d_D[:, 2, 0, 2] = [f_1 * D_2[0, 0] / self.gamma,
                                   -f_1 * f_2 * (D_1[0, 1] - D_2[0, 1]) /
                                    ↪  self.gamma,
                                   -2 * f_1 * f_2 * (D_1[0, 2] - D_2[0, 2])
                                    ↪  / self.gamma]

            Dr_d_D[:, :, 2, 0] = Dr_d_D[:, :, 0, 2]

            Dr_d_D[:, :, 1, 1] = [[0, 0, 0], [0, f_1, 0], [0, 0, 0]]
            Dr_d_D[:, :, 1, 2] = [[0, 0, 0], [0, 0, f_1], [0, f_1, 0]]
            Dr_d_D[:, :, 2, 1] = Dr_d_D[:, :, 1, 2]
            Dr_d_D[:, :, 2, 2] = [[0, 0, 0], [0, 0, 0], [0, 0, f_1]]
            return Dr_d_D
```

```
        dr_d_d1 = Dr_d_D_func(self.D_1, self.D_2, self.f_1, self.f_2)
        dr_d_d2 = Dr_d_D_func(self.D_2, self.D_1, self.f_2, self.f_1)

        def func_dr_df(f_1, f_2, D_1, D_2):
            Dr_d_f1 = np.zeros((3, 3))
            Dr_d_f1[0, 0] = (D_1[0, 0] - D_2[0, 0]) * (D_1[0, 0] * D_2[0,
             ↪  0]) / self.gamma ** 2
            Dr_d_f1[0, 1] = (D_1[0, 0] * D_2[0, 0] * (D_1[0, 1] - D_2[0,
             ↪  1])) / self.gamma ** 2
            Dr_d_f1[1, 0] = Dr_d_f1[0, 1]
            Dr_d_f1[0, 2] = (D_1[0, 0] * D_2[0, 0] * (D_1[0, 2] - D_2[0,
             ↪  2])) / self.gamma ** 2
            Dr_d_f1[2, 0] = Dr_d_f1[0, 2]
            Dr_d_f1[1, 1] = D_1[1, 1] - D_2[1, 1] - ((D_1[0, 1] - D_2[0,
             ↪  1]) ** 2 * (
                   D_1[0, 0] * (f_1 ** 2 - 2 * f_1 + 1) - D_2[
                0, 0] * f_1 ** 2)) / self.gamma ** 2
            Dr_d_f1[1, 2] = D_1[1, 2] - D_2[1, 2] - (
                    (D_1[0, 1] - D_2[0, 1]) * (D_1[0, 2] - D_2[0, 2]) * (
                    D_1[0, 0] * (f_1 ** 2 - 2 * f_1 + 1) - D_2[
                0, 0] * f_1 ** 2)) / self.gamma ** 2
            Dr_d_f1[2, 1] = Dr_d_f1[1, 2]
            Dr_d_f1[2, 2] = D_1[2, 2] - D_2[2, 2] - ((D_1[0, 2] - D_2[0,
             ↪  2]) ** 2 * (
                    D_1[0, 0] * (f_1 ** 2 - 2 * f_1 + 1) - D_2[
                0, 0] * f_1 ** 2)) / self.gamma ** 2
            return Dr_d_f1
        dr_d_f1 = func_dr_df(self.f_1, self.f_2, self.D_1, self.D_2)
        dr_d_f2 = func_dr_df(self.f_2, self.f_1, self.D_2, self.D_1)

        self.Dr_d_f1 = dr_d_f1
        self.Dr_d_f2 = dr_d_f2
        self.D_d_theta =
         ↪  np.matmul(np.matmul(-self.rot_mat_prime(-self.theta),
         ↪  self.D_r), self.rot_mat(self.theta)) +
         ↪  np.matmul(np.matmul(self.rot_mat(-self.theta), self.D_r),
         ↪  self.rot_mat_prime(self.theta))
        self.Dr_d_D1 = dr_d_d1
        self.Dr_d_D2 = dr_d_d2
        self.D_d_Dr = D_d_Dr

    def theta_grad(self):
        self.D_d_theta =
         ↪  -np.matmul(np.matmul(self.rot_mat_prime(-self.theta),
         ↪  self.D_r), self.rot_mat(self.theta)) +
         ↪  np.matmul(np.matmul(self.rot_mat(-self.theta), self.D_r),
         ↪  self.rot_mat_prime(self.theta))
        D_d_Dr = np.zeros((3, 3, 3, 3))
        R_1 = self.rot_mat(-self.theta)
        R_2 = self.rot_mat(self.theta)
        for i in range(3):
            for j in range(3):
                for k in range(3):
                    for l in range(3):
                        D_d_Dr[i, j, k, l] = R_1[i, k] * R_2[l, j]
        self.D_d_Dr = D_d_Dr
        self.D_d_Dr = D_d_Dr

    def update_theta(self, N):
        # gamma = 1
        # self.c_theta = self.c_theta * gamma + (1 - gamma) *
         ↪  ((np.mean(self.dC_dTheta)) ** 2)
        # learn = self.eta_theta / (np.sqrt(self.c_theta) + 1E-6)
        # self.learn_theta = np.min([learn, 0.4])
        eta = self.eta_theta / N
        dC_dtheta = eta * np.mean(self.dC_dTheta)
        self.theta -= dC_dtheta
        if self.theta > np.pi/2:
            self.theta -= np.pi
        elif self.theta < -np.pi/2:
            self.theta += np.pi
        self.dC_dTheta = []

    def bias_update(self):
        self.bias -= self.bias_eta*np.mean(self.bias_del)
        if self.bias <= -0.99:
            self.bias = -0.99
        self.bias_del = []

    def update_z(self, N):
        """
        Update weight for branch node
        N : Layer index
        :return:
        """
        #gamma = 0.975
        #self.c_z = self.c_z*gamma + (1-gamma)*(np.mean(self.dC_dW))**2
        #learn = self.eta_z/(np.sqrt(self.c_z) + 1E-6)
        #self.learn_w = np.min([learn, 1.5])
        dC_dW = np.mean(self.dC_dW)*self.eta_z
        self.w -= dC_dW
        if self.w <= 0:
            self.w = 0
        self.dC_dW = []

    def calc_delta(self, child_1):
        delta_new = np.zeros((3, 3))
        if child_1:  # if current node is child 1 of the parent node
            Dr_d_D = self.Dr_d_D1
        else:
            Dr_d_D = self.Dr_d_D2
        delta_new[0, 0] = np.sum(self.alpha * Dr_d_D[:, :, 0, 0])
        delta_new[0, 1] = np.sum(self.alpha * Dr_d_D[:, :, 0, 1])
        delta_new[0, 2] = np.sum(self.alpha * Dr_d_D[:, :, 0, 2])
        delta_new[1, 1] = np.sum(self.alpha * Dr_d_D[:, :, 1, 1])
        delta_new[1, 2] = np.sum(self.alpha * Dr_d_D[:, :, 1, 2])
        delta_new[2, 2] = np.sum(self.alpha * Dr_d_D[:, :, 2, 2])
        return delta_new


class Network:
    """
    Represents the network structure.
    Contains N layers
    """
    def __init__(self, N, D_1, D_2, D_correct):
        self.C = []
        self.C0 = []
        self.error = []
        self.N = N  # network depth
        self.D_1 = D_1  # phase 1 compliance (all samples)
        self.D_2 = D_2  # phase 2 compliance (all samples)
        self.D_correct = D_correct  # correct compliance after
        ↪  homogenization
        self.input_layer = []
        rng = np.random.default_rng()
        self.comp_correct = np.array([[1, 1, 1], [0, 1, 1], [0, 0, 1]])
        #zs = np.linspace(0.2, 0.8, 2**(N-1))
        for j in range(2 ** (N - 1)):
            samples = rng.uniform(size=(2, 1))
            if np.mod(j, 2) == 0:
                # Phase 1 input nodes
                in_node = Branch(D_1, D_1, np.pi*(samples[0][0] - 1/2),
                 ↪  True, 0.49 + samples[1][0]*0.02)
                in_node.D_r = D_1
            else:
                # Phase 2 input nodes
                in_node = Branch(D_2, D_2, np.pi*(samples[0][0] - 1/2),
                 ↪  True, 0.49 + samples[1][0]*0.02)
                in_node.D_r = D_2
            self.input_layer.append(in_node)

        self.layers = []
        self.ws = [node.w for node in self.input_layer]
        for i in range(0, N-1):
            self.layers.append(self.fill_layer(i))

    def fill_layer(self, i):
        if i == 0:
            prev_layer = self.input_layer
        else:
            prev_layer = self.layers[i-1]
        new_layer = []
        rng = np.random.default_rng()
        for j in range(int(len(prev_layer)/2)):
            samples = rng.uniform(size=(2, 1))
            new_layer.append(Branch(prev_layer[j*2], prev_layer[j*2 +
             ↪  1],np.pi*(samples[0][0] - 1/2), False, 0.49 +
             ↪  samples[1][0]*0.02))
        return new_layer

    def get_comp(self):
        D_bar = self.layers[-1][0].D_bar
        return (1 + self.layers[-1][0].bias)*D_bar

    def forward_pass(self):
        for node in self.input_layer:
            node.rotate_comp()
        for layer in self.layers:
            for node in layer:
                node.homogen()
                node.rotate_comp()
                node.gradients()

    def update_phases(self, D_1, D_2, DC):
        self.D_correct = DC
        for j, node in enumerate(self.input_layer):
            if np.mod(j, 2) == 0:
                # Phase 1 input nodes
                node.D_r = D_1
            else:
                # Phase 2 input nodes
                node.D_r = D_2

    def update_learn_rate(self, new_eta_t, new_eta_z):
        for layer in self.layers:
            for node in layer:
                node.eta_theta = new_eta_t
                node.eta_z = new_eta_z

    def calc_cost(self):
        D_bar = self.get_comp()
        self.del_C = self.comp_correct*(D_bar - self.D_correct)/(np.linalg.↲
            ↪  norm(self.comp_correct*self.D_correct, 'fro'))**2  # cost
            ↪  gradient
        #self.del_C = self.comp_correct*(D_bar -
            ↪  self.D_correct)/((np.linalg.norm(self.comp_correct*(D_bar -
            ↪  self.D_correct),
            ↪  'fro'))*(np.linalg.norm(self.comp_correct*self.D_correct,
            ↪  'fro')))
        self.C.append(((np.linalg.norm(self.comp_correct*(D_bar -
            ↪  self.D_correct),
            ↪  'fro'))**2)/((np.linalg.norm(self.comp_correct*self.D_correct,
            ↪  'fro'))**2))
        self.error.append((np.linalg.norm(self.comp_correct*(D_bar -
            ↪  self.D_correct),
            ↪  'fro'))/(np.linalg.norm(self.comp_correct*self.D_correct,
            ↪  'fro')))

    def backwards_prop(self):
        for i, layer in enumerate(reversed(self.layers)):
            m = 0
            for k, node in enumerate(layer):
                if i == 0:
```

```
            # output layer
            delta_new = self.del_C
            node.bias_del.append(np.sum(self.del_C*node.D_bar))
            dC_dW = 0
            node.dC_dW.append(dC_dW)
        else:
            parent_node = prev_layer[m]
            if np.mod(k, 2) == 0:
                delta_new = parent_node.calc_delta(True)
                Dr_df = parent_node.Dr_d_f1
            else:
                delta_new = parent_node.calc_delta(False)
                Dr_df = parent_node.Dr_d_f2
                m += 1
        node.delta = delta_new
        beta = delta_new + delta_new*node.bias
        node.beta = beta
        node.bias_del.append(np.sum(node.delta * node.D_bar))
        alpha = np.zeros((3, 3))
        alpha[0, 0] = np.sum(beta * node.D_d_Dr[:, :, 0, 0])
        alpha[0, 1] = np.sum(beta * node.D_d_Dr[:, :, 0, 1])
        alpha[0, 2] = np.sum(beta * node.D_d_Dr[:, :, 0, 2])
        alpha[1, 1] = np.sum(beta * node.D_d_Dr[:, :, 1, 1])
        alpha[1, 2] = np.sum(beta * node.D_d_Dr[:, :, 1, 2])
        alpha[2, 2] = np.sum(beta * node.D_d_Dr[:, :, 2, 2])
        node.alpha = alpha
        if i != 0:
            if (parent_node.ch_1.w + parent_node.ch_2.w) == 0:
                node.dC_dW.append(0)
            else:
                dC_dW = node.w * np.sum(parent_node.alpha*Dr_df)/(p⌋
                 ↪  arent_node.ch_1.w +
                 ↪  parent_node.ch_2.w)
                node.dC_dW.append(dC_dW)

        dC_d_theta = np.sum(beta*node.D_d_theta)
        node.dC_dTheta.append(dC_d_theta)
    prev_layer = layer
for j, node in enumerate(self.input_layer):
    if node.w <= 0:
        node.dC_dW = 0
        node.dC_dTheta.append(0)
    else:
```

```
                parent_ind = int((j - np.mod(j, 2)) / 2)
                parent_node = self.layers[0][parent_ind]
                if np.mod(j, 2) == 0:
                    delta_new = parent_node.calc_delta(True)
                    Dr_df = parent_node.Dr_d_f1
                else:
                    delta_new = parent_node.calc_delta(False)
                    Dr_df = parent_node.Dr_d_f2
                beta = delta_new + delta_new*node.bias
                node.theta_grad()
                dC_dW = node.w * np.sum(parent_node.alpha*Dr_df) /
                 ↪  (parent_node.ch_1.w + parent_node.ch_2.w)
                node.dC_dW.append(dC_dW)
                dC_d_theta = np.sum(beta * node.D_d_theta)
                node.dC_dTheta.append(dC_d_theta)
                node.delta = delta_new
                node.bias_del.append(np.sum(node.delta * node.D_bar))

    def learn_step(self):
        for i, node in enumerate(self.input_layer):
            #if np.mod(i, 2) == 0:
            node.update_z(1)
            node.update_theta(1)
            node.bias_update()

        for j, layer in enumerate(self.layers):
            for i, node in enumerate(layer):
                #if np.mod(i, 2) == 0:
                node.update_z(1)
                node.update_theta(1)
                node.bias_update()
        self.ws = [node.w for node in self.input_layer]


def component_vec(matrix):
    vec = np.zeros((1, 6))
    k = 0
    for i in range(3):
        for j in range(3):
            if i == 1 and j == 0:
                continue
            elif i == 2 and j == 0:
                continue
            elif i == 2 and j == 1:
                continue
            vec[0, k] = matrix[i, j]
            k += 1
    return vec
```

### C.3.2 Main script

The main script trains and validates a DMN. The main script includes functions for storing a DMN in a text file, creating a DMN from text file, training a DMN on a dataset, validating on a dataset, and some demonstration functions. The demonstration functions showcase the possible use of a DMN, and some visual representations of the datasets.

```
import numpy as np
import re
import DMN_2
import matplotlib.pyplot as plt
import time
import os


def run_scatter_plot():
    data = read_dataset("data_set_1")
    E_rve_1 = []
    E_rve_2 = []
    E_voigt = []
    E_reuss = []
    E_voigt_2 = []
    E_reuss_2 = []
    for i, line in enumerate(data):
        D_1 = line[1:10]
        D_2 = line[10:19]
        D_rve = line[19:]
        k = 0
        D1 = np.zeros((3,3))
        D2 = np.zeros((3, 3))
        for i in range(3):
            for j in range(3):
                D1[i, j] = D_1[k]
                D2[i, j] = D_2[k]
                k += 1
        C_reuss, C_voigt, D_reuss, D_voigt = elast_bounds(D1,D2)
        E_rve_1.append(1 / (D_rve[0]))
        E_rve_2.append(1 / (D_rve[4]))
        E_voigt.append(C_voigt[0, 0])
        E_reuss.append(C_reuss[0, 0])
        E_voigt_2.append(C_voigt[0, 1])
        E_reuss_2.append(C_reuss[0, 1])

    fig, axs = plt.subplots(2, 2, constrained_layout=True)
    E = "E_1"
    fig.suptitle(f"Morph. 1")
    axs[0,0].scatter(E_voigt, E_rve_1)
    axs[0, 1].scatter(E_reuss, E_rve_1)
    axs[1, 0].scatter(E_voigt_2, E_rve_2)
    axs[1, 1].scatter(E_reuss_2, E_rve_2)
    axs[0, 0].set_xlabel(f'Voigt {E}', fontweight='bold')
    axs[0, 1].set_xlabel(f'Reuss {E}', fontweight='bold')
    axs[1, 0].set_xlabel('Voigt E_2', fontweight='bold')
    axs[1, 1].set_xlabel('Reuss E_2', fontweight='bold')

    axs[0, 0].set_ylabel(f'RVE {E}', fontweight='bold')
    axs[0, 1].set_ylabel(f'RVE {E}', fontweight='bold')
    axs[1, 0].set_ylabel(f'RVE E_2', fontweight='bold')
    axs[1, 1].set_ylabel(f'RVE E_2', fontweight='bold')
    plt.savefig(f"test_scatter_1.svg")
    plt.show()


def elast_bounds(D_1, D_2):
    """
    calculate lower Reuss value and upper Voigt value for the
↪   representative elasticity parameters
    """
    vol_frac = 0.5
    D_voigt = np.zeros((3, 3))
    D_voigt[0, 0] = 1 / (vol_frac / D_1[0, 0] + (1 - vol_frac) / D_2[0, 0])
    D_voigt[0, 1] = 1 / (vol_frac / D_1[0, 1] + (1 - vol_frac) / D_2[0, 1])
    D_voigt[1, 0] = D_voigt[0, 1]
    D_voigt[1, 1] = 1 / (vol_frac / D_1[1, 1] + (1 - vol_frac) / D_2[1, 1])
    D_voigt[2, 2] = 1 / (vol_frac / D_1[2, 2] + (1 - vol_frac) / D_2[2, 2])

    C_voigt = np.zeros((1, 4))
    C_voigt[0, 0] = 1 / D_voigt[0, 0]
    C_voigt[0, 1] = 1 / D_voigt[1, 1]
    C_voigt[0, 3] = 1 / (2 * D_voigt[2, 2])
    C_voigt[0, 2] = -C_voigt[0, 0] * D_voigt[1, 0]

    D_reuss = np.zeros((3, 3))
    D_reuss[0, 0] = vol_frac * D_1[0, 0] + (1 - vol_frac) * D_2[0, 0]
    D_reuss[0, 1] = vol_frac * D_1[0, 1] + (1 - vol_frac) * D_2[0, 1]
    D_reuss[1, 0] = D_reuss[0, 1]
    D_reuss[1, 1] = vol_frac * D_1[1, 1] + (1 - vol_frac) * D_2[1, 1]
    D_reuss[2, 2] = vol_frac * D_1[2, 2] + (1 - vol_frac) * D_2[2, 2]

    C_reuss = np.zeros((1, 4))
    C_reuss[0, 0] = 1 / D_reuss[0, 0]
    C_reuss[0, 1] = 1 / D_reuss[1, 1]
    C_reuss[0, 3] = 1 / (2 * D_reuss[2, 2])
    C_reuss[0, 2] = -C_reuss[0, 0] * D_reuss[1, 0]
    return C_reuss, C_voigt, D_reuss, D_voigt


def file_reader(filename):
    f = open(filename, 'r')
    Lines = f.readlines()
    line = []
    for Line in Lines:
        nums = Line.split("#")
        points = []
        for num in nums:
            if num == "\n":
                line.append([points])
                break
            point = num.replace("[", '')
            point = point.replace("]",'')
            point = point.split(",")
            points.append([float(point[0]), float(point[1])])
    f.close()
    return line


def write_dmn(dmn, ind):
    data = []
    for node in dmn.input_layer:
        data.append(f"{str(node.theta)} ")
    data.append("\n")
    for layer in dmn.layers:
        for node in layer:
```

```python
            data.append(f"{str(node.theta)} ")
        data.append("\n")
    for node in dmn.input_layer:
        data.append(f"{str(node.w)} ")
    data.append("\n")
    for layer in dmn.layers:
        for node in layer:
            data.append(f"{str(node.w)} ")
        data.append("\n")
    for node in dmn.input_layer:
        data.append(f"{str(node.bias)} ")
    data.append("\n")
    for layer in dmn.layers:
        for node in layer:
            data.append(f"{str(node.bias)} ")
        data.append("\n")
    with open(f'data/DMN_{ind}.txt', 'w') as file:
        file.writelines(data)


def write_data(epoc_cost, zs, ind):
    data = []
    for i,ep_cost in enumerate(epoc_cost):
        data.append(f"{ep_cost} z:{zs[i,:]}\n")
    with open(f'data/Outdata_{ind}.txt', 'w') as file:
        file.writelines(data)


def read_dataset(file_name):
    with open(f'data/{file_name}.txt', 'r') as file:
        in_data = file.readlines()
    data = np.ndarray((len(in_data), 28))
    for i, s in enumerate(in_data):
        s_out = re.findall("[-+]?[.]?[\d]+(?:,\d\d\d)*[\.]?\d*(?:[eE][-+]?\⌟
        ↪  d+)?",
        ↪  s)
        nums = [float(val) for val in s_out]
        data[i] = nums
    return data


def make_dataset_sym(data):
    sym_data = []
    for i in range(200):
        data_line = data[i,:]
        d_corr = data[i, 19:28]
        d_21 = (d_corr[1] + d_corr[3])/2
        d_31 = (d_corr[2] + d_corr[6])/2
        d_32 = (d_corr[5] + d_corr[7])/2
        d_corr[1] = d_21
        d_corr[3] = d_21
        d_corr[2] = d_31
        d_corr[6] = d_31
        d_corr[5] = d_32
        d_corr[7] = d_32
        data_line[19:28] = d_corr
        data_string = ' '.join(str(x) for x in data_line)
        sym_data.append(f"{data_string}\n")
    with open(f'data/Symdata2.txt', 'w') as file:
        file.writelines(sym_data)


def extract_D_mat(data, ind):
    D_1 = data[ind, 1:10]
    D_2 = data[ind, 10:19]
    D_correct = data[ind, 19:28]
    D1 = np.zeros((3, 3))
    D2 = np.zeros((3, 3))
    DC = np.zeros((3, 3))
    k = 0
    for i in range(3):
        for j in range(3):
            D1[i, j] = D_1[k]
            D2[i, j] = D_2[k]
            DC[i, j] = D_correct[k]
            k += 1

    return D1, D2, DC


def create_dmn_from_save(dmn_file):
    """
    Create a nn object from saved weights
    :param dmn_file: data file with weights from text file
    :return: nn object
    """
    with open(f'data/{dmn_file}.txt', 'r') as file:
        in_data = file.readlines()
    N = int(len(in_data)/3)
    D1 = np.zeros((3, 3))
    D2 = np.zeros((3, 3))
    DC = np.zeros((3, 3))
    nn = DMN_2.Network(N, D1, D2, DC)
    theta_in = in_data[0].split(" ")
    for i, node in enumerate(nn.input_layer):
        node.theta = float(theta_in[i])
    for i, layer in enumerate(nn.layers):
        thetas = in_data[i+1].split(" ")
        for j, node in enumerate(layer):
            node.theta = float(thetas[j])
        ind = i
    zs = in_data[ind + 2].split(" ")
    for i, node in enumerate(nn.input_layer):
        node.w = float(zs[i])
    for j, layer in enumerate(nn.layers):
        ws = in_data[ind+3+j].split(" ")
        for k, node in enumerate(layer):
            node.w = float(ws[k])
    bias = in_data[ind + 4 + j].split(" ")
    for i, node in enumerate(nn.input_layer):
        node.bias = float(bias[i])
    for m, layer in enumerate(nn.layers):
        bias = in_data[ind + 5 + j + m].split(" ")
        for k, node in enumerate(layer):
            node.bias = float(bias[k])
    return nn


def run_train_sample(epoch, M, ind, N_s, data_file, nn=False,
↪   inter_plot=True, N=False, update_lam=True):
    """
    Train a DMN network for a set of epochs
    :param epoch: amount of epochs, one training session over dataset
    :param M: Minibatch size. Amount of samples in a minibatch
    :param nn: A Network object from DMN.If False, create new object
    :param inter_plot: interactive plotting on or off.
    :return: Trained network object. Epoch averaged cost. z activations
↪   for each training step.
    """
```

```
if not nn:
    D1 = np.zeros((3, 3))
    D2 = np.zeros((3, 3))
    DC = np.zeros((3, 3))
    nn = DMN_2.Network(N, D1, D2, DC)
    print("new network")
else:
    N = nn.N
    nn.ws = [node.w for node in nn.input_layer]
    print("old network")
cost = []
error = []
w_list = []
th_list = []
bias_list = []
th_list.append(np.zeros((int(epoch * N_s / M), 2 ** (N - 1))))
w_list.append(np.zeros((int(epoch*N_s/M), 2 ** (N - 1))))
bias_list.append(np.zeros((int(epoch*N_s/M), 2 ** (N - 1))))
for i, layer in enumerate(nn.layers):
    w_list.append(np.zeros((int(epoch*N_s/M), 2 ** (N - 2 - i))))
    th_list.append(np.zeros((int(epoch*N_s/M), 2 ** (N - 2 - i))))
    bias_list.append(np.zeros((int(epoch*N_s/M), 2 ** (N - 2 - i))))

m = 0
epoch_cost = np.zeros((epoch, 1))
epoch_error = np.zeros((epoch, 1))
epoch_ws = np.zeros((epoch, 2**(N-1)))
if inter_plot:
    plt.ion()
    fig, axs = plt.subplots(nn.N, 3, constrained_layout=True)
    fig.suptitle(fr"dataset: {data_file}")
    fig.set_size_inches(14, 9, forward=True)
    axs[nn.N - 1, 0].set_yscale("log")
    axs[nn.N-1, 1].set_yscale("log")
start_time = time.time()
for i in range(epoch):
    if update_lam:
        if i == 200:
            nn.lam = nn.lam*3/4
        if i == 500:
            nn.lam = nn.lam*3/4
        if i == 1000:
            nn.lam = nn.lam/2
```

```
np.random.shuffle(data)
k = 0
print("Epoch " + str(i))
batch_cost = 0
batch_error = 0
for l in range(int(N_s/M)):
    for j in range(M):
        (D1, D2, DC) = extract_D_mat(data, k)
        nn.update_phases(D1, D2, DC)
        nn.forward_pass()
        nn.calc_cost()
        nn.backwards_prop()
        k += 1
    nn.learn_step()

    # --Store values for plotting--
    bias_list[0][m,:] = [node.bias for node in nn.input_layer]
    w_list[0][m, :] = [node.w for node in nn.input_layer]
    th_list[0][m, :] = [node.theta for node in nn.input_layer]
    for p, layer in enumerate(nn.layers):
        w_list[p + 1][m, :] = [node.w for node in layer]
        th_list[p + 1][m, :] = [node.theta for node in layer]
        bias_list[p + 1][m, :] = [node.bias for node in layer]
    cost.append(np.sum(nn.C)/(2*M))
    batch_cost += np.sum(nn.C)/(2*M)
    batch_error += np.sum(nn.error) /M
    print(np.sum(nn.error)/M)
    nn.error = []
    nn.C = []
    if inter_plot:
        axs[0, 0].clear()
        axs[0, 1].clear()
        axs[0, 2].clear()
        axs[0, 0].plot(range(m), w_list[0][0:m, :])
        axs[0, 1].plot(range(m), th_list[0][0:m, :])
        axs[0, 2].plot(range(m), bias_list[0][0:m, :])
        for l in range(nn.N - 2):
            axs[l + 1, 0].clear()
            axs[l + 1, 1].clear()
            axs[l + 1, 2].clear()
            axs[l + 1, 0].plot(range(m), w_list[l+1][0:m, :])
            axs[l + 1, 1].plot(range(m), th_list[l + 1][0:m, :])
            axs[l + 1, 2].plot(range(m), bias_list[l + 1][0:m, :])
```

```
                axs[nn.N - 1, 2].clear()
                axs[nn.N - 1, 2].plot(range(m), bias_list[nn.N - 1][0:m, :])
            m += 1

        epoch_error[i] = M*batch_error/N_s
        epoch_cost[i] = batch_cost/(N_s/M)
        if np.mod(i,100) == 0:
            write_dmn(nn, f"{ind}_{i/100}")

        if inter_plot:
            axs[0,0].set_title(f"weights")
            axs[0,0].set_xlabel("learning steps")
            axs[0,0].set_ylabel("activation")

            axs[0,1].set_title(f"thetas")
            axs[0,1].set_xlabel("learning steps")
            axs[0,1].set_ylabel("rot angles")

            axs[0, 2].set_title(f"Bias")
            axs[0, 2].set_xlabel("learning steps")
            for j in range(nn.N - 2):
                axs[j + 1, 2].set_title(f"Bias")
                axs[j + 1, 2].set_xlabel("learning steps")

                axs[j+1,1].set_title(f"thetas")
                axs[j+1, 1].set_xlabel("learning steps")
                axs[j+1, 1].set_ylabel("rot angles")

                axs[j+1, 0].set_title(f"weights")
                axs[j+1, 0].set_xlabel("learning steps")
                axs[j+1, 0].set_ylabel("activation")
            # for j in range(nn.N - 2):
            #     axs[j].set_title(f"weights")
            #     axs[j].set_xlabel("learning steps")
            #     axs[j].set_ylabel("activation")
            axs[nn.N-1,0].clear()
            axs[nn.N-1,0].set_yscale("log")
            axs[nn.N-1,0].set_title("Error")
            axs[nn.N-1,0].set_xlabel("Epochs")
            axs[nn.N-1,0].plot(range(i), epoch_error[0:i])

            axs[nn.N-1,1].clear()
            axs[nn.N - 1, 2].set_title("Bias")
            axs[nn.N - 1, 2].set_xlabel("learn steps")
            axs[nn.N-1,1].set_yscale("log")
            axs[nn.N-1,1].set_title("Cost")
            axs[nn.N-1,1].set_xlabel("Epochs")
            axs[nn.N-1,1].plot(range(i), epoch_cost[0:i])
            fig.canvas.draw()
            fig.canvas.flush_events()
    plt.savefig(f"train_sample_{ind}_run.svg")
    print("runtime: " + str(time.time() - start_time) + " seconds")
    return nn, epoch_cost, epoch_ws


def showcase_temp_variation(dmn):
    T = np.linspace(20, 150, 100)
    E_1 = 6.2225*10**4 - 75*(T + 273.15)
    nu_1 = 0.36
    G_1 = E_1/(2*(1+nu_1))
    E_2 = 9*10**4
    nu_2 = 0.35
    G_2 = E_2 / (2 * (1 + nu_2))
    #  isotropic assumption
    E_out = np.zeros((100, 1))
    E_reuss = np.zeros((100, 1))
    E_voigt = np.zeros((100, 1))
    for i, t in enumerate(T):
        D_1 = np.array([[1/E_1[i], -nu_1/E_1[i], 0], [-nu_1/E_1[i],
         ↪  1/E_1[i], 0], [0, 0, 1/G_1[i]]])
        D_2 = np.array([[1/E_2, -nu_2/E_2, 0], [-nu_2/E_2, 1/E_2, 0], [0,
         ↪  0, 1/G_2]])
        C_reuss, C_voigt, D_reuss, D_voigt = elast_bounds(D_1, D_2)
        dmn.update_phases(D_1, D_2, np.zeros((3, 3)))
        dmn.forward_pass()
        D_out = dmn.get_comp()
        E_out[i] = 1/D_out[0, 0]
        E_voigt[i] = C_voigt[0, 0]
        E_reuss[i] = C_reuss[0, 0]
    fig = plt.figure()
    plt.plot(T, E_1)
    plt.plot(T, E_2*np.ones((100, 1)))
    plt.plot(T, E_out)
    plt.plot(T, E_voigt)
    plt.plot(T, E_reuss)
    plt.title("Temperature varying phase 1")
    plt.xlabel(r"Temperature $c^0$")
    plt.ylabel("E [MPa]")
    plt.legend(["Phase 1", "Phase 2", "Homogenized material", "Voigt",
    ↪  "Reuss"])
    plt.savefig("temp_varying.svg")
    return E_out


def validate_multiple(data_set):
    error = np.zeros((180, 1))
    for i in range(180):
        if i < 50:
            ind = f"dmn_3337330_{i}.0"
        elif i >= 50 and i <80:
            ind = f"dmn_3337331_{i-50}.0"
        elif i >= 80 and i <130:
            ind = f"dmn_3337332_{i-80}.0"
        else:
            ind = f"dmn_3337333_{i-130}.0"
        dmn = create_dmn_from_save(f"best_3/{ind}")
        error[i], error_v, error_r = run_validation(dmn, data_set)
    fig = plt.figure()
    plt.plot(range(180), error)
    plt.title("Error during training")
    plt.xlabel(r"100 Epochs")
    plt.ylabel("Error")
    plt.xscale("log")
    plt.yscale("log")
    plt.savefig("validation_error_morph_3.svg")


def run_validation(nn, valid_set):
    """
    :param nn: the DMN
    :param valid_set: data_set for validation.
    :return: validation cost. (Without cost function addition)
    """
    N_s = len(valid_set)
    nn.C = []
    nn.error = []
    voigt_error = []
    reuss_error = []
    comp_correct = np.array([[1, 1, 1], [0, 1, 1], [0, 0, 1]])
    for i in range(N_s):
        (D1, D2, DC) = extract_D_mat(valid_set, i)
        C_reuss, C_voigt, D_reuss, D_voigt = elast_bounds(D1, D2)

        reuss_error.append((np.linalg.norm(comp_correct * (D_reuss - DC),
         ↪  'fro')) / (np.linalg.norm(comp_correct * DC, 'fro')))
        voigt_error.append((np.linalg.norm(comp_correct * (D_voigt - DC),
         ↪  'fro')) / (np.linalg.norm(comp_correct * DC, 'fro')))
        nn.update_phases(D1, D2, DC)
        nn.forward_pass()
        nn.calc_cost()
    error = np.sum(nn.error)/N_s
    error_v = np.sum(voigt_error)/N_s
    error_r = np.sum(reuss_error)/N_s
    return error, error_v, error_r

#data_file = "data_comb"
data_file = "data_set_4"
#data_file = "Symdata2"
#data_file = "data_set_2"
#data_file = "comb_3_v2"
#data_file = "comb_2n5"
data = read_dataset(data_file)
run_scatter_plot()
new = False
#nn_old = create_dmn_from_save("DMN_1406")
#valid_cost, error_v, error_r = run_validation(nn_old, data)
#showcase_temp_variation(nn_old)
#validate_multiple(data)
print(1)
if new:
    N_s = 100
    N = 5
    mini_batch = 25
    ind = 1780
    nn, epoc_cost, epoch_ws = run_train_sample(5000, mini_batch, ind, N_s,
     ↪  data_file, N=N, inter_plot=True, update_lam=False )
    write_dmn(nn, ind)
    write_data(epoc_cost, epoch_ws, ind)
else:
    N_s = 100
    mini_batch = 25
    ind = 4
    nn_old = create_dmn_from_save(f"best_dmns/DMN_{ind}")
    nn, epoc_cost, epoch_zs = run_train_sample(1, mini_batch, ind+1, N_s,
     ↪  data_file, nn=nn_old, inter_plot=True, update_lam=False)
    write_dmn(nn, ind+1)
    write_data(epoc_cost, epoch_zs, ind+1)

valid_data = read_dataset(data_file)
valid_cost, error_v, error_r = run_validation(nn, valid_data)
print("validation cost: " + str(valid_cost))
print(1)
```

### C.3.3 Requirements

Module requirements for all python scripts.

pystencils
numpy
ezdxf
re
sympy
scipy
time
os
matplotlib

DEPARTMENT OF INDUSTRIAL AND
MATERIALS SCIENCE
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY