Lehrstuhl für Technische Mechanik

Python-based constitutitve model calibration of thermosets in Abaqus focusing on elastoplasticity and material damage

Projektarbeit

Marlene Ziegler
22522807

27.10.2025
Dr.-Ing. M. Ries

Friedrich-Alexander-Universität
Technische Fakultät

Projektarbeit

im Studiengang Maschinenbau

Python-based constitutive model calibration of thermosets in Abaqus focusing on elastoplasticity and material damage

von

Marlene Ziegler

Prüfer:	PD Dr.-Ing. habil. S. Pfaller
Betreuer:	Dr.-Ing. M. Ries
Ausgabe:	31.05.2025
Abgabe:	27.10.2025

Universität Erlangen-Nürnberg
Lehrstuhl für Technische Mechanik
Prof. Dr.-Ing. habil. P. Steinmann

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ort, Datum	Unterschrift

Contents

Acronyms

API application programming interface.

CAE computer aided engineering.

EER evaluated strain reactions.

ER evaluated reaction.

ESR evaluated stress reactions.

FAU Friedrich-Alexander Universität Erlangen-Nürnberg.

FE finite element.

MD molecular dynamics.

MDB model database.

MSE mean squared error.

ODB output database.

OLR optimised load reactions.

OMP optimised material parameters.

PBC periodic boundary conditions.

RLR reference load reactions.

RMSE root mean squared error.

List of Figures

List of Tables

1 Introduction

Polymers are an irreplaceable component of many applications [0]. Because of their adaptability, they can be used under various conditions. The addition of additives allows polymers to be used in applications with the highest technical requirements [0], [0]. These so-called composites are constructed by embedding a reinforcing material in form of e.g. fibres, in a polymer matrix [0]. The use of fibre-reinforced polymers is facilitated through adhesive bonding, which allows joining different materials without damaging them [0]. A widely-used group of adhesives are the epoxies, since they possess excellent mechanical properties, and high chemical resistance [0]. They consist of a resin and a hardener which are combined in specific mixing ratios. Adhesive bonding is an increasingly popular joining technique due to its applicability for composite materials, and their beneficial loading properties [0], [0]. The extension of usage in further applications depends on a profound understanding of their material behaviour. For investigations on atomistic level molecular dynamics (MD) is a widely used approach [0]. It is capable to model the curing and cooling procedures, and mechanical deformations of various materials [0] with MD simulations. However, investigations on atomistic level require high resolutions in space and time which cause high computational costs. Because of this issue, only small-scaled simulations are possible. If the findings are extrapolated on larger dimensions, a sufficient transferability must be ensured. To transfer the governed results to real-life applications, adequate engineering quantities must be extracted. For the description of the mechanical behaviour, the transfer takes place through material parameters. If the material parameters are known, the material behaviour can be calculated through functional relations, so-called constitutive models. Hence, an adaptation to the current problem is easily possible. However, with MD simulations a direct extraction of the material parameters is not possible. This is because of the fact, that the MD simulation is a method on an atomistic level, whereas material parameters are defined for a continuum-based perspective. Consequently, a continuum-based method such as the finite element (FE) method must be used. With FE analysis, the loading process on arbitrary bodies can be simulated. In the FE analysis, the mechanical responses for an applied load case is determined through the previously defined constitutive model and the material parameters. Conversely, for known material behaviour to an applied load, and a defined constitutive model, an inverse identification of the material parameters is enabled. This approach can be applied used in epoxy investigations to extract material parameters from the mechanical responses measured in MD simulations. Therefore, a procedure is required that is capable of integrating MD simulation results into a FE simulation to identify corresponding material parameters. In addition, the approach should be easy to handle by a user and require low computing power.

Scope of this work In this work, an algorithm is developed, that determines material parameters by FE analysis by considering the material properties detected via MD simulations. Therefore, the material behaviour in the simulation methods need to be compared. Based on the correspondence of the mechanical responses, the material parameters of the FE simulation are evaluated. This leads to an optimisation problem to

calculate the material parameters that fit best the mechanical behaviour detected in the MD simulations. The developed approach will be used to answer the following research questions: 1. Can we qualitatively replicate the mechanical responses generated by MD simulations, using FE analysis? 2. Can we identify unique material parameters which lead to appropriate matching mechanical responses? 3. Which impact have different loading conditions on the optimisation capability? To this end, we first explain the general characteristics of MD and FE analysis (Section 2.1 and Section 2.2), and present the selected tools to implement the optimisation procedure (Section 2.3 and Section 2.4). Then, the setup of the developed optimisation approach is outlined (Chapter 3). Afterwards, we verify the results and discuss possible issues (Section 4.1 and Section 4.2). The capability of the code is tested through multidimensional load applications and cyclic loadings. Finally, we discuss the dependency of the optimisation performance on the loading conditions and detect possible improvements (Section 4.3 and Section 4.4).

2 Basics

This chapter introduces the foundational methods used in this work. Since we process data from molecular dynamics (MD) simulations as input, we present the concept of this modelling strategy. Following, the basics of FE is presented, because the developed approach is based on a FE software. The last section outlines the procedure of the Nelder-Mead algorithm which performs the numerical optimisation, and defines the required return value.

2.1 Molecular dynamics

The MD simulation is a frequently used method for biological, chemical and physical investigations on a microscopic level [0]. Simulations at the atomistic level allow studies of the structure, interaction processes and energy state of a molecular system [0]. In MD simulations, Newton's equation is solved for each atom from the interactions with its neighbouring atoms. These interactions are modelled by potentials. Non-bonded interactions, such as van der Waals potentials, are considered within a cutoff radius [0]. The total potential energy of the system is used to identify the acting forces and accelerations of each particle. To follow the movements of the particles, time integration is necessary. Typical time step sizes are in the femtoseconds range, which makes only small time scales possible with reasonable computational costs [0]. Similar restriction holds for the system size, because of the increasing number of interactions with increasing domain dimensions. However, small dimensions lead to large surface-to-volume ratios which result in significant free surface effects. To avoid them, periodic boundary conditions (PBC) are used. They constrain the simulated volume as if it were integrated in an infinitely large domain. Regarding particle tracking, a particle that leaves the system at one surface, enters the system at the opposite surface. For the deformation of the whole volume element, the PBC restrict it in such a way that parallel surfaces remain parallel during the loading procedure. These boundary conditions result in a simulation of an infinitely long concatenation of the same volume element in each direction [0]. With these adaptations the results from MD simulations can be transferred to a larger system. Thus, MD simulations allow building samples with prescribed properties, followed by deformation tests to study the material behaviour [0].

Constitutive models During a deformation test, stress and strain components are measured for every time step in the simulation. This leads to a series of discrete points that describe the evolution the stresses and strains during the loading process. To deduce general relationships between the stresses and the strains from discrete measurements, constitutive models are required for a mathematical description [0]. The material specific properties are considered through material parameters. Depending on the deformation regime for which the constitutive model holds, different material parameters are useful. The material behaviour of polymers can be represented in an elastoplastic model [0]. Elastoplastic models combine two characteristic material behaviours – elasticity and plastification. An elastic process is characterised by the fact that the loading process is

reversible. This means, the loading and unloading processes follow the same path in a stress-strain diagram [0]. Generally, the path can follow any function. In combination with plastification, usually linear elastic behaviour is assumed [0], which can be specified with the material parameters Young's modulus E and Poisson's ratio ν. An elastoplastic material behaves linear-elastic until a stress limit is reached. After that so-called yield stress σ_0, the material response is irreversible, which leads to the exemplary stress-strain function in Figure 2.1 [0]. The loading path in the plastic regime can be described by various functions. In the scope of this work, we focus on hardening functions, where the stresses increase with increasing plastic strain [0]. They describe the material behaviour in the plastic regime through multiple plastic material parameters. An important property of elastoplastic material models is their rate-independence. These characteristics lead to identical material behaviour under loading processes with varying strain rates. However, this assumption is not valid for materials with viscous properties. The material response of polymers can include viscous parts as well, meaning their behaviour can be rate-dependent. Therefore, if an elastoplastic constitutive model is to be used, an adequate procedure must be employed to filter out the viscous component of the material response.

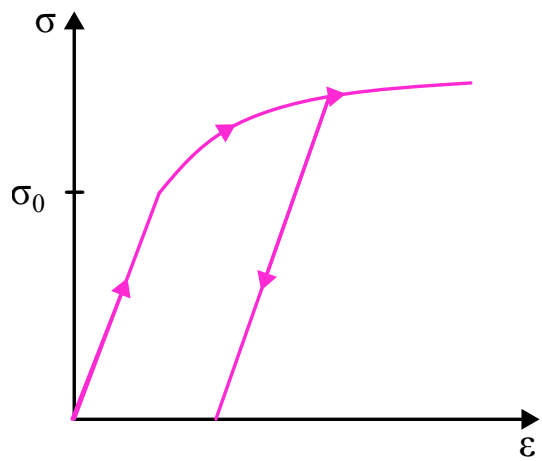


Figure 2.1: Exemplary stress-strain curve of an elastoplastic model with linear elastic behaviour up to the yield stress σ_0 and following plastic hardening based on [0].

Previous work This work focusses on the investigations by Ries et al. [0], who studied the curing and deformation properties of epoxy through MD simulation. They developed MD models for materials with numerous mixing ratios of resin and hardener[1]. The deformation tests from Ries et al. [0] build the motivation for the here developed optimisation approach. Ries et al. [0] performed uniaxial tensile tests, loading samples with linear strain up to a maximum value of 20%. The test samples were constrained by PBC, which allow lateral contractions. To record the stress and strain components without viscous effects, they developed a procedure to approximate the quasi-static material response. Therefore, only elastic and plastic reactions were considered. Their choice of constitutive models was based on the assumption of isotropic material behaviour. To describe the elastic material behaviour, they used the Neo-Hookean hyperelasticity model. The plastic reactions were modelled by the VOCE-model which defines the stresses during the hardening process through [0]:

[1] In the following the mixing ratio is specified in the notation resin:hardener

$$\sigma = \sigma_0 + \alpha(1 - \exp(-\beta\varepsilon_{pl})) + \gamma\varepsilon_{pl} \tag{2.1}$$
$$\sigma_0 : \text{ Yield stress}$$
$$\varepsilon_{pl} : \text{ Plastic strain}$$
$$\alpha, \beta, \gamma : \text{ Hardening parameters}$$

Together with the elastic material parameters Young's modulus E and Poisson's ratio ν, six constitutive parameters were available to fit the stress–strain data measured through MD simulation. The values of the parameters were calculated, using an external minimisation algorithm, which detailed procedure is described in [0]. The procedure of Ries et al. [0] is important since their data are used for the model assessment of the optimisation algorithm developed in this work. A detailed description of the optimisation setup is given in Chapter 3. In the validation studies of this work, optimisation tests were performed with materials mixing ratios 4:3, 6:3 and 8:3. Ries et al. [0] also used these mixing ratios to validate their results. Therefore, the optimisation process can easily be evaluated through a comparison of the obtained material parameters. However, a valid comparison requires two conditions: first, that stress and strain data are collected under similar loading conditions, – and – second, that the same constitutive model is used to determine the material parameters. Therefore, a detailed understanding of the methods used by Ries et al. [0] is essential, since they are adopted for the simulation process used in this work.

2.2 Finite element method

The finite element (FE) method is a widely used approach in engineering domains like heat transfer in solids, deformations of solids through prescribed loading and interactions between solid structures and fluids [0]. The purpose of FE method is to find solutions for field problems in complex regimes [0]. However, an analytical solution is only possible for simple problem formulations. Therefore, in FE analysis, the regime is discretised into a finite number of elements. The element behaviour is characterised by approximation functions with a finite number of parameters. Assembling the approximation functions of all elements, leads to an equation system that approximates the solution for the whole regime [0]. The FE method is primarily used in structural mechanics to provide information about forces and deformations. The general procedure of the FE method, based on Willner [0] and Steinke [0], is presented in the following:

1. Discretisation
2. Construction of stiffness matrix
3. Coordinate transformation
4. Assembly
5. Application of boundary conditions
6. Solving equation system

In the first step, we discretise the continuum in finite elements. The shape and size of the elements depend on the geometry of the regime, and the required level of precision. To achieve more accurate solutions, smaller elements are necessary. Next, a local stiffness

matrix $\boldsymbol{K}$ is created for every element, which connects the acting forces $\boldsymbol{S}$ with the element deformations $\boldsymbol{u}$ via

$$\boldsymbol{S} = \boldsymbol{K} \cdot \boldsymbol{u}. \tag{2.2}$$

Although the equation holds for an element, the calculated forces and deformations are defined at the element nodes. The entries in the stiffness matrix depend on the used element type. They contain material-specific information defined by material parameters. The stiffness matrices are constructed in local coordinate systems. To connect them, a transformation into a global system is necessary. In the fourth step, the equation systems of all elements are combined into a global matrix system. In the assembly, neighbouring elements share their nodes, which needs to be considered during the construction of the equation system. Through prescribed loadings at the boundary of the continuum, certain forces or deformations are known. They are inserted in the equation system as boundary conditions. In the final step the equation system is solved, yielding the displacements for every node [0], [0].

Application A main advantage of FE method is its high flexibility. Many different geometries can be modelled through an appropriate choice of element shapes. The accuracy can be adjusted with the element size. Decreasing element sizes lead to more accurate results but require higher computational effort. However, FE simulations are normally quite fast for easy element geometries. In addition, multiple commercial tools are available for FE simulations. They offer multiple options to define the properties throughout the entire analysis, which makes them applicable in a wide range of problems.
As described in Chapter 1, the aim of this work is to create an approach to determine the material parameters of materials, investigated with MD simulations. Since a FE simulation requires significantly less computational effort whilst still producing sufficiently accurate results, we decided to base this work on FE analysis. Therefore, the model used in the MD simulations must be transferred into a FE model. As reference, we use the MD simulations performed by Ries et al. [0]. To achieve this, we must transfer the model properties and testing conditions into a FE environment. In particular, the material behaviour, the boundary conditions, and the applied load must be transferred as exactly as possible. A description of the implementation is given in Chapter 3.

2.3 Abaqus scripting interface

The task addressed in this thesis is implemented using Abaqus/CAE 2024. The commercial software is a widespread tool for FE analysis. At the Institute of Applied Mechanics (LTM) of the Friedrich-Alexander Universität Erlangen-Nürnberg (FAU), the software is frequently used, which simplifies subsequent works with the algorithm. In addition, Abaqus has an integrated Python-based scripting tool called "Abaqus Scripting Interface". It works as an application programming interface (API) to use the object-oriented programming language Python in the Abaqus environment [0]. The "Abaqus Scripting Interface" allows access to the functionalities of Abaqus/CAE from scripts. Functions, such as the creation and modification of models and jobs, can be controlled via code. Additionally, the output data written for successful executed jobs can be processed [0]. Since it is a Python extension, the standard programming functionalities are also available. The integration of Abaqus functionalities in a standard program structure, makes the "Abaqus Scripting Interface" an appropriate tool for the realisation of the task. Because of this property, the implementation is possible in a single script whose structure

is shown in Chapter 3. The implementation permits a parameter-based analysis which makes value adaptations very fast and easy. If these parameter values are stored in an input file, the user only needs to modify the input-file. This enables fast parameter studies with multiple values, which is used in this work to test multiple combinations of material parameters.

2.3.1 EasyPBC plugin

EasyPBC is an Abaqus plugin which automatically creates PBC. The plugin was developed by Omairey, Dunning, & Sriramula [0]. It is not an official Abaqus extension, thus it is not available online. Because of its utilisation in previous works, the plugin was already in use at the institute. The PBC must constrain parallel surfaces to remain parallel during deformation. To realise this property in Abaqus, EasyPBC generates node sets of the surface nodes. Opposing nodes are linked via constraint equations to couple their motion. In addition, reference points are created with each one linked to a surface. Therefore, applying load to a reference point causes corresponding reactions on the connected surface which is used for a simplified load application.

2.4 Mathematical basics

To find the material parameters, that best fit the material behaviour measured in the MD-simulation, a mathematical formulation is necessary. This leads to an optimisation problem, where a calculated error, defined as an objective function of the material parameter values, should be minimised. First, we discuss the numerical method to minimise the error, followed by the construction of the error value.

2.4.1 Numerical optimisation

To solve an optimisation problem various mathematical algorithms are available [0]. We decided to use the Nelder-Mead algorithm, which is a widely used gradient-free optimisation algorithm [0]. In a gradient-free algorithm, the derivatives of the function are neglected in the process. The objective function is based on the results of the FE analysis, which makes it impossible to determine its derivatives directly. Therefore, only gradient-free algorithms are applicable. In addition, ignoring the derivatives saves significant computational costs, which leads to fast convergence times [0]. Because of its simple structure, the algorithm is a standard feature in many numerical libraries [0]. In Python, it is available in the `scipy.minimize()` function. The function call is described in detail in Section 3.4. Here, we focus on the procedure of the numerical algorithm, which is visualised in Figure 2.2. The Nelder-Mead algorithm is capable of finding a local minimum of a scalar function, depending on n optimisation variables. In this work, the optimisation variables are the material parameters. The definition of the objective function can be found in Chapter 3. Assuming the objective function is known, the first step is to create $n + 1$ points $\mathbf{P}$ in an n-dimensional space. In Figure 2.2, an exemplary numerical optimisation with three optimisation variables is depicted. First, the positions of the points must be determined. This is done by an initial guess $\hat{x}$ for every optimisation variable. To process three optimisation variables, the initial guess would look like this:

$$\hat{\mathbf{x}} = [\hat{x}^0, \hat{x}^1, \hat{x}^2] \tag{2.3}$$

with $\hat{x}^i \equiv$ initial guess of the i-th optimisation variable

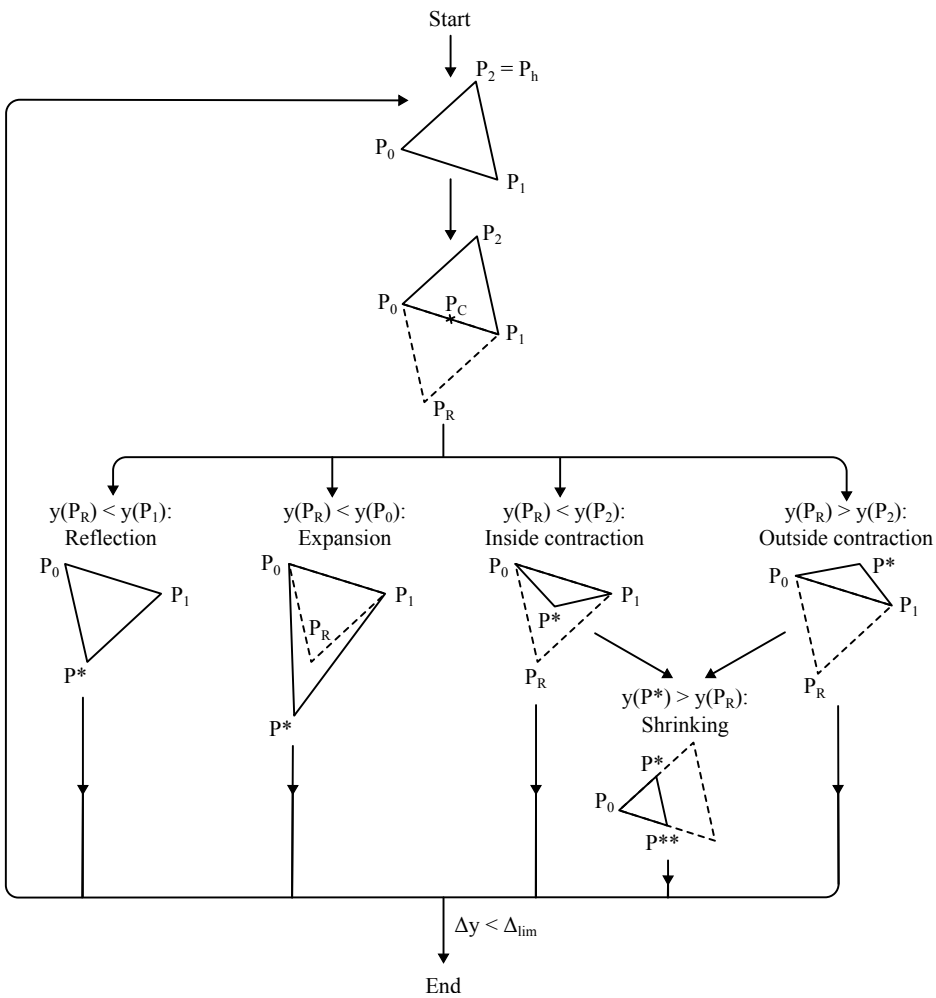


Figure 2.2: Pocedure of the Nelder-Mead algorithm for an exemplary optimisation with three optimisation variables.

Based on Equation 2.3, the initial points $\hat{\mathbf{P}}_\mathbf{i}$ are constructed. The first one is defined as $\hat{\mathbf{P}}_\mathbf{1} = \hat{\mathbf{x}}$. For the other points, the value of one variable in the initial guess is changed each. The points are connected to each other in such a way that an n-dimensional simplex is created. A simplex is a general geometric object in n-dimensional space consisting of $n+1$ points. At the top of Figure 2.2 it can be seen, that in two-dimensional space a simplex is equivalent to a triangle. In the next step, the function values corresponding to the points $\mathbf{P}_\mathbf{i}$ are evaluated and sorted by size. The highest function value y_h maps the worst value combination $\mathbf{P}_\mathbf{h}$ of the optimisation parameters, as signed at the upper triangle in Figure 2.2. When the algorithm starts, a centroid of all points of the simplex except $\mathbf{P}_\mathbf{h}$ is determined. At this point $\mathbf{P}_\mathbf{C}$, $\mathbf{P}_\mathbf{h}$ is reflected at a new position $\mathbf{P}_\mathbf{R}$, which is depicted in the second triangle in Figure 2.2. Before the new point $\mathbf{P}^*$ is positioned, the corresponding function value $y(\mathbf{P}_\mathbf{R})$ needs to be evaluated. Depending on its value, one of

the four possible operations presented in Figure 2.2 is executed. The operation *Reflection* is performed, if $y(\mathbf{P_R})$ is smaller than the second-worst function value. If $y(\mathbf{P_R})$ is even smaller than $y(\mathbf{P_0})$, which is currently the best value, an *Expansion* is performed. In the other cases, one of the *Contraction* operations is executed. The worst case occurs, if both *Contraction* operations bring an increased function value $y(\mathbf{P}^*)$ compared to $y(\mathbf{P_R})$ and $y(\mathbf{P_h})$. As a consequence, a *Shrinking* operation is required to minimise the whole simplex towards the currently best position. This is shown in the bottom right of Figure 2.2, where $\mathbf{P_1}$ and $\mathbf{P_2}$ are moved closer to the current minimum point $\mathbf{P_0}$. To choose the correct operation depending on the current simplex, multiple evaluations of y at different points $\mathbf{P}$ might be necessary. Thus, multiple function evaluations are accomplished during one optimisation iteration. When an improved position $\mathbf{P}^*$ is found, the algorithm starts again with the new simplex [0]. If the variations of the functions values y_i meet a certain lower limit, the minimum of the function with its corresponding parameter is found which is marked at the bottom of Figure 2.2.

To ensure a successful search, the initial simplex should be scaled regularly [0] which is possible through a regular distribution of the points $\hat{\mathbf{P}}_\mathbf{i}$ in space. This can be difficult if the values of the optimisation variables differ greatly in size. Therefore, it is necessary to normalise the variable values within the range of 0 to 1. The algorithm is vulnerable to becoming stuck in local minima because of the *Shrinking* operation [0]. Therefore, a smart choice of initial values is helpful, to avoid starting points near a local minimum. However, if the trend of the objective function is unknown, this can be challenging. In addition, the number of optimisation parameters should be constrained. So far, stable convergence behaviour of the Nelder-Mead algorithm has mostly been studied for small numbers of variables [0], [0].

2.4.2 Root mean squared error

To use the Nelder-Mead algorithm, we need to construct an objective function that returns a scalar value. It should be preempted at this point that several values must be minimised for an adequate optimisation result. In order to handle this issue, it is necessary to condense all applicable data into a single value. As a representative value, we choose the root mean squared error (RMSE). It is a frequently used value to express variations of two data sets - usually a reference data set f and a test data set $\hat{f}$ [0]. The RMSE is based on the difference between the data points at position i. This deviation is composed for all points, and then combined in the following formula

$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(f_i - \hat{f}_i)^2} \tag{2.4}$$

With this definition, the proportion of all deviations in the RMSE is equal. The value of the RMSE is always positive and tends towards zero for perfectly matching data sets. In addition, the unit of the RMSE matches that of the data points. These characteristics are advantageous for the evaluation of its value. Since we must include the deviations between M multiple data sets, we extend the formulation to

$$\mathrm{RMSE} = \sqrt{\frac{1}{M}\sum_{j=1}^{M}\left[\frac{1}{N}\sum_{i=1}^{N}(f_i - \hat{f}_i)^2\right]_j} \tag{2.5}$$

In this formulation, the RMSE is only unit-based, if all data sets have the same unit. Otherwise, the units of each data set are neutralised through weights with the corre-

sponding inverse unit. Therefore, the resulting RMSE is unitless. The application of Equation 2.5 in the developed algorithm is explained in Section 3.2.

3 Algorithm setup

To extract engineering quantities for materials modelled with MD simulations, an easy and fast approach is needed to determine material parameters appropriate to the results from the MD simulations. With deformation tests in MD simulations, the mechanical response can be recorded. The aim of the developed optimisation algorithm is to find material parameters which best represent the mechanical behaviour. In this chapter, we describe the workflow of the optimisation approach. First, we have a closer look at the structure of the approach. Then, we introduce the required input data, and finally, the implementation is explained.

In deformation tests performed with MD simulations, the material behaviour during the loading process is recorded (see Section 2.1). Therefore, stress and strain components in all directions at discrete simulation time steps are available. Stress and strain data, measured during a loading process, are referenced as *load reactions* in the following. In Subsection 3.1.2, we present their structure in detail. To extract material parameters from these load reactions, a constitutive model is required, which describes the stress-strain relation of a material through a functional relationship. The constitutive model, with its corresponding material parameters, used in this work is presented in Section 2.1. The evaluation of the material parameters is achieved by a FE simulation, as described in Section 2.2. The FE simulation returns the stress and strain responses from a material, based on its prescribed material parameters, constitutive model and loading conditions. Consequently, load reactions measured in MD simulations can be compared with the ones computed in FE simulations. To represent the mechanical behaviour measured in MD simulations best, a small difference, i.e. a good match, between the FE and MD load reactions is favourable. Since the material parameters define the load reactions in the FE simulation, their quality is implicitly measured. In other words, we have to minimise the deviations to the load reactions measured with MD simulations to find the best material parameters, which is often referred to as inverse parameter identification. We use the Nelder-Mead algorithm, introduced in Subsection 2.4.1, to perform this optimisation. The numerical algorithm is capable to minimise the value of a scalar function by optimising multiple parameters. Its function value defines the quality of the optimised material parameters. Since we want to fit the whole loading process, the deviations at all steps of the loading procedure should be taken into account. To achieve this, we design an expression, explained in Section 3.2, to summarise all these differences into a single error value.

3.1 Input data

Next, the required input data are introduced. There are multiple types of input data which are processed at different steps in the algorithm. In order to ensure the traceability, clear definitions are required for the inputs at every step.

3.1.1 Load cases and evaluated reactions

A *load case* defines the direction in which a load acts, i.e. the deformation direction. For the experiment, reproducible and easy cases are preferable, so we only allow loading in normal and principal shear directions. We use the ABAQUS plug-in EasyPBC to apply these loadings. For a consistent naming, we adopt the signatures from EasyPBC for the load cases, assigned in Table 3.1 (a). To model a more complex loading situation, it is possible to apply a series of different load cases. For example, we can apply a tensile strain in xx-direction, followed by a shear strain in xz-direction. Nevertheless, in one load case, only one direction is considered to avoid mutual influence. The optimisation algorithm requires the load reactions without any constraints, for every load case. The only applied constraints are the PBC which are described in Section 2.1. After the application of a load case, we have to decide which material responses we use to compare with the load parameters. We have the possibility to read out the stress and strain components in all normal and shear directions (see Table 3.1 (b)). The quantities we choose for the comparison are called *evaluated reactions*. For a high accuracy of our material parameters, we try to choose evaluated reactions which provide the most information about the material behaviour. These measurements vary depending on the applied load case. In Figure 3.1 an exemplary load case E11 (green) with possible corresponding evaluated reactions (yellow) is depicted.

Table 3.1: Mapping of load directions and list of available evaluated reactions.

(a) Mapping of load directions to load cases with adopted names from EasyPBC.

Load direction	Load case
xx	E11
yy	E22
zz	E33
xy	G12
yz	G23
xz	G13

(b) List of all available evaluated stress and strain reactions.

Evaluated stress reactions	Evaluated strain reactions
σ_{xx}	ε_{xx}
σ_{yy}	ε_{yy}
σ_{zz}	ε_{zz}
σ_{xy}	ε_{xy}
σ_{yz}	ε_{yz}
σ_{xz}	ε_{xz}

Application In Table 3.2 the studied test cases are listed with the corresponding load cases and evaluated reactions. For the verification of the algorithm, we use the simple tensile load case E11. In all other directions, we impose no restrictions except the PBC. As evaluated reaction, we use σ_{xx} and the lateral strains ε_{yy} and ε_{zz}. The normal stress permits to deduce the Young's modulus and the plastic parameters. The lateral strains are necessary for the identification of the Poisson's ratio. Applying a strain in xx-direction will lead to decreasing dimensions in yy- and zz-direction to keep a state of minimum stress. Simultaneously, this means that the lateral stresses do not contain any useful information, because they are numerically zero. The validation study is realised with the same load case. Equivalent to the verification study, we take σ_{xx}, ε_{yy} and ε_{zz} to extract information about the material parameters. In the Tensile-Shear

combination tests, we handle load case E11, then G12 and finally combine them. Through the additional obtained information, we try to improve the uniqueness of the determined material parameters. As evaluated reaction for the load case G12, we use the stresses σ_{xy}. As a last study, we investigate the application of cyclic loading in the load case E11. We perform this study with varying load parameters (see Subsection 3.1.2).

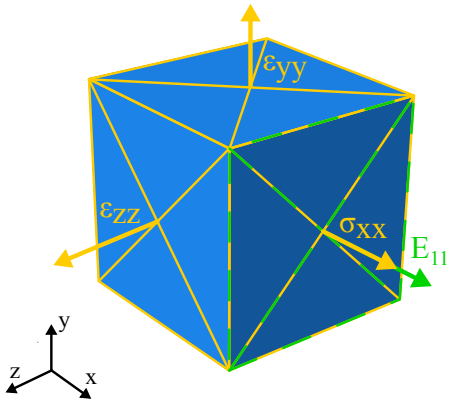


Figure 3.1: Illustration of exemplary load case E11 acting on the front surface in xx-direction (represented in green) with evaluated stress and strain reactions $\sigma_{xx}, \varepsilon_{yy}$ and ε_{zz} on the surfaces (represented in yellow).

3.1.2 Load parameters and load reactions

For the optimisation process we use data from MD simulations as inputs. These data are referenced as *reference data* in the following. They contain stress and strain components for every time step from the MD simulation, in all normal and shear directions. We split the reference data into *load parameters* and *reference load reactions*. Load parameters refer to the quantitative values of the prescribed load case during the loading procedure. Since we only process load cases identical to the ones in the MD simulations, we can transfer the load parameters directly in the FE model. A detailed description of the load application in Abaqus can be found in Section 3.3.

The reference load reactions represent the material response during the MD simulation. From these data we extract the stress and strain components according to the chosen evaluated reaction, and neglect the remaining components, since they contain little information about the material behaviour. As a result, we can register the reference load reactions for the corresponding load parameters to monitor the evolution of the material behaviour during the loading process. To perform the inverse parameter identification, we need corresponding data from the FE simulation. In the way described in Section 3.4, we can read out stress and strain components in all directions during the loading process. Similar to the reference load reactions we extract the components corresponding to the evaluated reaction. These values are called *optimised load reactions*. For an appropriate comparison, we must register the optimised and the reference load reactions at equivalent points in the loading period. In Section 3.3 we explain, how this is implemented in the algorithm.

Application In Table 3.2, the test series with their loading conditions are listed. In the verification and validation studies, we apply a linear tensile strain up to a maximum value of 20%. We analyse different mixing ratios in the validation studies, which demonstrate different mechanical behaviours. As a preparation for studies with cyclic loading, we investigate a pure tensile strain following the first quarter of a sinus period over time with a maximum amplitude of 15% . With this loading trajectory, we study the optimisation behaviour for non-linear loading parameters. We reuse this load parameters, and apply them as a shear strain in load case G12. Since in the previous tests always the load case E11 is used, we test the algorithm performance for another load case. To obtain more information about the material behaviour, we combine the load cases of E11 and G12 in one optimisation. As a last study we investigate one and a half period of a sinusoidal loading for a tensile load case in xx-direction. We use amplitudes of 1%, 5% and 8%. Important to notice is, that the previously introduced load parameters proceed in a wide strain range. Assuming that the material starts to plastify quite fast, the majority of the load steps are located in the plastic domain of the material. Conversely, the load parameters contain only little information about the elastic material behaviour. Through the use of load parameters with small amplitudes, we try to get a larger proportion of data points in the elastic domain. In Section 4.1 the issue about this unequal distribution in the material domains becomes clear.

Table 3.2: Overview of test series with corresponding loading conditions, mixing ratios of the material and evaluated reactions.

Test series	**Load case**	**Load parameters**		**Mixing ratio**	**Evaluated reaction**
		Trajectory	**Amplitude**		
Verification	E11	Linear	20%	6:3	$\sigma_{xx}, \varepsilon_{yy}, \varepsilon_{zz}$
Validation I	E11	Linear	20%	4:3	$\sigma_{xx}, \varepsilon_{yy}, \varepsilon_{zz}$
Validation II	E11	Linear	20%	6:3	$\sigma_{xx}, \varepsilon_{yy}, \varepsilon_{zz}$
Validation III	E11	Linear	20%	8:3	$\sigma_{xx}, \varepsilon_{yy}, \varepsilon_{zz}$
Pure tensile strain	E11	Sinus ($\frac{1}{2}\pi$)	15%	6:3	$\sigma_{xx}, \varepsilon_{yy}, \varepsilon_{zz}$
Simple shear strain	G12	Sinus ($\frac{1}{2}\pi$)	15%	6:3	σ_{xy}
Tensile & Shear strain	E11 G12	Sinus ($\frac{1}{2}\pi$)	15%	6:3	$\sigma_{xx}, \varepsilon_{yy}, \varepsilon_{zz}, \sigma_{xy}$
Cyclic tensile strain	E11	Sinus (3π)	1% 5% 8%	6:3	$\sigma_{xx}, \varepsilon_{yy}, \varepsilon_{zz}$ $\sigma_{xx}, \varepsilon_{yy}, \varepsilon_{zz}$ $\sigma_{xx}, \varepsilon_{yy}, \varepsilon_{zz}$

3.2 Error calculation

For a representative value including all necessary data, we select the RMSE, which is explained in Subsection 2.4.2. We insert the reference load reactions and the optimised load reactions as data sets in Equation 2.5. The extraction of the load reactions from the FE simulation is described in Section 3.4. The access to reference load reactions is explained in Subsection 3.1.2. We compute the deviations of the optimised load reactions at each load step. Figure 3.2 displays this procedure for exemplary sets of reference and optimised load reactions. Here, σ_{xx} is the selected evaluated reaction. For every load step LS their corresponding reference load reactions (RLR) $\sigma_{\mathrm{LS}}^{\mathrm{RLR}}$ and optimised load reactions (OLR) $\sigma_{\mathrm{LS}}^{\mathrm{OLR}}$ are logged. The blue arrow highlights the deviation $\Delta\sigma_{\mathrm{LS}}$ for one exemplary load step, according to Equation 3.1. As described in Subsection 3.1.2, the distribution of the data points is unfavourable for the determination of the elastic parameters. To support the algorithm in finding the elastic parameters, we applied a weight of 100 at the data point in the elastic domain, i.e. below 1% strain. This is achieved by a weight array $\boldsymbol{w}$ which entries are one except the first entry w_1. Formally, the applied weights have the inverse unit of the evaluated load reaction, i.e. for stress load reactions, $[w_{\mathrm{LS}}] = \mathrm{MPa}^{-1}$ would be the correct unit for the related weights. According to Equation 3.2, we calculate the mean value of the weighted arrays. The resulting value is called *mean squared error (MSE)* for one evaluated reaction. We compute MSE_σ or MSE_ε for every selected evaluated reaction.

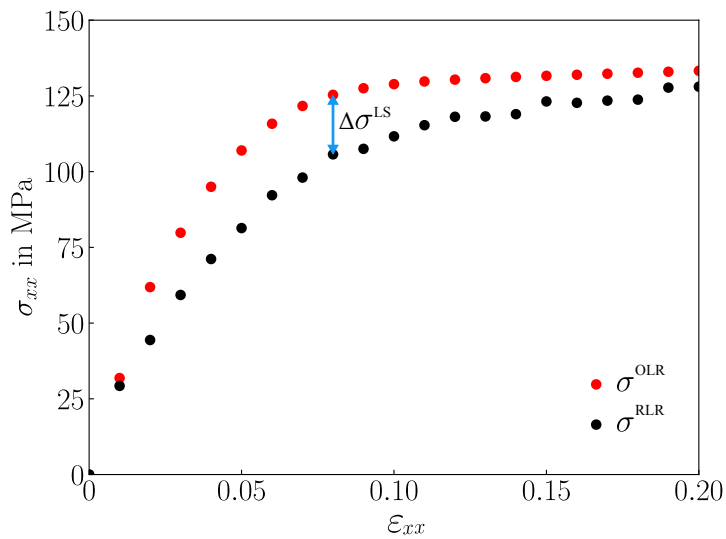


Figure 3.2: Exemplary reference load reactions σ^{RLR} and optimised load reactions σ^{OLR} with visualization of error calculation.

$$\Delta\sigma_{\mathrm{LS}} = \sigma_{\mathrm{LS}}^{\mathrm{RLR}} - \sigma_{\mathrm{LS}}^{\mathrm{OLR}} \qquad \Delta\varepsilon_{\mathrm{LS}} = \varepsilon_{\mathrm{LS}}^{\mathrm{RLR}} - \varepsilon_{\mathrm{LS}}^{\mathrm{OLR}} \tag{3.1}$$

$$\mathrm{MSE}_\sigma = \frac{\sum_{\mathrm{LS}} w_{\mathrm{LS}}(\Delta\sigma_{\mathrm{LS}}^2)}{\sum_{\mathrm{LS}} w_{\mathrm{LS}}} \qquad \mathrm{MSE}_\varepsilon = \frac{\sum_{\mathrm{LS}} w_{\mathrm{LS}}(\Delta\varepsilon_{\mathrm{LS}})^2}{\sum_{\mathrm{LS}} w_{\mathrm{LS}}} \tag{3.2}$$

For the tensile load case E11, for example, we must perform this for the evaluated reaction $\sigma_{xx}, \varepsilon_{yy}$ and ε_{zz}. In order to construct a single error value of these MSEs, we

must ensure a common scale. Otherwise, their influence on the overall error may vary significantly. In general, the MSEs of evaluated strain reactions are much smaller than the ones from evaluated stress reactions, such that loading dependent weights w_σ and w_ε are introduced. The exact weights depend on the load case and the used load parameter set. From the weighted MSE we construct the RMSE, as shown in Equation 3.3. Since the algorithm is able to process multiple load cases in one optimisation run, we can calculate the RMSE for every load case and apply associated weights w_{LC}. Additionally, multiple load parameters sets (LP) can be processed, which leads to RMSE values for every load parameter set with the weight w_{LP}. As stated in Equation 3.4, we assemble the individual RMSEs in a double sum over the load cases and the load parameter sets. This *Error* is the value we return the numerical algorithm. In the following sections we have a closer look at the implementation of this minimisation process.

$$\mathrm{RMSE} = \sqrt{\frac{\sum_{\mathrm{ESR}} w_\sigma \cdot \mathrm{MSE}_\sigma + \sum_{\mathrm{EER}} w_\varepsilon \cdot \mathrm{MSE}_\varepsilon}{N_{\mathrm{ESR}} + N_{\mathrm{EER}}}} \tag{3.3}$$

$$\mathrm{Error} = \sum_{\mathrm{LP}} \sum_{\mathrm{LC}} w_{\mathrm{LP}} w_{\mathrm{LC}} \cdot \mathrm{RMSE}_{\mathrm{LC,LP}} \tag{3.4}$$

N_{ESR} : Number of evaluated stress reactions

N_{EER} : Number of evaluated strain reactions

3.3 Preprocessing

Before starting with the optimisation process, we need preprocessing steps to create a working Abaqus model with the required properties. The user-defined properties are transferred in an input-file. Table 3.3 lists an extract of this file, containing only parameters relevant for the optimisation process. The input file contains more parameters for different namings, which are neglected here. In Appendix B the whole input file is included. The user has multiple options to configure the optimisation process. It is possible to test multiple initial value combinations for the material parameters calling the script once. In Table 3.3 this is visible in the column *Data format* for the material parameters. Here, the user can pass an array with multiple initial value for every material parameter. This function is important to validate the optimisation results with varying input values.

Table 3.3: Input parameters for optimisation process.

Input parameter	Directions	Category	Data format	Unit
Young's modulus	–	value	array	MPa
	–	minimum	scalar	MPa
	–	maximum	scalar	MPa
Poisson's ratio	–	value	array	–
	–	minimum	scalar	–
	–	maximum	scalar	–
Yield stress	–	value	array	MPa
	–	minimum	scalar	MPa
	–	maximum	scalar	MPa
Alpha, beta, gamma	–	value	array	–
	–	minimum	scalar	–
	–	maximum	scalar	–
Load parameters	–	filename	string	–
	–	weight	scalar	–
Load case	E11, E22, E33, G12, G23, G13	active	boolean	–
		weight	scalar	–
Stress evaluation	xx, yy, zz, xy, yz, xz	active	boolean	–
		weight	scalar	–
Strain evaluation	xx, yy, zz, xy, yz, xz	active	boolean	–
		weight	scalar	–
Load weighting	normal stress, normal strain, shear stress, shear strain	weight	scalar	–

In Figure 3.4 the structure of the algorithm is depicted. The complete code is attached in Section B.2. The white boxes show the individual phases referred to in the text in italics. The coloured boxes represent the performed loops. The upper part belongs to the preprocessing, which starts with the phase *Read input file*.

In this phase all entries from the input-file are extracted. To process the arrays with the initial material parameters, the algorithm loops over all arrays at a time to extract one initial value for each parameter. The entries with the same index result in one initial value combination which is visualised in Table 3.4 (a). As a consequence, all arrays need to be of same length. For every initial value combination, the algorithm creates a new model database (MDB) in Abaqus, and a new folder structure to set the working directory and store the results.

We start the first loop with the phase *Create model*. We model a cube with size 1x1x1 and mesh it with 6x6x6 elements. Because of the regular geometry, discretising with hexagonal structured elements is feasible. The number of elements is a compromise between a coarse mesh for fast computation, and a minimum number to avoid convergence errors. Although we use a hyperelastic material in our optimisation process, we first apply an isotropic elastic material, and attribute E and ν their initial values.

Table 3.4: Loops in the preprocessing of the algorithm: (a)Exemplary arrangement of initial value combination of material parameters for five combinations; (b) Exemplary model creation for two load cases E11 and G12 in combination with three load parameter sets.

(a)

Material Parameter	Combination				
	1	**2**	**3**	**4**	**5**
E	E_1	E_2	E_3	E_4	E_5
ν	ν_1	ν_2	ν_3	ν_4	ν_5
σ_0	σ_{0_1}	σ_{0_2}	σ_{0_3}	σ_{0_4}	σ_{0_5}
α	α_1	α_2	α_3	α_4	α_5
β	β_1	β_2	β_3	β_4	β_5
γ	γ_1	γ_2	γ_3	γ_4	γ_5

(b)

Model	Load case	Load parameters
Model 0	E11	Data set 1
Model 1	E11	Data set 2
Model 2	E11	Data set 3
Model 3	G12	Data set 1
Model 4	G12	Data set 2
Model 5	G12	Data set 3

The elastic material is necessary because of the usage of EasyPBC in the phase *Create job*. We start EasyPBC, which creates a job with the load case prescribe in the input. As discussed in Subsection 2.3.1, we use EasyPBC for the automatic construction of PBC. Aside from that, we adopt the generated load application corresponding to the load case.

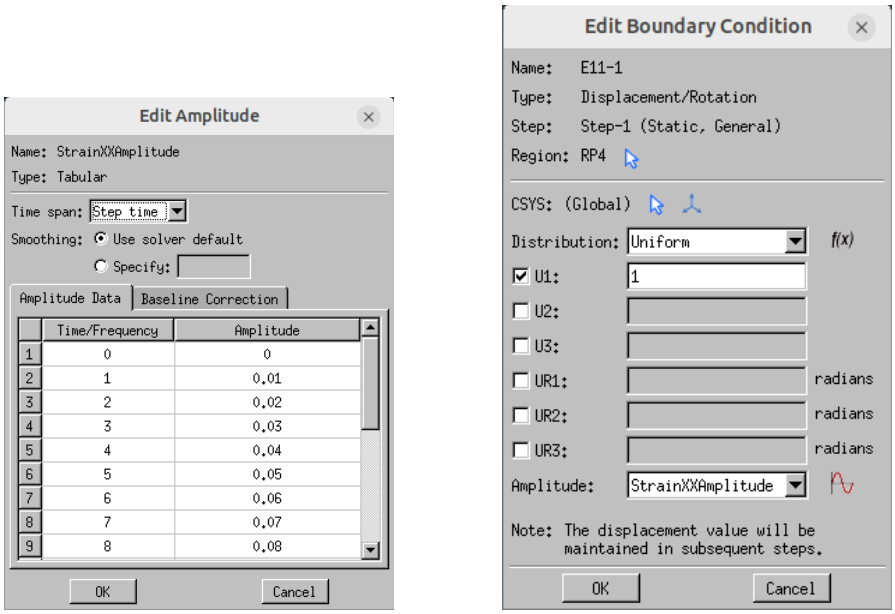
However, the settings from EasyPBC contain some default values, we adjust in the phase *Modify properties*. EasyPBC applies for every load case a uniform displacement with a constant default value. We need to adopt this value, since we want to apply the loading parameters from the MD simulations, which are not constant. In Abaqus, we can solve this issue by creating an *amplitude* to apply the load gradually (see Figure 3.3 (a)). Therefore, we enter the load parameters as an amplitude at certain time steps. For the time steps we use sequential numbering. We use these time steps later to adapt the evaluation points of the optimised load reactions. The value in the boundary condition editor is then set to one because this defines the factor by which the amplitude is multiplied (see Figure 3.3b). As a result, the load is applied step-wise with step sizes defined through the amplitude values, i.e. the load parameters. In the Abaqus environment, these steps are referenced as *load steps*.

We use the time steps, created in the amplitude menu as request points for the optimised load reactions. Thus, at every time step, the optimised load reactions caused by the corresponding load step are written. Through the transfer of the load steps from the reference data, the reference and the optimised load reactions are written at equivalent evaluation points in the loading process.

Afterwards, we modify the increment settings. EasyPBC automatically creates increments with fixed size and without non-linear geometry effects. In order to avoid convergence errors, we use automatic incrementation. Especially in the first load steps, we observe large deformations. If we try to resolve such large deformations in one incrementation step, Abaqus runs into convergence errors. With automatic incrementation, Abaqus can dynamically adapt the number of increments per load step depending on the current deformation. The non-linear geometry effects must be considered for the same reason.

In the last phase of the preprocessing we *store* the model in a dictionary. We use this dictionary later to call the models for the optimisation. We perform the preprocessing for all prescribed load cases which is highlighted in the *load case loop* in Figure 3.4.

Furthermore, the *load parameter loop* in Figure 3.4 visualises the loop over the processed load parameter sets. This leads to individual models for every load case and every load parameter set, which is exemplary listed for two load cases and three load parameter sets in Table 3.4 (b).



(a)

(b) Boundary condition menu in Abaqus.

Figure 3.3: Abaqus menus: (a) Amplitude menu to create time-dependent amplitudes for the deformation; (b) Boundary condition menu to apply the created amplitude in the requested direction.

3.4 Optimisation process

In the following section, we describe the optimisation process visualised in the *optimisation loop* in Figure 3.4. We start the process by calling the `scipy.minimize`-function.

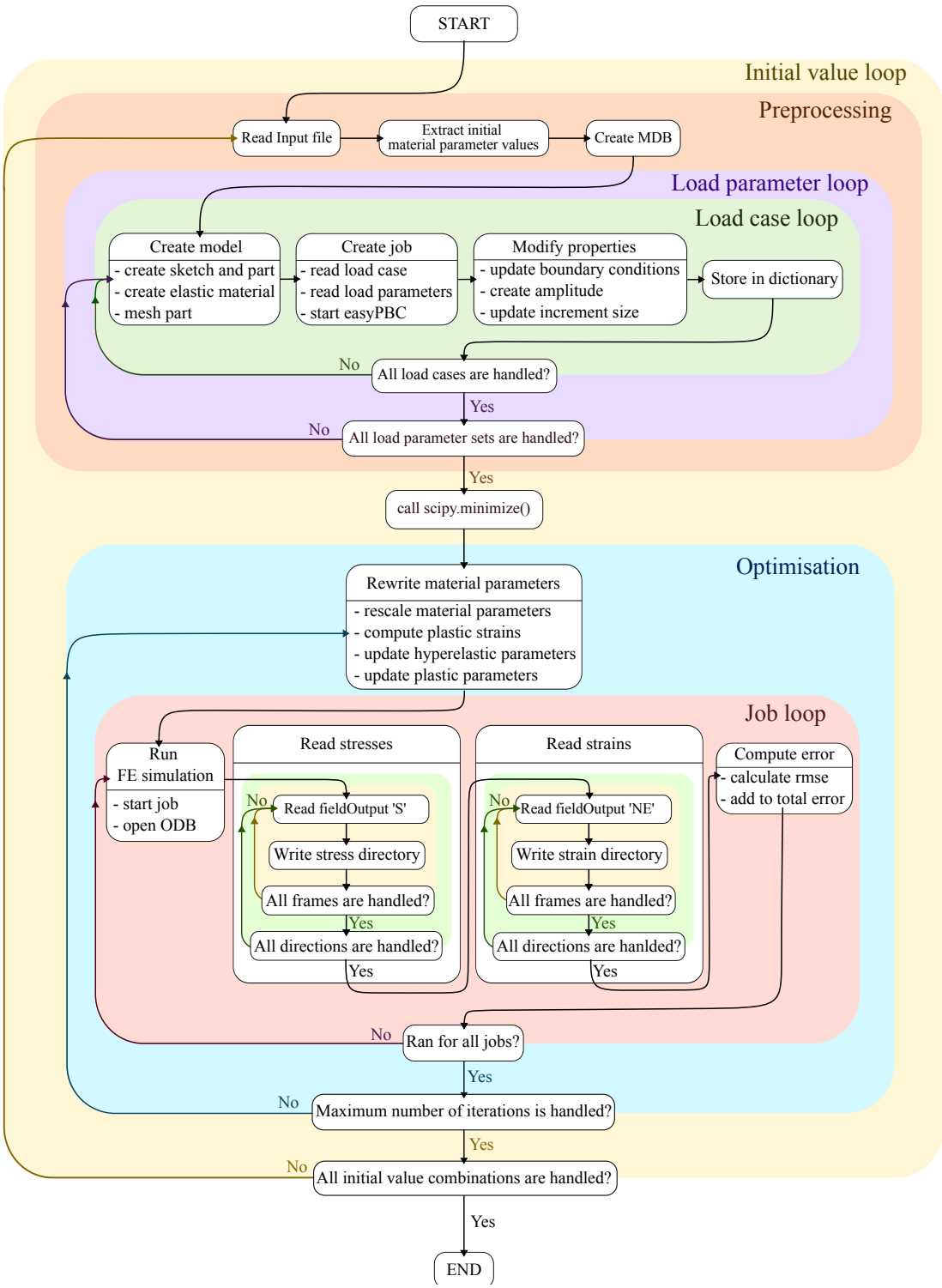


Figure 3.4: Flowchart code.

Table 3.5: List of general input parameters for the scipy.minimize-function with the contents passed in the function-call.

Parameter	Content	Data format	Explanation
Objective function	Optimisation function	–	Function whose scalar value should be minimised
Initial guess	Material parameters	array	Scaled initial values for the optimisation parameters
Additional arguments	Cube parameters	object	Model information from input file
	Load parameters	dictionary	Load parameters from MD-simulations
	Work directory	string	Path to store results
	Evaluation counter	scalar	Counter for the performed function evaluations
method	Nelder-Mead	–	Numerical algorithm
bounds	Limits for material parameter	array	Upper and lower boundary values for every optimisation parameter
maxiter	Number of iterations	scalar	Maximum number of iterations as termination criterion

We pass this function various parameters, listed in Table 3.5. The `scipy.minimize`-function calls our self-written optimisation function, where the evaluation takes place. The *initial guess* stores an array with inital values for all parameters that should be optimised. Additionally, we pass information about the models that we created in the preprocessing and the load parameters. We use all models created in the preprocessing for one optimisation call.

We start the process with the phase *Rewrite material parameters* for all models. Since they all describe the same material, we write the same values for every model. For a better performance of the Nelder-Mead algorithm, the optimisation parameters are scaled in the bounds from zero to one. To rewrite the values in the models, we have to rescale them first. Then, we can use the rescaled parameters to compute the values for the hardening-function with the formula for VOCE-hardening (Equation 2.1). We remove the elastic material and substitute it with a hyperelastic material which is suitable for high non-linear deformations. To achieve this, we convert the Young's modulus and the Poisson's ratio into the hyperelastic parameters C_{10} and D_1 via the relations

$$C_{10} = \frac{E}{4(1+\nu)} \qquad D_1 = \frac{6(1-2\nu)}{E}. \tag{3.5}$$

Now we can update the material parameters in all models. In the next phases, we handle the models successively. We start the job of the first model to perform the Abaqus analysis and open the resulting *output database (ODB)* in the *Run FE-simulation* phase. In the ODB all simulation results are stored.

With the phase *Read stresses* the evaluation begins. First, we read the *FieldOutput* variable 'S' and write the data in a directory. In the FieldOutput, Abaqus stores the

values of various quantities, e.g. stresses, strains and energy components, during the simulation. The variable 'S' contains the stress components in all directions, i.e. the stress components of the optimised load reactions. We read out every *frame* from the ODB, since one frame corresponds with one load step. Additionally, we loop over all directions (xx, yy, zz, xy, yz, xz).

We employ the same for the strain values in the phase *Read strains*. Here it is important to read out the correct strain variable 'NE' (nominal strain). For hyperelastic materials, ABAQUS uses the logarithmic strain ('LE') as standard value. Because of its logarithmic scaling, we cannot compare them to the reference data. We store all values for all frames and directions in a dictionary, similar to the stress values. Consequently, all optimised load reactions are stored, and the phase *Compute error* can be executed.

In this step, the error is computed as described in Section 3.2. For a better structure of the algorithm this part is captured in a separate function. We call this function, and pass the optimised load reactions and load parameters. The function computes the RMSE according to Equation 3.3 and returns it. Multiplied with its corresponding weights for the load case and the load parameters, we add this value to the total error according to Equation 2.5. Now we restart the *job loop* in Figure 3.4 by starting the FE simulation for the next job. Once all the jobs are processed, and we compute the total error value, we return it to the `scipy.minize()` function. The internal Nelder-Mead algorithm reduces the error and returns the corresponding material parameters. This completes one optimisation iteration. In Figure 3.4, the *iteration loop* is visualised with the blue box. This process will run until the defined number of maximum iterations is exceeded, or convergence is reached. When the changes in the total error are very small, the internal convergence criterion is reached and the algorithm stops.

3.5 Data storage

To evaluate the optimisation process at the end, parameter values need to be recorded during the preprocessing and the optimisation. This ensures that the values of all iterations are saved, since the variable values are rewritten in every new iteration. In Table 3.6 an overview about the stored parameters is given. In the preprocessing, we extract the current initial value combination from the input file.

All files created from ABAQUS like the CAE, ODB and the job-files are stored in a subfolder. In the optimisation part, multiple functions are called after the phase *Compute error* to store the current variable values. The material parameters for every iteration are stored. In addition, we store multiple interim results during the error calculation. For every job the weighted MSEs are stored for every iteration, so that the impact and evolution of every evaluated reaction can be understood. To track the impact of the applied weights, all RMSE values are stored individually, and weighted. In addition, we store the total error, which is the sum of all weighted RMSEs of one iteration. Finally, we have one file each for the material parameters, MSEs, RMSEs, and the total errors. All these files are handled in the same way. After the computation of the total error, the variable values of the current iteration are stored. The stress and strain components are stored in separate files for each iteration. For one iteration, a file is created which contains the stress and strain dictionaries from the steps *Read stress* and *Read strain*. This leads to as many files as iterations are performed for every job. At the end of the optimisation process, we store the final message returned from the `scipy.minimize()` function. PYTHON automatically sends a message where the cause of termination is stated. This message becomes important if the algorithm stops before the maximum

number of iterations is reached. Then we can reproduce whether this happens because the convergence limit is reached or a numerical error occurred. The described procedure to store the data is done for every initial value combination separately.

Table 3.6: Stored parameters during the preprocessing and the optimisation during one iteration with data format.

Stored parameters	**Data format**	**Explanation**
Input parameters	JSON	Copy of the input file with current initial value combination
Material parameters	CSV	Table with all material parameters for each evaluation
weighted MSE	CSV	File per job with weighted MSE per evaluation
RMSE	CSV	File with RMSE per job, each stored alone, with weight for load case, and weight for load parameters
Total error	CSV	Total error after each evaluation
Optimised load reactions	CSV	File per job with all optimised load reactions in one evaluation
`scipy.minimize()` message	TXT	Final message about the termination status of the scipy-function

4 Results

In this chapter the results from the test cases listed in Table 3.2 will be presented. First, we discuss the verification results to understand the capabilities and issues of the optimisation approach. In the next step, we study the validation tests with different load parameter sets. Then, we focus on tensile and shear tests with non-linear loading conditions. Finally, we present the results of the cyclic load parameters. All studies were performed with five different initial value combinations to ensure reproducibility. In the following plots, they are numbered sequentially.

4.1 Verification

In this section, we present the results of the verification study. We tested a material with mixing ratio 6:3 under tensile strain in xx-direction. We applied a linear strain up to a maximum value of 20%. As evaluated reactions we used σ_{xx}, ε_{yy} and ε_{zz}. To reconstruct the optimisation history, the trends of selected quantities are presented in the following.

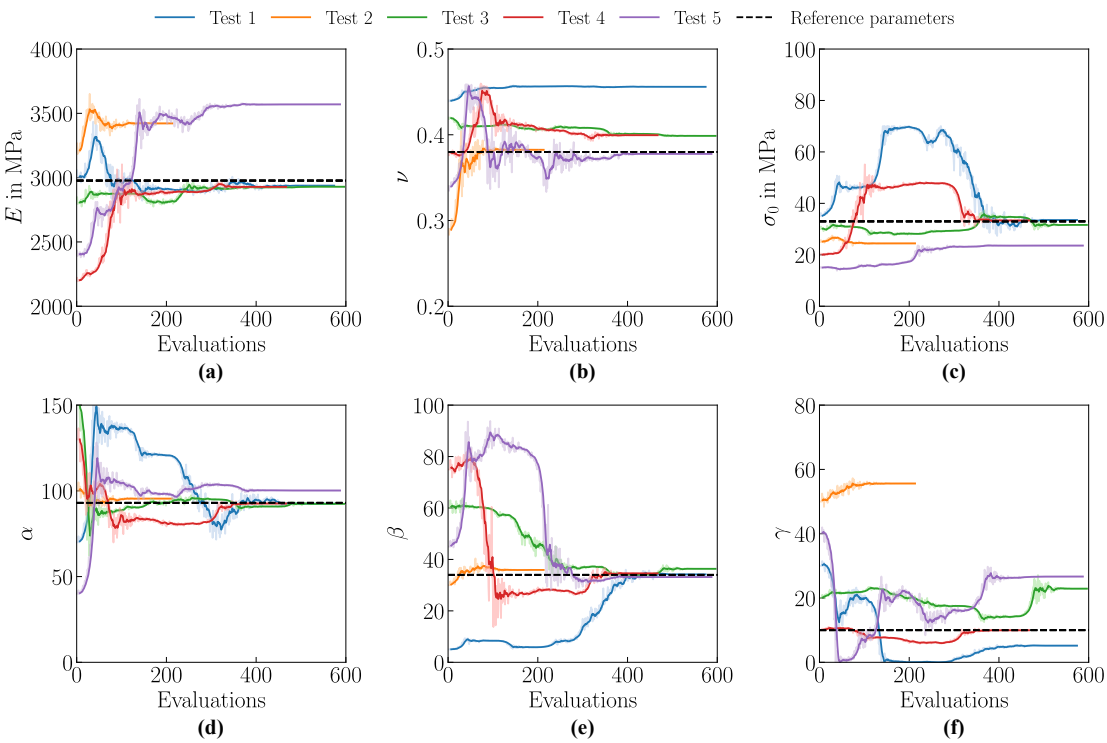


Figure 4.1: Evolution of the optimised material parameters: (a) Young's Modulus E; (b) Poisson's Ratio ν; (c) yield stress σ_0; hardening coefficients (d) α; (e) β; (f) γ; over the optimisation evaluations for material with mixing ratio 6:3 under linear tensile strain; with respective reference values obtained by Ries et al. [0].

First, the material parameters are presented, since their optimisation is the main purpose of the procedure. The progress of each material parameter during the optimisation

is plotted in Figure 4.1. Figure 4.1 (a) and (b) show the elastic parameters Young's modulus and Poisson's ratio. As explained in Section 3.4, in the FE model, C_{10} and D_1 were used as elastic parameters. However, since E and ν are the more illustrative quantities, only these are represented. With Equation 3.5 the parameters can easily be converted into each other. Figure 4.1 (c)-(f) depict the yield stress and the hardening coefficients α, β and γ, which define the plastic hardening. The varying number of completed evaluations for the tests is caused by the internal convergence criterion of the Nelder-Mead algorithm, as stated in Subsection 2.4.1. If this criterion is met, the algorithm stops, even if the maximum number of iterations is not reached.

It should be stated, that in all performed tests, the values of all parameters converge. To verify the quality of the optimised material parameters, we compare them with the material parameters determined by Ries et al. [0], which are added as black lines in the plots. For E and σ_0 (Figure 4.1 (a) and (c)) tests 1, 3 and 4 are similar to the reference values, whereas test 2 and 5 results in much higher values. In Figure 4.1 (b) the distribution is contrary, such that test 2 and 5 match the reference value, and tests 1, 3, and 4 lead to higher values for the Poisson's ratio. For α and β all tests lead to values similar to the reference value. In Figure 4.1 (f) the evolution of γ is depicted. For all tests the parameters converges to another value. Only in test 4 the optimised value corresponds to the reference value. In general, the trend of the yield stress, α and β ends in all tests to optimisation results similar to the corresponding reference values. For the elastic parameters, we observe contrary behaviour within all tests, where either the reference value for E matches or the reference value for ν. To verify the quality of the optimised material parameters, the load reactions are considered.

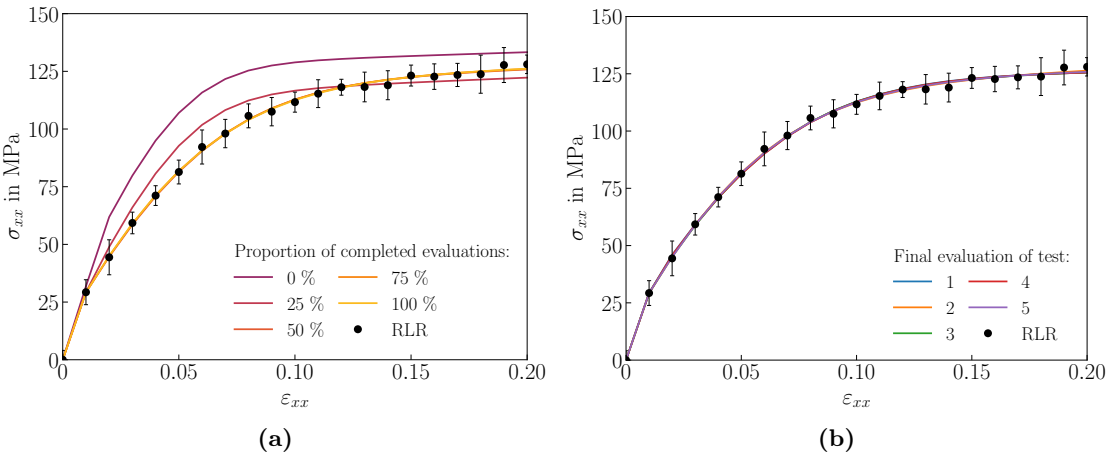


Figure 4.2: Optimised and reference load reactions (RLR) σ_{xx} with standard deviations for material with mixing ratio 6:3 under linear tensile strain ε_{xx}: (a) progress of optimised σ_{xx} during the optimisation for an exemplary test; (b) final optimised σ_{xx} of all tests.

In Figure 4.2, the load reactions σ_{xx} are plotted against the applied strain ε_{xx}. The progress of the values during an exemplary test is shown in Figure 4.2 (a). The reference load reactions are plotted as black dots with corresponding standard deviations. The optimised load reactions match the reference load reactions almost perfect already after 50% of the performed evaluations. The standard deviations are much higher than the difference between the reference points and the optimised load reactions. In Figure 4.2 (b) the optimised load reactions after the final evaluation of each test are plotted. The curves of the optimised load reactions match the reference load reactions during the whole

loading procedure. However, ε_{yy} and ε_{zz} were used as evaluated reaction as well. Because of the isotropic material behaviour, their load reactions are equal in size. Therefore, only one load reaction (ε_{yy}) is plotted in Figure 4.3b (b) after the final evaluation. Similar, as for the stress load reactions, a high correlation between the optimised load reactions and the reference load reactions can be seen. The observations of all evaluated reaction suggest that in all tests the RMSE was effectively reduced. To support this assumption the progress of the RMSE for all the tests in Figure 4.3 (a). We observe that in all tests the RMSE is reduced up to a limit value between one and two. The RMSE decreases quite fast in the first optimisation iterations, and then approaches slowly to its minimum value.

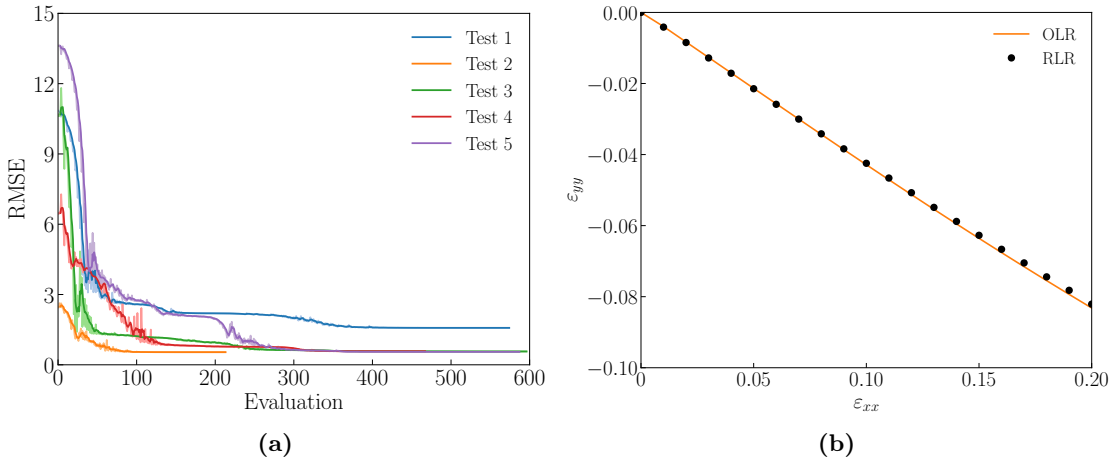


Figure 4.3: Optimisation results for material with mixing ratio 6:3 under linear tensile strain: (a) progress of root mean squared error (RMSE) during the optimisation for all tests; (b) final optimised load reactions and reference load reactions (RLR) ε_{yy} over applied linear tensile strain ε_{xx} for an exemplary test.

Discussion The results of the optimised load reactions presented in Figure 4.2 and Figure 4.3 indicate an effective error reduction of the algorithm for all tests. Independent of the initial value combination, equally high correlation levels are reached. Regarding the plastic parameters, in all tests the yield stress, α and β are determined similar to their reference values, whereas γ results in different values for every test. The variety in the solutions for E and ν indicates difficulties in the modelling of the elastic behaviour. These results illustrate, that multiple optimised material parameters yield equal load reactions. To understand this phenomenon, the impact of the material parameters on the load reactions is analysed in detail. As described in Subsection 3.1.2, most load steps are placed in the plastic domain of the material. Only the first load step is located in the elastic regime. Improving this single data point is not worthwhile because it has a very small influence on the total error. However, these issues should be reduced previously through an additional weight on the first data point (see Section 3.2). Nevertheless, issues with the modelling of the elastic behaviour occur.

To understand the variety in the solutions of γ, the impact of each hardening parameter will be investigated in detail. Therefore, the focus is taken on the plastic parameters. Via the VOCE-hardening (Equation 2.1) the stress load reaction σ_{xx} can be computed as a function of the plastic material parameters. In Figure 4.4 an exemplary trend of the VOCE-function is plotted with arbitrary chosen values for the plastic parameters. The impact of each parameter on the curve is mapped by adjusting its value sequentially.

The yield stress σ_0 stays constant, since it only acts as an offset value, which influences the point at which the material starts to plastify. The parameter α has the greatest impact on the shape of the curve, while a variety of 50% in the parameter γ has hardly any visible effect. The small influence of γ explains its high variance in Figure 4.1. In general, the plot indicates a high flexibility in adjusting the shape of the curve through different value combinations. This consideration verifies the assumption, that various material parameter combinations lead to similar load reactions.

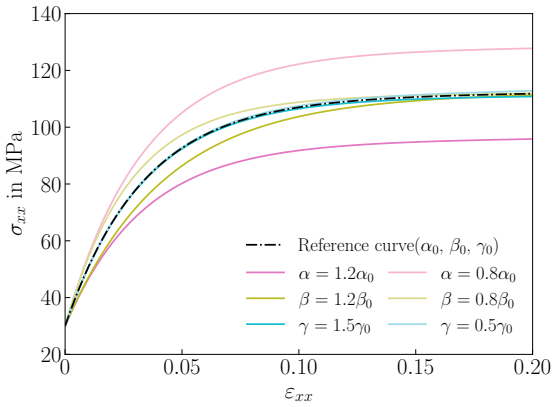


Figure 4.4: Exemplary trend of VOCE-hardening function according to Equation 2.1 with reference parameters α_0, β_0 and γ_0; and visualisation of parameter influence on the curve trend through successive parameter adaptations.

The presented results show, that the optimisation algorithm works in general. It is able to reduce the error of the load reactions within an adequate number of function evaluations. However, the approach is unable to find unique material parameters. To improve this, the solution space is decreased. As already stated, the elastic parameters affect only the first load step. Thus, it is possible to compute their values directly via

$$E = \frac{\Delta\sigma_{xx_1}}{\Delta\varepsilon_{xx_1}} \qquad \nu = \frac{\Delta\varepsilon_{yy_1}}{\Delta\varepsilon_{xx_1}} \tag{4.1}$$

Then, only the plastic material parameters need to be optimised. By adapting the algorithm in this way, the optimisation should lead to unique values for the remaining material parameters. In the validation study this configuration is tested for material with mixing ratios 4:3, 6:3 and 8:3.

4.2 Validation

In the validation study we performed tests under the same loading conditions as in the verification study. We loaded the material in load case E11 with a linear tensile strain up to 20%. The tests were performed with materials of three different mixing ratios 4.3, 6:3 and 8:3. For all materials, we analysed five different combinations of initial values. Since the results of the verification study indicate issues in the identification of the elastic parameters, we predefined E and ν in the validation study.

Test series 6:3 In the first test series, the same material as in Section 4.1, with mixing ratio 6:3, is used. According to Equation 4.1 the elastic parameters are determined to

$$E = 2916 \text{ MPa} \qquad \nu = 0.41$$

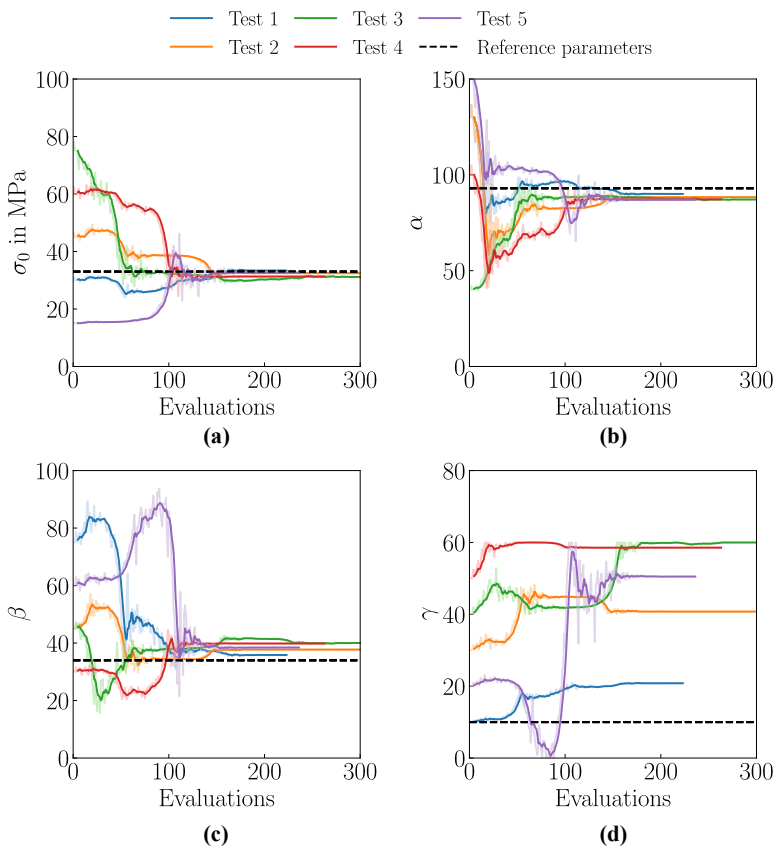
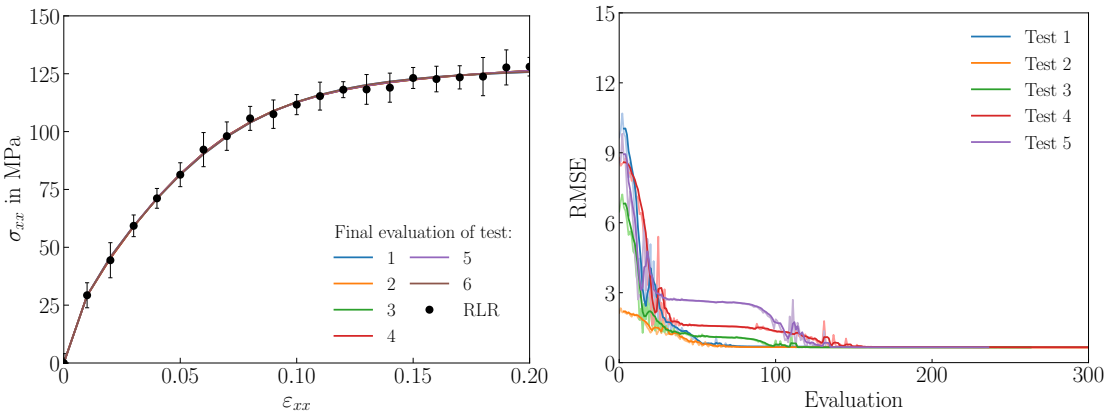


Figure 4.5: Evolution of the optimised material parameters: (a) yield stress σ_0; hardening coefficients (b) α; (c) β; (d) γ; over the optimisation evaluations for material with mixing ratio 6:3 under linear tensile strain with respective reference values obtained by Ries et al. [0] and predefined elastic parameters Young's modulus E and Poisson's ratio ν.

The optimised plastic material parameters are shown in Figure 4.5. Here the numbers of evaluations are reduced compared to the verification study. Because of the reduced number of optimisation parameters, less function evaluations are required until a solution is found. Equal to Figure 4.1, the corresponding reference data from Ries et al. [0] are depicted in the plots. In all tests, the values of σ_0, α and β demonstrate a converging trend towards the reference value (see Figure 4.5) (a)-(c). For γ the converged values

vary for each individual test, which can be seen in Figure 4.5 (d). All performed tests lead to in γ values higher than the reference value. Similar to the verification study, the quality of the optimised material parameters is verified by the reference load reactions. In Figure 4.6 (a) the final optimised load reactions σ_{xx} for all tests are plotted. The curves of all tests show a perfect match of the reference load reactions. The results of the evaluated strain reactions ε_{yy} and ε_{zz} are added in Appendix A, since their presentation would be beyond the scope of this work. It can be stated, that in all tests the optimised load reactions match the reference load reactions of the lateral strains. Their influence on the optimisation is implicitly included in the RMSE, which is depicted in Figure 4.6 (b). Similar to the verification study, the RMSE decreases rapidly in the beginning, and then holds a minimum value.



(a) **(b)**

Figure 4.6: Optimisation results for material with mixing ratio 6:3 under linear tensile strain with predefined elastic parameters Young's modulus E and Poisson's ratio ν: (a) final optimised load reactions σ_{xx} of all tests with corresponding reference load reactions (RLR) with standard deviations; (b) progress of root mean squared error (RMSE) during the optimisation for all tests.

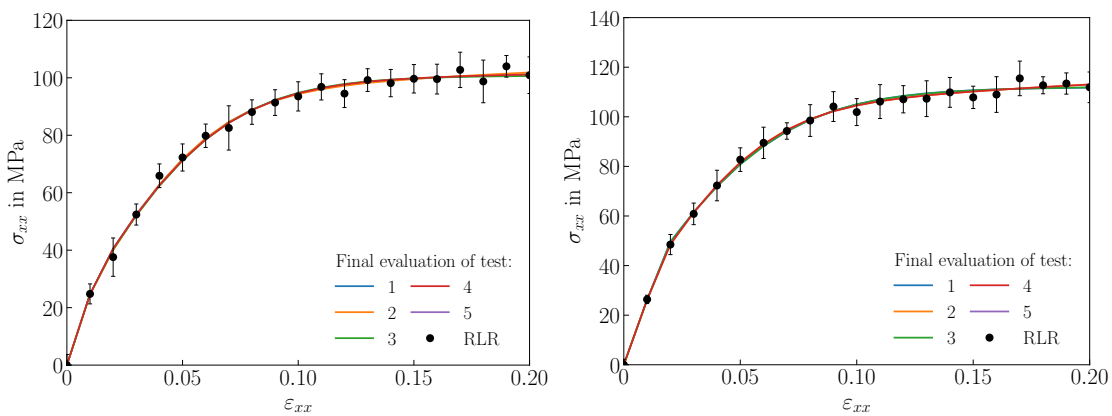
Test series 4:3 and 8:3 Next, the results from the validation studies for materials with mixing ratio 4:3 and 8:3 are presented. The results of the two test series are discussed together, since they lead to similar conclusions. In addition, the discussions of the optimised material parameters and the RMSE are reduced on the final values. Since these two validation series focus on the improvement of the final values of optimised material parameters, no relevant information is neglected through this diminution. The corresponding evolution plots are added in Appendix A, where the plastic material parameters and the RMSE show the same trends for both mixing ratios as for the mixing ratio 6:3. The final values of the plastic material parameters and the RMSEs are summarised in Table 4.1 and Table 4.2. The fixed elastic parameters, and the reference values are also included in the table. For both mixing ratios, the values for the yield stress, α and β were determined within a small range of variations for all tests. All these determined values are close to their respective reference value. Only γ shows high variations between each test. For both mixing ratios, no test matches the reference value for γ. The RMSE reaches in all tests an equally low value. Finally, the optimised load reactions σ_{xx} are evaluated in Figure 4.7. They represent a high correlation of all tests with their corresponding reference load reactions.

Table 4.1: Final values for the optimised material parameters yield stress σ_0, and hardening coefficients α, β and γ for material with mixing ratio 4:3 under linear tensile strain with predefined Young's modulus E and Poisson's ratio ν, and respective reference values obtained by Ries et al. [0].

Test	E (MPa)	ν	σ_0 (MPa)	α	β	γ	RMSE
1	2478	0.43	28.48	66.97	50.54	45.74	0.68
2	2478	0.43	30.60	68.21	44.95	17.99	0.70
3	2478	0.43	30.89	68.40	44.31	13.80	0.70
4	2478	0.43	31.37	68.01	43.97	13.25	0.70
5	2478	0.43	30.03	67.99	46.39	23.78	0.69
Reference values	2552	0.40	29.74	71.44	41.70	3.97	–

Table 4.2: Final values of the optimised material parameters yield stress σ_0, and hardening coefficients α, β and γ for material with mixing ratio 8:3 under linear tensile strain with predefined Young's modulus E and Poisson's ratio ν and respective reference values obtained by Ries et al. [0].

Test	E (MPa)	ν	σ_0 (MPa)	α	β	γ	RMSE
1	2636	0.42	44.45	60.27	51.33	60.00	0.58
2	2636	0.42	46.39	63.09	43.52	20.25	0.61
3	2636	0.42	46.69	64.12	41.94	9.06	0.62
4	2636	0.42	44.45	60.26	51.32	60.00	0.58
5	2636	0.42	45.97	61.62	46.00	35.83	0.60
Reference values	2702	0.39	43.66	64.81	45.69	29.48	–



(a) **(b)**

Figure 4.7: Final optimised load reactions σ_{xx} with corresponding reference load reactions (RLR) with standard deviations under linear tensile strain: (a) for material with mixing ratio 4:3; (b) material with mixing ratio 8:3.

Discussion The performed validation studies demonstrate an improved convergence behaviour of the optimisation approach. For all mixing ratios, the optimised values for σ_0, α and β show only small deviations to the corresponding reference value, independent of the initial value combination. The value of γ still varies in a wide range, and has no correlation with the reference value. However, the impact of γ on the trend of the hardening curve is relatively small as demonstrated in Figure 4.4. Therefore, its unique definition is challenging. The RMSE and the stress load reactions ensure a high correlation to the reference load reactions. These results improve the reliability of the developed optimisation approach. Overall, it can be stated that for a single linear applied load case with fixed elastic parameters, the algorithm is able to find plastic parameters within a small range of variations which appropriately match the material behaviour.

4.3 Tensile-Shear combination

In this section the optimisation results for sinusoidal load application are presented. Similar to the validation studies, we calculated the elastic parameters previously with the first stress-strain data. First, a tensile strain was applied with the load case E11. In the second test series the same load parameters were adapted as shear load case G12, and finally, the load cases were combined in a single optimisation. In all test series, material with the mixing ratio 6:3 was simulated. The load parameters follow the first quarter of a sinus function up to a maximum amplitude of 15%.

4.3.1 Pure tensile tests

In the first test series, a single tensile load case was applied. Similar as in the last validation studies, we focus on the final results of the optimisation approach. Therefore, the material parameters from the final evaluation are summarised in Table 4.3. All tests lead to similar values for the yield stress, α and β. The values of γ show high variances.

Table 4.3: Final values for the optimised material parameters yield stress σ_0, and hardening coefficients α, β and γ for material with mixing ratio 6:3 under sinusoidal tensile strain with predefined Young's modulus E and Poisson's ratio ν.

Test	**E (MPa)**	$\boldsymbol{\nu}$	$\boldsymbol{\sigma_0}$ **(MPa)**	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	$\boldsymbol{\gamma}$
202	2599	0.42	45.50	78.14	31.23	55.46
283	2599	0.42	46.37	84.24	28.15	0.00
365	2599	0.42	45.47	77.64	31.49	59.95
132	2599	0.42	46.61	82.87	28.31	10.97
183	2599	0.42	46.07	81.84	29.18	22.11

To classify the results, the load reactions σ_{xx} and the RMSE value are plotted in Figure 4.8. The optimised load reactions show a high correlation to the reference load reactions (see Figure 4.8 (a)). We have deliberately omitted the standard deviations in this plot, as this would otherwise reduce clarity. In Figure 4.8 (b), the RMSE starts at quite high values, compared to the ones in the validation studies. However, the value decreases fast after the first evaluations, and convergences to a limit value around two.

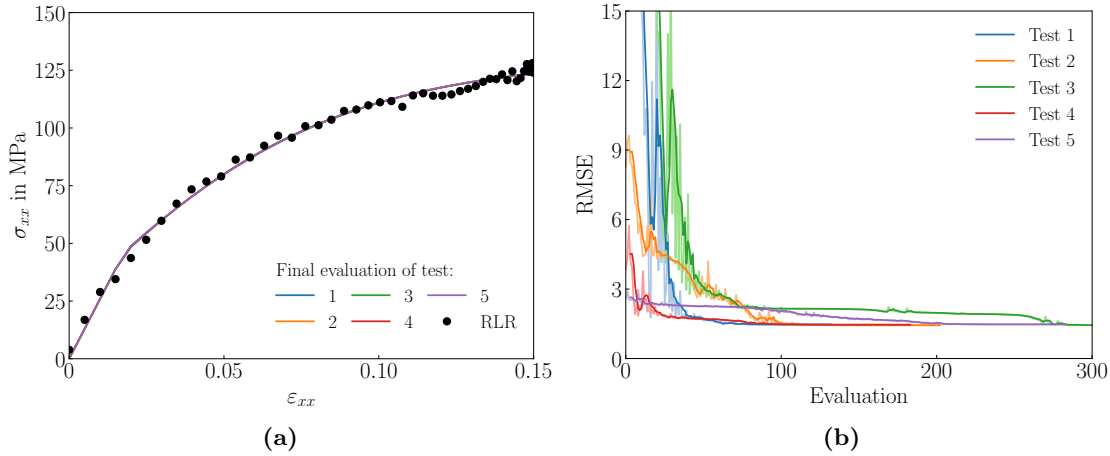


Figure 4.8: Optimisation results for material with mixing ratio 6:3 under sinusoidal tensile strain with predefined elastic parameters Young's modulus E and Poisson's ratio ν: (a) final optimised load reactions σ_{xx} of all tests with corresponding reference load reactions (RLR); (b) progress of root mean squared error (RMSE) during the optimisation for all tests.

4.3.2 Simple shear tests

In the next test series, a single shear strain was applied. Here, the computation of the elastic parameters, needs to be adapted. Since the Young's modulus cannot be computed directly. From the shear stress, we first calculate the shear modulus G with the following formula

$$G = \frac{\Delta\sigma_{xy}}{2\Delta\varepsilon_{xy}} = 1619 \text{ MPa} \tag{4.2}$$

which can be transferred into E through

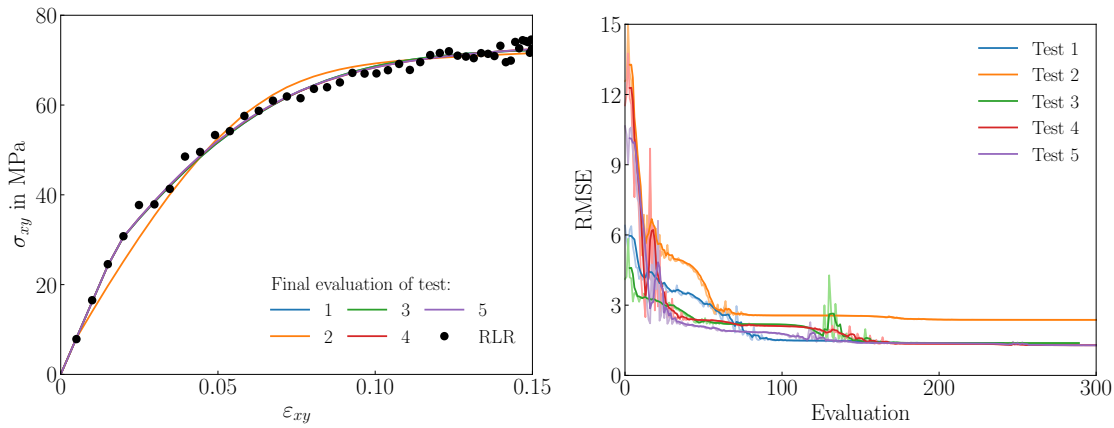
$$E = 2G(1+\nu) \tag{4.3}$$

We chose the value of ν in a way, that the resulting Young's modulus is below 4500 MPa, which was the selected upper limit value in the previous tests. The final values for all material parameters are listed in Table 4.4. In this study, the value of σ_0 met the limits in every test, since the upper limit is 50 MPa and the lower limit 15 MPa. For γ the limits of 0 and 100 are reached as well. For α and β similar values are reached for all tests except test 2.

Table 4.4: Final values for the optimised material parameters yield stress σ_0, and hardening coefficients α, β and γ for material with mixing ratio 6:3 under sinusoidal shear strain with predefined Young's modulus E and Poisson's ratio ν.

Test	**E (MPa)**	$\boldsymbol{\nu}$	$\boldsymbol{\sigma_0}$ **(MPa)**	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	$\boldsymbol{\gamma}$
1	4402	0.36	50.00	76.72	69.66	0.00
2	4402	0.36	15.00	103.41	128.88	100.00
3	4402	0.36	50.00	76.71	69.79	0.00
4	4402	0.36	50.00	70.51	78.81	100.00
5	4402	0.36	50.00	70.69	77.88	98.32

In Figure 4.9 (a) the final results of the optimised load reactions σ_{xx} with the corresponding reference data are plotted. All tests lead to appropriate matches of the load reactions except test 2. The trend for test number 2 differs from the other curves, but still shows a high correlation with the reference load reactions. The RMSE trend looks similar as for the tensile tests with test 2 converging at a slightly higher limit value, depicted in Figure 4.9 (b).



(a) **(b)**

Figure 4.9: Optimisation results for material with mixing ratio 6:3 under sinusoidal shear strain with predefined elastic parameters Young's modulus E and Poisson's ratio ν: (a) final optimised load reactions σ_{xx} of all tests with corresponding reference load reactions (RLR); (b) progress of root mean squared error (RMSE) during the optimisation for all tests.

4.3.3 Combined tensile-shear tests

In this test series, the previously presented load cases were combined. Similar, we fixed the elastic parameters before we started the optimisation. However, the defined values for E and ν must represent the elastic behaviour for both load cases. A comparison of the values we chose in the separate tests, shows a high variation. As a first choice, we used the values of E and ν from the simple shear tests. The plastic parameters, optimised in this test series are summarised in Table 4.5. The yield stress met its upper limit of 50 MPa in every test. For α and β the algorithm found similar values for all initial value combinations. γ met its lower limit of zero in four of five tests.

Table 4.5: Final values for the optimised material parameters yield stress σ_0, and hardening coefficients α, β and γ for material with mixing ratio 6:3 under sinusoidal combined loading of shear and tensile strain with predefined Young's modulus E and Poisson's ratio ν.

Test	**E (MPa)**	$\boldsymbol{\nu}$	$\boldsymbol{\sigma_0}$ **(MPa)**	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	$\boldsymbol{\gamma}$
1	4402	0.36	49.99	74.20	64.57	9.22
2	4402	0.36	49.99	74.95	63.42	0.00
3	4402	0.36	49.82	75.12	63.59	0.00
4	4402	0.36	49.99	74.94	63.56	0.00
5	4402	0.36	49.99	74.94	63.50	0.00

Figure 4.10 depicts the optimised load reactions σ_{xx} for both load cases. For the tensile load case Figure 4.10 (a) shows high deviations between the reference load reactions and the optimised load reactions. Only the first and the last point of the reference data are matched by the optimisation tests. In between, the optimised stresses are consequently higher than the reference data. In contrast, the optimised shear stresses in Figure 4.10 (b) show a high correlation to the reference stresses. The optimised shear stresses are slightly lower than the reference data. Figure 4.10 (c) shows the total error of this test series. Qualitatively, the progress of the error is similar to the ones shown before. However, the absolute value of the error is around twelve, which is significantly higher than in all other studies. We observe this behaviour in all tests within this series.

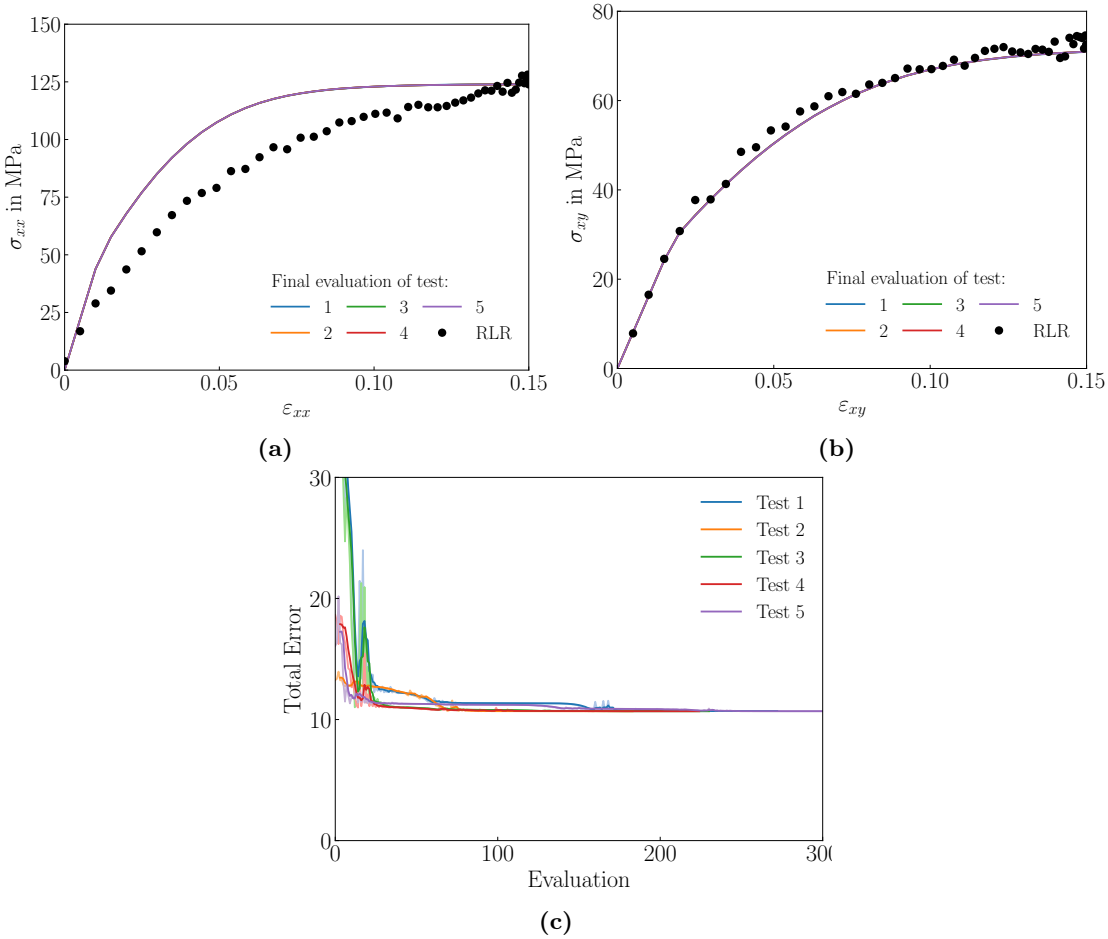


Figure 4.10: Optimisation results for material with mixing ratio 6:3 under sinusoidal combined loading of shear and tensile strain with predefined Young's modulus E and Poisson's ratio ν: (a) optimised and reference load reactions (RLR) σ_{xx} caused by tensile strain; (b) optimised and reference load reactions (RLR) σ_{xy} caused by shear strain; (c) evolution of the total error during the optimisation iterations.

4.3.4 Discussion

The previously presented results expose some properties of the optimisation approach, which need to be discussed. The tensile load case showed similar behaviour as the validation tests. The material parameters σ_0, α and β were determined uniquely, which lead to adequate stress-strain curves. In contrast to the RMSE of the validation study, the

RMSE in this study converges at a higher value. This could be explained through the different reference data, used for the studies. In the sinusoidal data set much more data points are included (see Figure 4.8a). Therefore, more deviations are possible, which leads to a higher absolute error. Especially, at the end of the loading process, the data become volatile, which makes it impossible to match them with a continuous function. However, the results show, that for a sinusoidal loading, applied as load case E11, the optimisation approach gives adequate results.
The tests for the simple shear load case show similar results in the load reactions. The optimised load reactions match the reference data as good as possible through the distribution of data. However, test 2 behaves like an outlier. Looking at the material parameters in Table 4.4, this behaviour becomes understandable. There, test 2 meets the lower limit of the yield stress, and as a consequence, for α and β different values were determined too. However, in all tests σ_0 meets one of its limits, which is a behaviour we never saw before in one of the tensile load cases. Therefore, a connection to the load case of shear load, seems obvious. A possible issue could be the information content of a singular shear test. As evaluated reaction, we only used the corresponding shear stress σ_{xy}. If these data contain less information, a unique definition of the material parameters is not possible, which is similar to the results of the verification tests in Section 4.1. The value of the yield stress determines the stress value, where the plastification starts. Difficulties with the determination of this value, could indicate incorrect interpretation of the limit between the elastic and the plastic regime. These issues could occur from the distribution of the reference data in the domains. The density of the reference data gets higher with increasing load. Therefore, at the transition from elastic to plastic behaviour, only few points are given, which restrains a clear separation. In addition, for the definition of the elastic parameters multiple combinations were possibly as described in Subsection 4.3.3. The influence of different elastic parameters needs additional investigations in some future works, since their impact on the optimisation of the yield stress is not known so far.

Another fact, that could explain difficulties with the shear load case, are the reference data itself. The relaxation procedure used in the MD simulations to eliminate the viscous parts of the stress response, was applied only for tensile load applications before. Therefore, the accuracy of this procedure for simple shear is not verified so far.

Consequently, the optimisation of the remaining plastic parameters is affected by the behaviour of the yield stress. Since it meets its limit, the numerical algorithm has reduced options to optimise the other parameters. Therefore, the similarity of the hardening parameters is a logical consequence of the behaviour of σ_0.

In the combined tests, the issues on the shear loading show up again. The yield stress showed similar behaviour, such that its upper limit was met in every test. In addition, the elastic parameters computed from the tensile reference data were different from the ones computed from the shear reference data. This led to an insufficient fitting of the tensile load reactions. In Figure 4.10 (a) the slope of the fixed elastic curve is much higher than the reference data for the tensile load case, which leads to a higher yield stress, than the stress in the reference data. Therefore, it is impossible for the optimisation approach to fit the reference data in the further loading process. We assume the reasons for this inadequate optimisation behaviour in the shear load case. The problems mentioned before, are also valid in combination with other load cases. Therefore, a detailed study on the optimisation behaviour for shear load application is recommended in the future.

4.4 Cyclic Tests

In this section results from tests with cyclic load parameters are presented. We applied a one and a half sinus period as a tensile load case E11. To gain more information about the elastic behaviour, we combined load parameters with amplitudes of 1%, 5% and 8% in the optimisation. For this test series, the elastic parameters where part of the optimisation, because the values we computed from each data set for E and ν were different. The material parameters from the final optimisation evaluations are presented in Table 4.6. The optimised Young's Modulus were determined within a small range of deviations. For the Poisson's ratio, all tests end close to its upper limit value of 0.45. In the plastic parameters, strong deviations between each test can be detected. β met into its upper limit in three tests. In addition, the total error of each test in added in Table 4.6. Compared to the previous studies, the absolute value of the error is significantly higher.

Table 4.6: Final optimised material parameters Young's modulus E, Poisson's ratio ν, yield stress σ_0, and hardening coefficients α, β and γ for material with mixing ratio 6:3 under sinusoidal tensile strain applied in 1.5 loading cycles with amplitude 1, 5 and 8%.

Test	**E (MPa)**	$\boldsymbol{\nu}$	$\boldsymbol{\sigma_0}$ **(MPa)**	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$	$\boldsymbol{\gamma}$	**Total Error**
1	1956.17	0.44	65.46	53.36	100.00	0.16	15.99
2	1995.66	0.44	81.89	27.93	60.96	37.76	16.25
3	1981.51	0.44	71.29	48.01	100.00	3.03	15.98
4	2020.59	0.44	91.19	87.56	2.98	12.12	16.59
5	2013.95	0.44	43.40	76.39	100.00	0.00	16.29

To evaluate the quality of the material parameters, we study the optimised load reactions σ_{xx} for each amplitude in Figure 4.11 (a)-(c). For more details, the evolution of the load reactions of each amplitude are plotted separately. In addition, only one exemplary test is shown. A degradation of the match of the reference load reactions can be observed for the amplitude of 1%. For the amplitude of 5% and 8% we notice positive progress during the optimisation run. Especially, in the first rise, both load reactions match adequately with their corresponding reference data. However, during the unloading the error increases for both amplitudes. For 8% amplitude the deviation is apparent. In Figure 4.11 (d) the Young's modulus E at specific points of the loading cycle is plotted for every amplitude. We computed the values with the reference data at the start of the loading (E_1), at the transition from positive to negative strain (E_2) and at the end of the first loading cycle (E_3). For all amplitudes the Young's modulus decreases during the loading cycle. From the start of the loading to the end of the first cycle, the Young's modulus reduces about 200 MPa in the tests with 1% and 5% amplitude. However, in the test with an amplitude of 8%, the Young's modulus already decreases by 700 MPa between the first and the second data point.

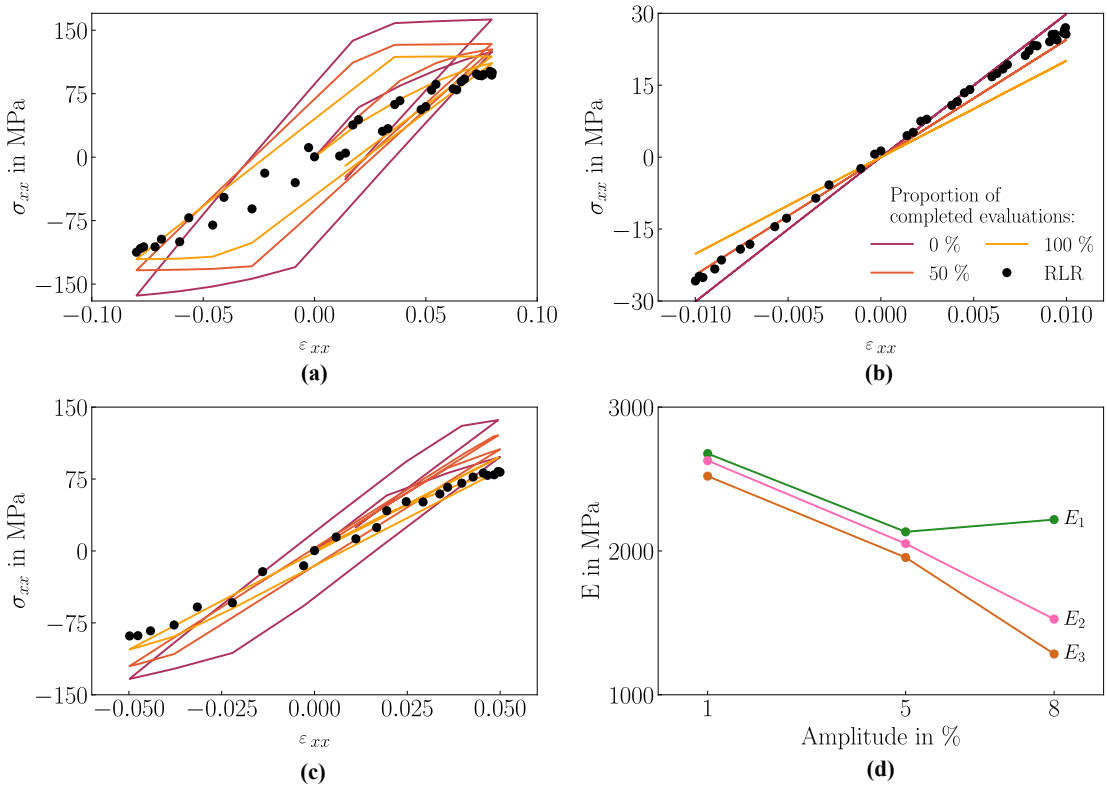


Figure 4.11: Optimised and reference load reactions (RLR) σ_{xx} for material with mixing ratio 6:3 under sinusoidal tensile strain applied in 1.5 loading cycles with load parameter set for: (a) 8% amplitude; (b) 1% amplitude; (c) 5% amplitude; (d) values of Young's modulus E during the load application with E_1 at start of the loading cycle, E_2 at transition to negative strain, E_3 at the end of the loading cycle.

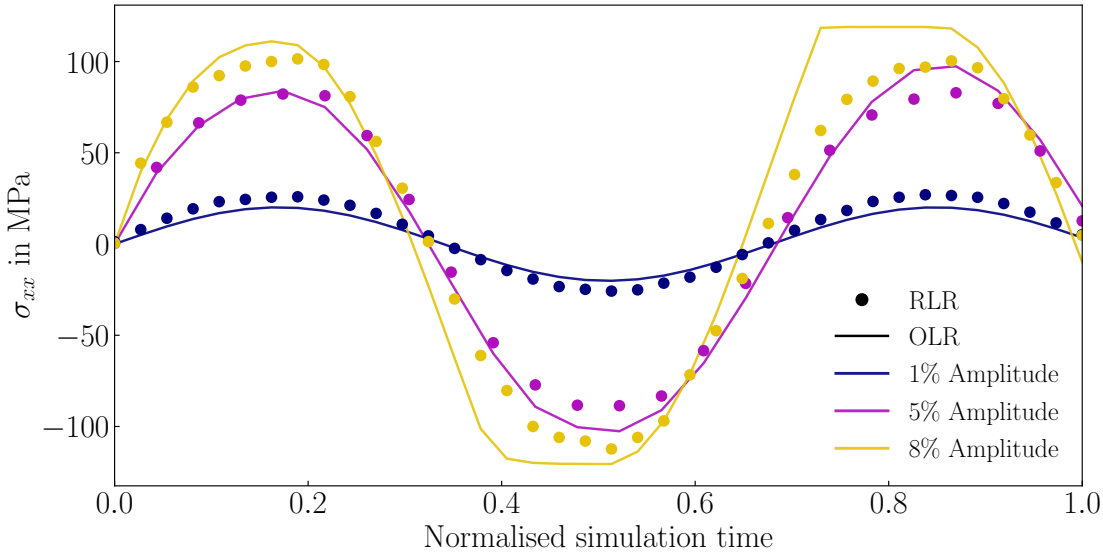


Figure 4.12: Optimised and reference load reactions (RLR) σ_{xx} for material with mixing ratio 6:3 under sinusoidal tensile strain applied in 1.5 loading cycles with amplitude 1, 5 and 8% over normalised simulation time for an exemplary test.

To simplify the visualisation for the periodic loading, we plotted the final load reactions from the same exemplary test over a normalised simulation time in Figure 4.12. For 1% amplitude, the optimised load reactions have constantly smaller magnitudes than the reference data. Nevertheless, the reference and the optimised data both follow a sinusoidal function during the whole simulation time. For the other two amplitudes, the data match in the first increasing load steps. During the rest of the loading path, the deviations increase with their maximum values right before or when the maximum stress magnitude is reached.

Discussion The presented test series provides additional information about the capabilities of the optimisation approach. First, the algorithm is able to run an optimisation with three load parameter sets at a time. Since the RMSE decreases, the optimisation works in principle. However, the result in terms of measured load reactions σ_{xx}, is insufficient. Only the first loading path is adequately fitted with the optimised material parameters. During the negative loading, the deviations increase, which could be explained through changes in the material properties. We assume, that in both cases the plastic regime is reached before the maximum amplitude. Therefore, a hardening process started, which leads to material damage. However, not only the plastic behaviour is influenced. The decrease of the Young's modulus during the optimisation process, depicted in Figure 4.11 (d), demonstrates, that the elastic properties are also affected. Because of the intensification of the effect as the amplitude increases, we also assume increasing material damage. These observations indicate material damage which influences the elastic and the plastic material properties. Since the damage models included in Abaqus do not handle material damage in the elastic regime, a self-written subroutine is necessary to determine the damage behaviour. The inclusion of material damage would lead to an improvement of the approach to process cyclic load parameters, which should be part of future investigations.

5 Conclusion

Summary In this thesis, an optimisation approach to find material parameters for epoxies modelled with MD simulations, was developed. The mechanical behaviour of the material should be represented as good as possible through a constitutive model with corresponding material parameters. As constitutive model we use an elastoplastic model with a VOCE-hardening curve to describe the plastification. This model is defined with two elastic and four plastic material parameters. To monitor the quality of the material parameters, we perform simulations with the FE software ABAQUS. We implemented the whole optimisation algorithm in a single PYTHON script, since ABAQUS has an application programming interface which enables its control via PYTHON commands.
Before we start the optimisation loop, a model with specific properties is created in the preprocessing. The optimisation is started with an initial guess for the material parameters. With these parameters we perform the FE simulation, and extract the resulting load reactions. We compare the load reactions with the ones from the MD simulations, which we use as reference data. The deviations are summarised in a single RMSE value. This value is reduced through the numerical Nelder-Mead algorithm, which is able to optimise a scalar function in a multidimensional space. The algorithm adapts the material parameters and a new evaluation starts. To verify the algorithm, we used reference data from MD simulations performed by RIES et al. [0] for materials with mixing ratio 6:3, where a linear tensile strain up to a maximum strain of 20% was applied. We performed tests with the same loading conditions for materials with mixing ratio 4:3, 6:3 and 8:3 to validate the performance of the optimisation algorithm. For material with mixing ratio 6:3, we applied sinusoidal strain up to a maximum amplitude of 15%. We performed optimisation tests with these load parameters as tensile loading, shear loading, and finally their combination. In the final tests, we applied 1.5 loading cycles of sinusoidal tensile strains with 1%, 5% and 8% amplitude.

Conclusion The developed optimisation approach is able to minimise the error between the reference data and the optimised load reactions through adaptation of the material parameters. The verification study showed, that the optimisation procedure achieves a perfect match of the load reactions for pure tensile loading through a linear strain application. However, the identified material parameters strongly deviate within the test series. We attribute this behaviour to the relatively high number of optimisation variables. Therefore, we predefined the elastic parameters E and ν, and optimised the plastic parameters. In the validation study, the algorithm reliably found similar material parameters independent of the initial value combination. In addition, the optimised material parameters agree with the reference values found by RIES et al. [0]. The studies using sinusoidal load application provide information about the algorithm performance in different load cases. The pure tensile loading led to results of similar quality as the validation study, whereas in the shear load case issues about the optimisation of the plastic yield occurred. Possible reasons for this behaviour were discussed in Subsection 4.3.4. In the reference data for the cyclic tests, the material properties change during the loading process, which might be explained through arising material damage. Since we disre-

garded damage models in our ABAQUS simulation, the optimised load reactions show high variations from the reference data. The performed tests demonstrate, that from the tested loading procedures, optimisation of tensile loadings gives reliable results. For the other tested configurations, the results of the material parameters are still arbitrary.

Outlook The processing of shear load cases should be part of future investigations. Here, the focus should be taken on the transition from elastic to plastic material behaviour. The tests performed in this work demonstrate open issues with the definition of the yield stress, which represents the starting point of the plastic regime. For an adequate representation of cyclic loading, a damage model should be included in the ABAQUS model. Because of unusual material behaviour, a subroutine would be necessary. Furthermore, the properties of the numerical optimisation algorithm could be part of future investigations. Here, the initial value combinations were picked arbitrary. Since this may lead to getting stuck in local minima, the impact of the initial value combination on the optimisation approach should be studied. In addition, the choice of the numerical algorithm could be reflected. The sensitivity of the algorithm to the initial values could be reduced by choosing alternatives.

Bibliography

[0] M. Ries et al. "Deciphering elastoplast properties from atomistic structure: Reactive coarse-grained MD for epoxies". In: ().

[0] S. Koltzenburg, M. Maskos, & O. Nuyken. *Polymere: Synthese, Eigenschaften und Anwendungen*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2024. doi: 10.1007/978-3-662-64601-4. url: https://link.springer.com/10.1007/978-3-662-64601-4 (visited on 10/24/2025).

[0] M. Ries. "Characterization and modeling of polymer nanocomposites across the scales". PhD thesis. Erlangen: Friedrich-alexander Universität erlangen-Nürnberg, 2023.

[0] D. Rajak et al. "Fiber-Reinforced Polymer Composites: Manufacturing, Properties, and Applications". In: *Polymers* 11.10 (Oct. 12, 2019), p. 1667. doi: 10.3390/polym11101667. url: https://www.mdpi.com/2073-4360/11/10/1667 (visited on 10/24/2025).

[0] G. Jeevi, S. K. Nayak, & M. Abdul Kader. "Review on adhesive joints and their application in hybrid composite structures". In: *Journal of Adhesion Science and Technology* 33.14 (July 18, 2019). Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/0169424 pp. 1497–1520. doi: 10.1080/01694243.2018.1543528. url: https://doi.org/10.1080/01694243.2018.1543528 (visited on 10/17/2025).

[0] S. G. Prolongo, G. Del Rosario, & A. Ureña. "Comparative study on the adhesive properties of different epoxy resins". In: *International Journal of Adhesion and Adhesives* 26.3 (June 2006), pp. 125–132. doi: 10.1016/j.ijadhadh.2005.02.004. url: https://linkinghub.elsevier.com/retrieve/pii/S0143749605000333 (visited on 10/24/2025).

[0] R. Campilho et al. "eXtended Finite Element Method for fracture characterization of adhesive joints in pure mode I". In: *Computational Materials Science* 50.4 (Feb. 2011), pp. 1543–1549. doi: 10.1016/j.commatsci.2010.12.012. url: https://linkinghub.elsevier.com/retrieve/pii/S0927025610006695 (visited on 10/17/2025).

[0] A. Pramanik et al. "Joining of carbon fibre reinforced polymer (CFRP) composites and aluminium alloys – A review". In: *Composites Part A: Applied Science and Manufacturing* 101 (Oct. 2017), pp. 1–29. doi: 10.1016/j.compositesa.2017.06.007. url: https://linkinghub.elsevier.com/retrieve/pii/S1359835X1730235X (visited on 10/17/2025).

[0] M. Ries. "Mechanical behavior of adhesive joints: A review on modeling techniques". In: *Computer Methods in Materials Science* 24.4 (2024). doi: 10.7494/cmms.2024.4.1010. url: https://www.cmms.agh.edu.pl/2024_4_1010/ (visited on 10/06/2025).

[0] M. E. Tuckerman & G. J. Martyna. "Understanding Modern Molecular Dynamics: Techniques and Applications". In: *The Journal of Physical Chemistry B* 104.2 (Jan. 1, 2000), pp. 159–178. doi: 10.1021/jp992433y. url: https://pubs.acs.org/doi/10.1021/jp992433y (visited on 10/21/2025).

[0] W. F. van Gunsteren & H. J. C. Berendsen. "Computer Simulation of Molecular Dynamics: Methodology, Applications, and Perspectives in Chemistry". In: *Angewandte Chemie International Edition in English* 29.9 (1990). _eprint: https://onlinelibrary.wiley.com/d pp. 992–1023. doi: 10.1002/anie.199009921. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/anie.199009921 (visited on 10/21/2025).

[0] S. Gorbunov, A. Volkov, & R. Voronkov. "Periodic boundary conditions effects on atomic dynamics analysis". In: *Computer Physics Communications* 279 (Oct. 2022), p. 108454. doi: 10.1016/j.cpc.2022.108454. url: https://linkinghub.elsevier.com/retrieve/pii/S0010465522001734 (visited on 10/06/2025).

[0] O. Büyüköztürk et al. "Structural solution using molecular dynamics: Fundamentals and a case study of epoxy-silica interface". In: *International Journal of Solids and Structures* 48.14 (July 2011). Publisher: Elsevier BV, pp. 2131–2140. doi: 10.1016/j.ijsolstr.2011.03.018. url: https://linkinghub.elsevier.com/retrieve/pii/S002076831100120X (visited on 07/24/2025).

[0] J. Mergheim. "Lecture Notes Materials Modelling and Simulation". In: ().

[0] I. M. Ward, I. M. Ward, & J. Sweeney. *Mechanical properties of solid polymers*. 3. ed. Chichester: Wiley, 2013. 461 pp.

[0] N. Saabye Ottosen & M. Ristinmaa. *The Mechanics of Constitutive Modeling*. Elsevier, 2005. doi: 10.1016/B978-0-08-044606-6.X5000-0. url: https://linkinghub.elsevier.com/retrieve/pii/B9780080446066X50000 (visited on 10/21/2025).

[0] Voce. "A Practical Strain-Hardening Function". In: Metallurgia (1948).

[0] M. Jung & U. Langer. *Methode der finiten Elemente für Ingenieure: Eine Einführung in die numerischen Grundlagen und Computersimulation*. Wiesbaden: Springer Fachmedien Wiesbaden, 2013. doi: 10.1007/978-3-658-01101-7. url: https://link.springer.com/10.1007/978-3-658-01101-7 (visited on 10/21/2025).

[0] K. Willner. "Vorlesungsskript Methode der finiten Elemente". Vorlesungsskript. Vorlesungsskript. Erlangen. (Visited on 10/09/2025).

[0] V. Jagota, A. P. S. Sethi, & K. Kumar. "Finite Element Method: An Overview". In: *Finite Element Method* ().

[0] P. Steinke. *Finite-Elemente-Methode: Rechnergestützte Einführung*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. doi: 10.1007/978-3-642-53937-4. url: https://link.springer.com/10.1007/978-3-642-53937-4 (visited on 10/13/2025).

[0] D. Systems. *Abaqus Scripting User's Guide*. 2015.

[0] S. L. Omairey, P. D. Dunning, & S. Sriramula. "Development of an ABAQUS plugin tool for periodic RVE homogenisation". In: *Engineering with Computers* 35.2 (Apr. 2019), pp. 567–577. doi: 10.1007/s00366-018-0616-4. url: http://link.springer.com/10.1007/s00366-018-0616-4 (visited on 10/13/2025).

[0] L. M. Rios & N. V. Sahinidis. "Derivative-free optimization: a review of algorithms and comparison of software implementations". In: *Journal of Global Optimization* 56.3 (July 2013), pp. 1247–1293. doi: 10.1007/s10898-012-9951-y. url: https://link.springer.com/10.1007/s10898-012-9951-y (visited on 10/21/2025).

[0] F. Gao & L. Han. "Implementing the Nelder-Mead simplex algorithm with adaptive parameters". In: *Computational Optimization and Applications* 51.1 (Jan. 2012). Publisher: Springer Science and Business Media LLC, pp. 259–277. doi: 10.1007/s10589-010-9329-3. url: http://link.springer.com/10.1007/s10589-010-9329-3 (visited on 07/24/2025).

[0] N. Pham, A. Malinowski, & T. Bartczak. "Comparative Study of Derivative Free Optimization Algorithms". In: *IEEE Transactions on Industrial Informatics* 7.4 (Nov. 2011), pp. 592–600. doi: 10.1109/TII.2011.2166799. url: https://ieeexplore.ieee.org/document/6011694/ (visited on 10/04/2025).

[0] S. Singer & S. Singer. "Efficient Implementation of the Nelder–Mead Search Algorithm". In: *Applied Numerical Analysis & Computational Mathematics* 1.2 (2004). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/anac.200410015, pp. 524–534. doi: 10.1002/anac.200410015. url: https://onlinelibrary.wiley.com/doi/abs/10.1002/anac.200410015 (visited on 10/04/2025).

[0] J. A. Nelder & R. Mead. "A Simplex Method for Function Minimization". In: *The Computer Journal* 7.4 (Jan. 1, 1965). Publisher: Oxford University Press (OUP), pp. 308–313. doi: 10.1093/comjnl/7.4.308. url: https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/7.4.308 (visited on 07/26/2025).

[0] M. Baudin. "Nelder-Mead User's Manual". In: ().

[0] M. A. Luersen & R. Le Riche. "Globalized Nelder–Mead method for engineering optimization". In: *Computers & Structures* 82.23 (Sept. 2004). Publisher: Elsevier BV, pp. 2251–2260. doi: 10.1016/j.compstruc.2004.03.072. url: https://linkinghub.elsevier.com/retrieve/pii/S0045794904002378 (visited on 07/24/2025).

[0] D. A. Morrow et al. "A method for assessing the fit of a constitutive material model to experimental stress–strain data". In: *Computer Methods in Biomechanics and Biomedical Engineering* 13.2 (Apr. 2010), pp. 247–256. doi: 10.1080/10255840903170686. url: http://www.tandfonline.com/doi/abs/10.1080/10255840903170686 (visited on 10/13/2025).

A Additional results

A.1 Validation plots

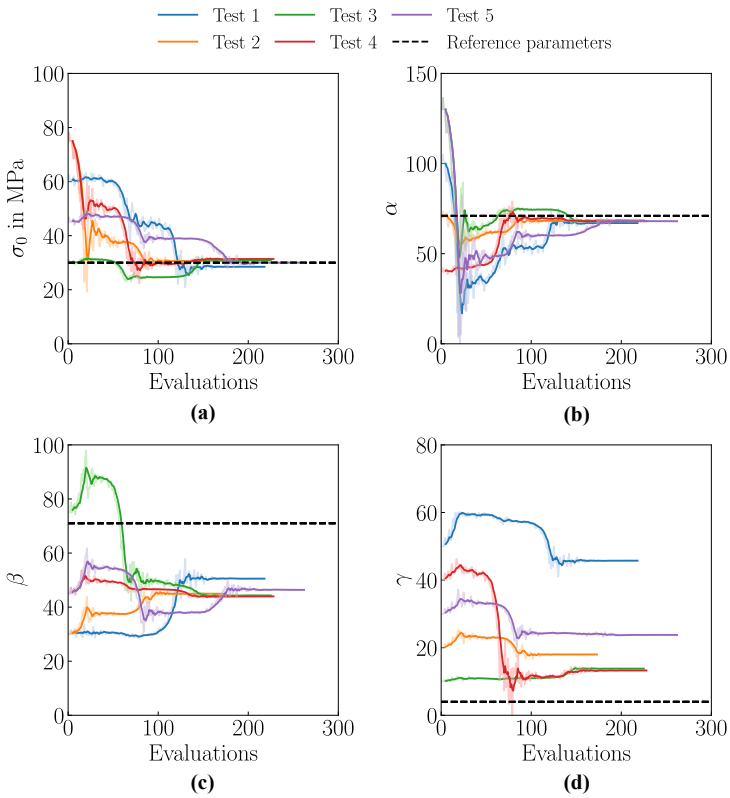


Figure A.1: Evolution of the optimised material parameters: (a) yield stress σ_0; hardening coefficients (b) α; (c) β; (d) γ; over the optimisation evaluations for material with mixing ratio 4:3 under linear tensile strain with respective reference values obtained by Ries et al. [0] and predefined elastic parameters Young's modulus E and Poisson's ratio ν.

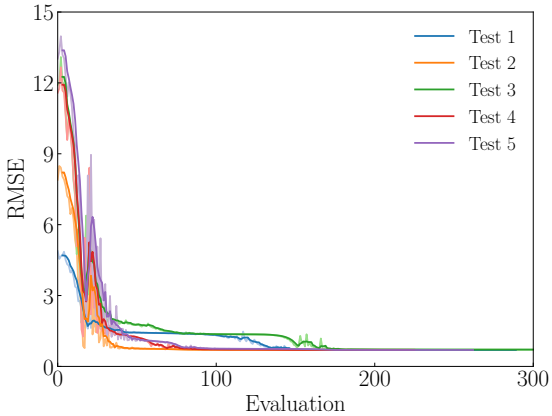


Figure A.2: Final optimised load reactions and reference load reactions (RLR) ε_{yy} over applied linear tensile strain ε_{xx} for an exemplary test with material with mixing ratio 4:3 with predefined elastic parameters Young's modulus E and Poisson's ratio ν.

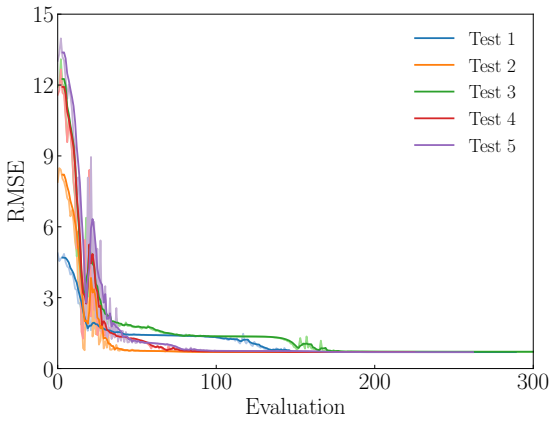


Figure A.3: Evolution of the root mean squared error (RMSE) during the optimisation for all tests for material with mixing ratio 4:3 under linear tensile strain.

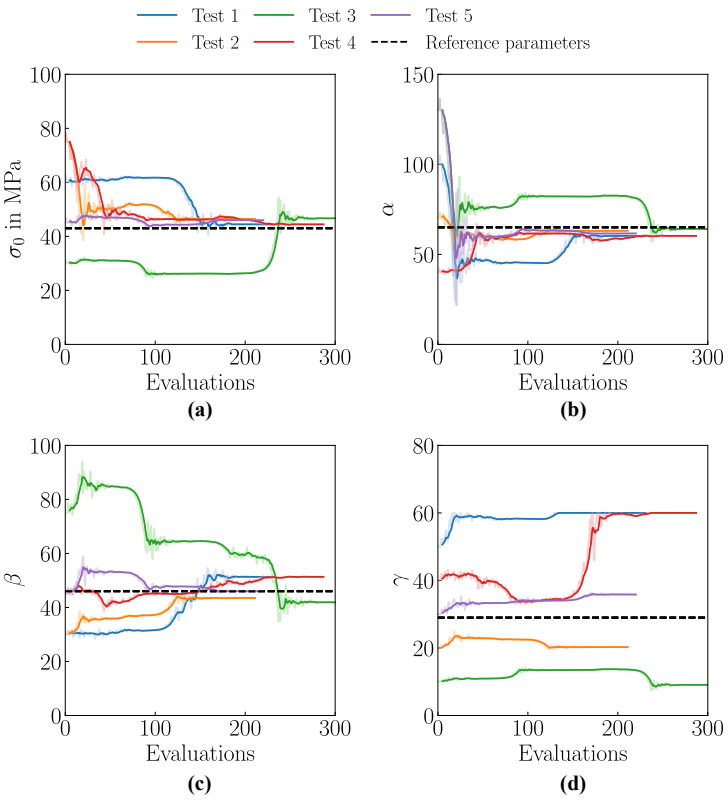


Figure A.4: Evolution of the optimised material parameters: (a) yield stress σ_0; hardening coefficients (b) α; (c) β; (d) γ; over the optimisation evaluations for material with mixing ratio 8:3 under linear tensile strain with respective reference values obtained by Ries et al. [0] and predefined elastic parameters Young's modulus E and Poisson's ratio ν.

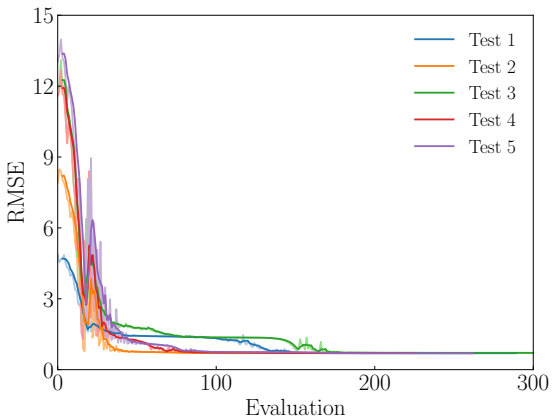


Figure A.5: Final optimised load reactions and reference load reactions (RLR) ε_{yy} over applied linear tensile strain ε_{xx} for an exemplary test with material with mixing ratio 8:3 with predefined elastic parameters Young's modulus E and Poisson's ratio ν.

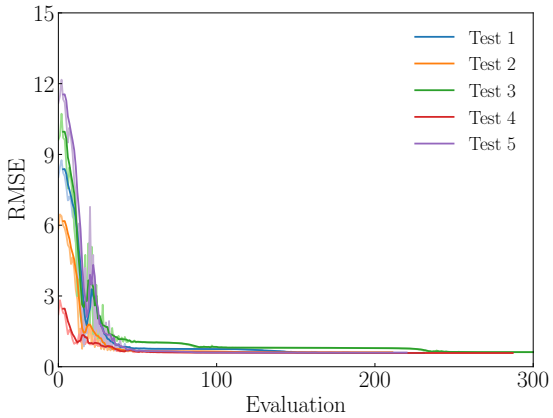


Figure A.6: Evolution of the root mean squared error (RMSE) during the optimisation for all tests for material with mixing ratio 8:3 under linear tensile strain.

A.2 Tensile-Shear combination plots

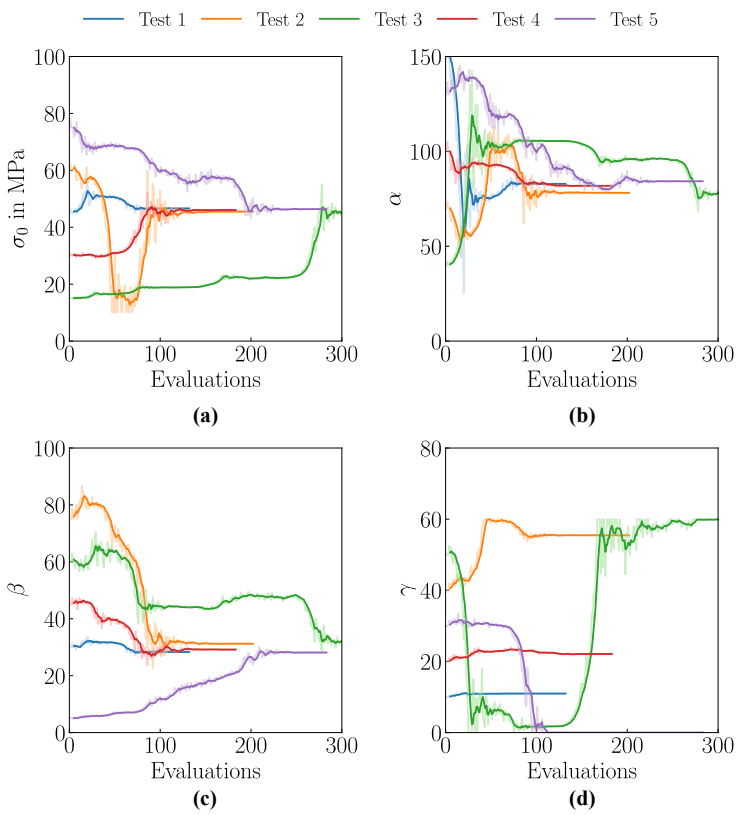


Figure A.7: Evolution of the optimised material parameters: (a) yield stress σ_0; hardening coefficients (b) α; (c) β; (d) γ; over the optimisation evaluations for material with mixing ratio 6:3 under pure sinusoidal tensile strain and predefined elastic parameters Young's modulus E and Poisson's ratio ν.

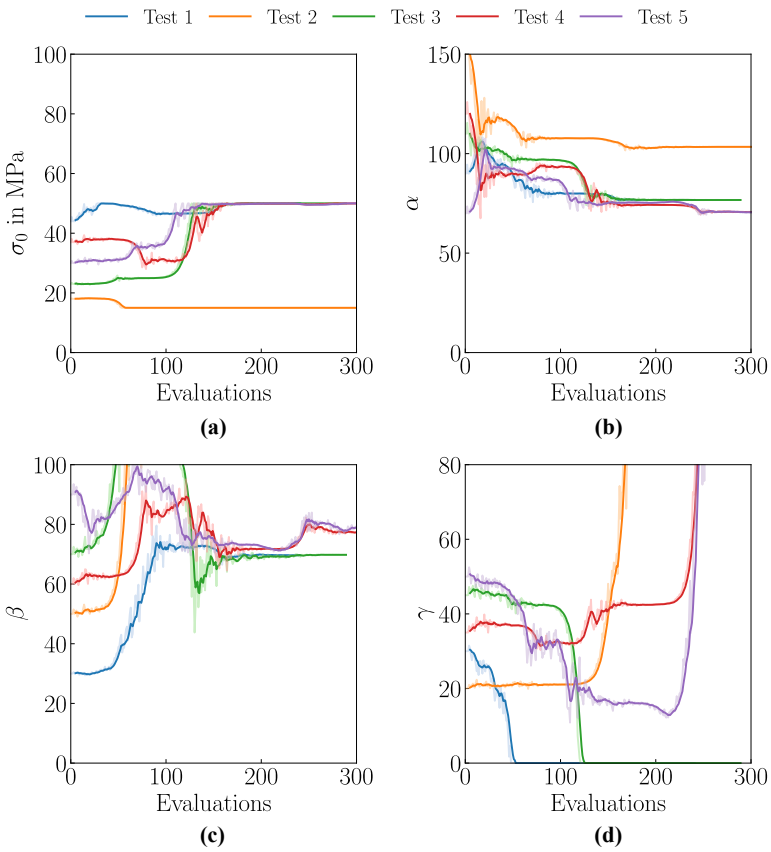


Figure A.8: Evolution of the optimised material parameters: (a) yield stress σ_0; hardening coefficients (b) α; (c) β; (d) γ; over the optimisation evaluations for material with mixing ratio 6:3 under pure sinusoidal shear strain and predefined elastic parameters Young's modulus E and Poisson's ratio ν.

XX STRAIN STRAIN FÜR VALIDIERUNGSVERSUCHE XX CODE UND INPUT FILE

B Code

B.1 Input file

```
{   "Modelname": "Cube",
    "CubeSize": 1.0,
    "MaterialName": "CyclicMD",
    "MaterialType": "elasto-plastic",
    "YoungsModulus": {
        "value": [2500, 3000],
        "min": 2000,
        "max": 4500
    },
    "PoissonRatio": {
        "value": [0.32, 0.38],
        "min": 0.23,
        "max": 0.45
    },

    "PlasticYield": {
        "value": [50.0, 35.0],
        "min": 10.0,
        "max": 100.0
    },
    "Alpha": {
        "value": [100.0, 70.0],
        "min": 0.0,
        "max": 200.0
    },
    "Beta": {
        "value": [45.0, 20.0],
        "min": 0.0,
        "max": 100.0
    },
    "Gamma": {
        "value": [10.0, 40.0],
        "min": 0.0,
        "max": 60.0
    },
    "C10": {
        "min": 310.0,
        "max": 650.0
    },
    "D1": {
        "min": 0.00018,
        "max": 0.0018
    },
    "PlasticMaterialParameters": "voce.dat",
    "NumberOfElementsPerEdge": 6,
```

```
46          "MDDataFiles": [
47              {"filename": "exemplary_reference_data.dat", "weight":
    1.0}
48          ],
49          "PrescribedDirections": {
50              "E11": {"active": 1, "weight": 1.0},
51              "E22": {"active": 0, "weight": 0.0},
52              "E33": {"active": 0, "weight": 0.0},
53              "G12": {"active": 1, "weight": 1.0},
54              "G13": {"active": 0, "weight": 0.0},
55              "G23": {"active": 0, "weight": 0.0}
56          },
57          "StressAnalysisDirection": {
58              "xx": 1,
59              "yy": 0,
60              "zz": 0,
61              "xy": 1,
62              "xz": 0,
63              "yz": 0
64          },
65          "StrainAnalysisDirection": {
66              "xx": 0,
67              "yy": 1,
68              "zz": 1,
69              "xy": 0,
70              "xz": 0,
71              "yz": 0
72          },
73          "weights": {
74              "normalStress": 1,
75              "normalStrain": 1e4,
76              "shearStress": 1,
77              "shearStrain": 1e30
78          },
79          "NumberOfIterations": 300,
80          "Testname": "tensile_shear_combi_test"
81      }
82
83
```

B.2 Optimisation algorithm

```
from abaqus import *
import job
from fontTools.misc.bezierTools import epsilon
from odbAccess import *
from abaqusConstants import *
from odbMaterial import *
from odbSection import *

import sketch
import part
import assembly
import step
import load
import mesh
import optimization
import job
import visualization
import connectorBehavior
import regionToolset

import sys
import numpy as np
import os
import glob
import json
import csv

from copy import deepcopy

# path to location of EasyPBC plug-in
sys.path.insert(0,'/home/rzlin/ib92ifar/abaqus_plugins/EasyPBC V.1.4')
import easypbc

from scipy.optimize import minimize

#read input file and create CubeParameter object
def read_parameter_file(filename, input_directory):
    with open(filename, 'r') as file:
        input_data = json.load(file)

    params = CubeParameters(
        model_name=input_data['Modelname'],
        part_name=f"{input_data['Modelname']}_Part",
        instance_name=f"{input_data['Modelname']}_Instance",
        section_name=f"{input_data['Modelname']}_Section",
        cube_size=input_data['CubeSize'],
        material_name=input_data['MaterialName'],
        material_type=input_data['MaterialType'],
        youngs_modulus_bounds=np.array([input_data['YoungsModulus']['min'], input_data['YoungsModulus']['max']]),
        youngs_modulus = input_data['YoungsModulus']['value'],
        poisson_ratio_bounds=np.array([input_data['PoissonRatio']['min'], input_data['PoissonRatio']['max']]),
        poisson_ratio=input_data['PoissonRatio']['value'],  # Initial value
        plastic_yield_bounds=np.array([input_data['PlasticYield']['min'], input_data['PlasticYield']['max']]),
```

```
        plastic_yield=input_data['PlasticYield']['value'],  # Initial value
        alpha_bounds=np.array([input_data['Alpha']['min'], input_data['Alpha']['max']]),
        alpha=input_data['Alpha']['value'],  # Initial value
        beta_bounds=np.array([input_data['Beta']['min'], input_data['Beta']['max']]),
        beta=input_data['Beta']['value'],  # Initial value
        gamma_bounds=np.array([input_data['Gamma']['min'], input_data['Gamma']['max']]),
        gamma=input_data['Gamma']['value'],  # Initial value
        C10_bounds=np.array([input_data['C10']['min'], input_data['C10']['max']]),
        D1_bounds=np.array([input_data['D1']['min'], input_data['D1']['max']]),
        plastic_param_file=input_data['PlasticMaterialParameters'],
        element_number=input_data['NumberOfElementsPerEdge'],
        md_data_dict = {file_info["filename"]: file_info["weight"] for file_info in input_data["MDDataFiles"]},
        prescribed_directions= input_data['PrescribedDirections'],
        stress_analysis_direction=input_data['StressAnalysisDirection'],
        strain_analysis_direction=input_data['StrainAnalysisDirection'],
        weights=input_data['weights'],
        iteration_number=input_data['NumberOfIterations'],
        test_name=input_data['Testname'],
        input_dirctory = input_directory
    )

    return params

#create CubeParameters object to store input parameters
class CubeParameters:
    def __init__(self, model_name, part_name, instance_name, section_name,
                 cube_size, material_name, material_type,
                 youngs_modulus_bounds, youngs_modulus, poisson_ratio_bounds, poisson_ratio,
                 plastic_param_file, plastic_yield_bounds, plastic_yield,
                 alpha_bounds, alpha, beta_bounds, beta,
                 gamma_bounds, gamma, C10_bounds, D1_bounds,
                 element_number, md_data_dict, prescribed_directions,
                 stress_analysis_direction, strain_analysis_direction, weights, iteration_number, test_name, input_dirctory):

        self.input_dirctory = input_dirctory
        self.model_name = model_name
        self.part_name = part_name
        self.instance_name = instance_name
        self.section_name = section_name
        self.cube_size = cube_size
        self.material_name = material_name
        self.material_type = material_type
        self.youngs_modulus_bounds = youngs_modulus_bounds
        self.youngs_modulus = youngs_modulus
        self.poisson_ratio_bounds = poisson_ratio_bounds
        self.poisson_ratio = poisson_ratio
        self.plastic_param_file = plastic_param_file
        self.plastic_strain = self.read_plastic_strain()
        self.number_plast_values = len(self.plastic_strain)
        self.plastic_yield_bounds = plastic_yield_bounds
```

```
        self.plastic_yield = np.array(plastic_yield)
        self.alpha_bounds = alpha_bounds
        self.alpha = np.array(alpha)
        self.beta_bounds = beta_bounds
        self.beta = np.array(beta)
        self.gamma_bounds = gamma_bounds
        self.gamma = np.array(gamma)
        self.C10_bounds = C10_bounds
        self.D1_bounds = D1_bounds

        self.element_number = element_number
        self.md_data_dict = md_data_dict
        self.prescribed_directions = prescribed_directions
        self.stress_analysis_direction = stress_analysis_direction
        self.strain_analysis_direction = strain_analysis_direction
        self.weights = weights
        self.iteration_number = iteration_number
        self.test_name = test_name

        self.shear_modulus = self.youngs_modulus/ (2*(1 + self.
    poisson_ratio))
        self.C10 = self.shear_modulus / 2
        self.bulk_modulus = self.youngs_modulus / (3 * (1- 2 * self.
    poisson_ratio))
        self.D1 = 2 / self.bulk_modulus

        self.C10_scaled = ((self.C10 - self.C10_bounds[0]) /
        (self.C10_bounds[1] -self.C10_bounds[0]))

        self.D1_scaled = ((self.D1 - self.D1_bounds[0]) /
        (self.D1_bounds[1] -self.D1_bounds[0]))

        self.youngs_modulus_scaled = ((self.youngs_modulus - self.
    youngs_modulus_bounds[0]) /
        (self.youngs_modulus_bounds[1] -self.youngs_modulus_bounds[0]))

        self.poisson_ratio_scaled = ((self.poisson_ratio - self.
    poisson_ratio_bounds[0] ) /
        (self.poisson_ratio_bounds[1] - self.poisson_ratio_bounds[0]))

        self.plastic_yield_scaled = ((self.plastic_yield - self.
    plastic_yield_bounds[0]) /
        (self.plastic_yield_bounds[1] - self.plastic_yield_bounds[0]))

        self.alpha_scaled = ((self.alpha - self.alpha_bounds[0]) /
        (self.alpha_bounds[1] -self.alpha_bounds[0]))

        self.beta_scaled = ((self.beta - self.beta_bounds[0]) /
        (self.beta_bounds[1] -self.beta_bounds[0]))

        self.gamma_scaled = ((self.gamma - self.gamma_bounds[0]) /
        (self.gamma_bounds[1] -self.gamma_bounds[0]))

        self.number_of_params = self.check_array_lengths()

    #read plastic strain values to apply them as amplitude
    def read_plastic_strain(self):
        input_file_path = os.path.join(self.input_dirctory, self.
    plastic_param_file)
        data = np.loadtxt(input_file_path, delimiter=' ', skiprows=1)
```

```
        strain = data[:, 1]
        return strain

    # checl if all material parameter arrays have the same length
    def check_array_lengths(self):
        arrays = [
            self.plastic_yield,
            self.alpha,
            self.beta,
            self.gamma
        ]

        lengths = [len(array) for array in arrays]
        if all(length == lengths[0] for length in lengths):
            print(f"All arrays have the same length: {lengths[0]}")
            return lengths[0]
        else:
            raise ValueError("Arrays have the different length.")

#create reference data object
class MDData:
    def __init__(self, filename, input_directory):
        self.filename = filename
        self.input_directory = input_directory
        self.data = self.read_md_file()

    # function to read md-data from file
    def read_md_file(self):
        md_path = os.path.join(self.input_directory, self.filename)
        with open(md_path, 'r') as file:
            next(file)
            md_data = {
                "step": [],
                "strain_xx": [],
                "strain_yy": [],
                "strain_zz": [],
                "strain_xy": [],
                "strain_xz": [],
                "strain_yz": [],
                "stress_xx": [],
                "stress_yy": [],
                "stress_zz": [],
                "stress_xy": [],
                "stress_xz": [],
                "stress_yz": []
            }

            # column readout depends on md-data file layout
            for line in file:
                values = line.split()
                md_data["step"].append(float(values[0]))
                md_data["strain_xx"].append(float(values[1]))
                md_data["strain_yy"].append(float(values[2]))
                md_data["strain_zz"].append(float(values[3]))
                md_data["strain_xy"].append(float(values[4]))
                md_data["strain_xz"].append(float(values[5]) if len(values) > 5 else 0.0)
                md_data["strain_yz"].append(float(values[6]) if len(values) > 6 else 0.0)
                md_data["stress_xx"].append(float(values[7]))
```

```
                md_data["stress_yy"].append(float(values[8]) if len(values) > 8 else 0.0)
                md_data["stress_zz"].append(float(values[9]) if len(values) > 9 else 0.0)
                md_data["stress_xy"].append(float(values[10]) if len(values) > 10 else 0.0)
                md_data["stress_xz"].append(float(values[11]) if len(values) > 11 else 0.0)
                md_data["stress_yz"].append(float(values[12]) if len(values) > 12 else 0.0)

        return md_data

#create Cube object in Abaqus
class MDBCube:
    def __init__(self, cube_parameters, filename, work_directory, direction):
        self.parameters = deepcopy(cube_parameters)
        self.filename = filename
        self.work_directory = work_directory
        self.md_data = self.read_md_file(filename)
        self.parameters.model_name = self.rename_model(self.parameters.model_name, self.filename, direction)

    def read_md_file(self, filename):
        with open(filename, 'r') as file:
            next(file)
            md_data = {
                "step": [],
                "strain_xx": [],
                "strain_yy": [],
                "strain_zz": [],
                "strain_xy": [],
                "strain_xz": [],
                "strain_yz": [],
                "stress_xx": [],
                "stress_yy": [],
                "stress_zz": [],
                "stress_xy": [],
                "stress_xz": [],
                "stress_yz": []
            }

            # column readout depends on md-data file layout
            for line in file:
                values = line.split()
                md_data["step"].append(float(values[0]))
                md_data["strain_xx"].append(float(values[1]))
                md_data["strain_yy"].append(float(values[2]))
                md_data["strain_zz"].append(float(values[3]))
                md_data["strain_xy"].append(float(values[4]))
                md_data["strain_xz"].append(float(values[5]) if len(values) > 5 else 0.0)  # Addresses possible missing data
                md_data["strain_yz"].append(float(values[6]) if len(values) > 6 else 0.0)
                md_data["stress_xx"].append(float(values[7]))
                md_data["stress_yy"].append(float(values[8]) if len(values) > 8 else 0.0)
                md_data["stress_zz"].append(float(values[9]) if len(values) > 9 else 0.0)
```

```
                md_data["stress_xy"].append(float(values[10]) if len(values) > 10 else 0.0)
                md_data["stress_xz"].append(float(values[11]) if len(values) > 11 else 0.0)
                md_data["stress_yz"].append(float(values[12]) if len(values) > 12 else 0.0)

        return md_data

    # rename the model corresponding to the testname in the inputfile
    def rename_model(self,base_model_name, filename, direction):
        #base_model_name = parameters.model_name
        modefied_file_name = filename.replace('.', '_')
        short_file_name = modefied_file_name[-9:-4]
        modified_model_name = f"{base_model_name}_{short_file_name}_{direction} "
        return modified_model_name

    # create model
    def create_cube(self, i):
        # create skecth, part and model
        my_model = mdb.Model(name=self.parameters.model_name)
        sketch = my_model.ConstrainedSketch(name='__profile__', sheetSize=2 * self.parameters.cube_size)
        sketch.rectangle(point1=(0, 0), point2=(self.parameters.cube_size, self.parameters.cube_size))
        my_part = my_model.Part(name=self.parameters.model_name, dimensionality=THREE_D, type=DEFORMABLE_BODY)
        my_part.BaseSolidExtrude(sketch=sketch, depth=self.parameters.cube_size)
        my_part.Set(faces=my_part.faces, name='CubeFaces')

        # create material and assign it to a section
        my_material = mdb.models[self.parameters.model_name].Material(name=self.parameters.material_name)
        my_model.materials[self.parameters.material_name].Elastic(
            table=((self.parameters.youngs_modulus, self.parameters.poisson_ratio),))
        self.create_plastic_material(my_material, i)
        my_model.HomogeneousSolidSection(material=self.parameters.material_name, name=self.parameters.section_name,
                                         thickness=None)
        my_region = my_part.Set(cells=my_part.cells[:], name='Entire_Part')
        my_part.SectionAssignment(region=my_region, sectionName=self.parameters.section_name, offset=0.0,
                                  offsetField='', thicknessAssignment=FROM_SECTION)
        print(f"Section '{self.parameters.section_name}' assigned to part '{self.parameters.part_name}'.")

        # create instance and mesh part
        my_instance = my_model.rootAssembly.Instance(name=self.parameters.instance_name, part=my_part, dependent=OFF)
        element_size = self.parameters.cube_size / self.parameters.element_number
        my_model.rootAssembly.seedPartInstance(regions=(my_instance,), size=element_size, deviationFactor=element_size,
                                               minSizeFactor=0.1)
        my_model.rootAssembly.generateMesh(regions=(my_instance,))
```

```
        # create node- and elementsets at the surfaces
        for direction in ['xx', 'yy', 'zz']:
            coord_direction = ['xx', 'yy', 'zz'].index(direction)
            self.node_set_max_coord( coord_direction, f'Max_{direction}_Nodes', 1)
            self.node_set_max_coord( coord_direction, f'Min_{direction}_Nodes', -1)
            self.element_set_max_coord( coord_direction, f'Max_{direction}_Elements', 1)
            self.element_set_max_coord( coord_direction, f'Min_{direction}_Elements', -1)
        print(f"Cube model '{self.parameters.model_name}' created and Element- and Nodesets generated.")
        return my_model

    # search coordinate of surface node
    def search_maximum_node_coordinate(self, coord_direction, coord_id):
        target_coord = float('-inf') if coord_id == 1 else float('inf')
        instance = mdb.models[self.parameters.model_name].rootAssembly.instances[self.parameters.instance_name]
        for node in instance.nodes:
            act_coord = node.coordinates[coord_direction]
            if coord_id == 1 and act_coord > target_coord:
                target_coord = act_coord
            elif coord_id == -1 and act_coord < target_coord:
                target_coord = act_coord
        return target_coord

    # create surface nodeset
    def node_set_max_coord(self,  coord_direction, node_set_name, coord_id):
        target_coord = self.search_maximum_node_coordinate(coord_direction, coord_id)
        matching_nodes = []
        instance = mdb.models[self.parameters.model_name].rootAssembly.instances[self.parameters.instance_name]
        for node in instance.nodes:
            if node.coordinates[coord_direction] == target_coord:
                matching_nodes.append(node.label)
        if matching_nodes:
            created_set = mdb.models[self.parameters.model_name].rootAssembly.SetFromNodeLabels(
                name=node_set_name,
                nodeLabels=((self.parameters.instance_name, matching_nodes),)
            )
            print(f"Nodeset '{node_set_name}' with {len(matching_nodes)} nodes created.")
            return created_set
        else:
            print("Did not found matching node.")
            return None

    # create surface elementset
    def element_set_max_coord(self,  coord_direction, element_set_name, coord_id):
        target_coord = self.search_maximum_node_coordinate(coord_direction, coord_id)
        matching_elements = set()
```

```
        instance = mdb.models[self.parameters.model_name].rootAssembly.instances[self.parameters.instance_name]
        for element in instance.elements:
            for node_index in element.connectivity:
                node = instance.nodes[node_index]
                if node.coordinates[coord_direction] == target_coord:
                    matching_elements.add(element.label)
                    break
        if matching_elements:
            created_set = mdb.models[self.parameters.model_name].rootAssembly.SetFromElementLabels(
                name=element_set_name,
                elementLabels=((self.parameters.instance_name, list(matching_elements)),)
            )
            print(f"Elementset '{element_set_name}' with {len(matching_elements)} elements generated.")
            return created_set
        else:
            print("Did not found matching element.")
            return None

    # create plastic material section
    def create_plastic_material(self, material, n):
        plastic_stresses = []
        plastic_stresses = self.parameters.plastic_yield[n] + self.parameters.alpha[n] * (1 - np.exp(-self.parameters.beta[n] * self.parameters.plastic_strain)) + self.parameters.gamma[n] * self.parameters.plastic_strain
        plastic_data = []
        for m in range(self.parameters.number_plast_values):
            plastic_data.append(( plastic_stresses[m], self.parameters.plastic_strain[m],))
        material.Plastic(table=plastic_data)
        return None

    # create amplitude for load application
    def create_amplitude(self, direction):
        strain_map = {
        'E11': ('strain_xx', 'StrainXXAmplitude'),
        'E22': ('strain_yy', 'StrainYYAmplitude'),
        'E33': ('strain_zz', 'StrainZZAmplitude'),
        'G12': ('strain_xy', 'StrainXYAmplitude'),
        'G13': ('strain_xz', 'StrainXZAmplitude'),
        'G23': ('strain_yz', 'StrainYZAmplitude')
        }

        if direction not in strain_map:
            raise ValueError(f"Invalid direction specified: {direction}")

        strain_key, amplitude_name = strain_map[direction]
        strain_data = self.md_data[strain_key]
        step_data = self.md_data["step"]
        amplitude_points = tuple(zip(step_data, strain_data))
        mdb.models[self.parameters.model_name].TabularAmplitude(
            name=amplitude_name,
            timeSpan=STEP,
            smooth=SOLVER_DEFAULT,
            data=amplitude_points
        )
```

```
        print(f"Amplitude '{amplitude_name}' with {len(amplitude_points)} points created.")
        return amplitude_name

    # update increment setting and FieldOutput
    def update_increment_size(self):
        step_times = self.md_data["step"]
        if len(step_times) < 2:
            print("Not enough step-values to calculate increment size.")
            return

        total_time = step_times[-1]
        step_name = 'Step-1'
        mdb.models[self.parameters.model_name].steps[step_name].setValues(
            nlgeom=ON,
            initialInc=total_time*1e-2,
            maxNumInc=10000,
            maxInc = 1.0,
            noStop=OFF,
            timeIncrementationMethod=AUTOMATIC,
            timePeriod=total_time
        )
        time_points_name = 'OutputPoints'
        mdb.models[self.parameters.model_name].TimePoint(name=time_points_name, points=((step_times[0], step_times[-1] ,
        1.0), ))

        field_output_request = mdb.models[self.parameters.model_name].fieldOutputRequests['F-Output-1']
        field_output_request.setValues(variables=('S', 'PE', 'PEEQ', 'PEMAG', 'NE', 'LE', 'U', 'RF'))
        field_output_request.setValues(timePoint=time_points_name)
        return None

    # update boundary condition
    def update_boundary_condition(self, direction):
        bc_map_u = {
        'E11': ('E11-1', 'u1'),
        'E22': ('E22-1', 'u2'),
        'E33': ('E33-1', 'u3'),
        'G12': ('G12-1', 'u1', 'u2'),
        'G13': ('G13-1', 'u1', 'u3'),
        'G23': ('G23-1', 'u2', 'u3')
        }

        if direction not in bc_map_u:
            raise ValueError(f"Invalid direction specified: {direction}")

        bc_name = bc_map_u[direction][0]
        displacements = bc_map_u[direction][1:]

        model = mdb.models[self.parameters.model_name]
        if bc_name in model.boundaryConditions:
            amplitude_name = self.create_amplitude(direction)

            set_values = {displacements[0]: 1.0}
            #  for shear strain apply whole displacement in one direction
            if len(displacements) > 1:
                set_values[displacements[1]] = 0.0
```

```
            set_values['amplitude'] = amplitude_name
            model.boundaryConditions[bc_name].setValues(**set_values)
            print(f"Boundary Condition '{bc_name}' with amplitude '{amplitude_name}' updated in directions {displacements}.")
        else:
            print(f"Boundary Condition named '{bc_name}' not found.")

    # call easyPBC to create job
    def create_job(self, filename, direction):
        directions = {"E11": False, "E22": False, "E33": False,
                      "G12": False, "G13": False, "G23": False}

        if direction in directions:
            directions[direction] = True
        else:
            raise ValueError(f"Invalid direction specified: {direction}")

        easypbc.feasypbc(part=self.parameters.model_name, inst=self.parameters.instance_name, meshsens=1E-07, CPU=1,
                         E11=directions["E11"], E22=directions["E22"], E33=directions["E33"], G12=directions["G12"],
                         G13=directions["G13"], G23=directions["G23"],
                         onlyPBC=False, CTE=False, intemp=0, fntemp=100)
        modified_job_name = f"job-{direction}_{filename.replace('.dat', '')}"
        mdb.jobs.changeKey(fromName=f'job-{direction}', toName=modified_job_name)

        return modified_job_name

    # save MDB
    def save_mdb(self):
        mdb_file_name = os.path.join(self.work_directory, f"{self.parameters.model_name}.mdb")
        mdb.saveAs(mdb_file_name)
        if 'Model-1' in mdb.models:
            del mdb.models['Model-1']
        return None


# Optimisation function
def optimization_multiple_stresses(scaled_material, parameters, md_data, result_dir, evaluation_count):
    odb = None
    # variable to store rmse values for all jobs
    total_rmse = 0
    rmse_list = []
    # rescale material parameters
    material = []

    # for optimisation of elastic and plastic parameters use commented out lines
    material.append(parameters.youngs_modulus)
    material.append(parameters.poisson_ratio)
    #material.append(scaled_material[0] * (parameters.youngs_modulus_bounds[1] - parameters.youngs_modulus_bounds[0]) + parameters.youngs_modulus_bounds[0])
```

```
    #material.append(scaled_material[1] * (parameters.poisson_ratio_bounds[1] - parameters.poisson_ratio_bounds[0]) + parameters.poisson_ratio_bounds[0])
    material.append(scaled_material[0] * (parameters.plastic_yield_bounds[1] - parameters.plastic_yield_bounds[0]) + parameters.plastic_yield_bounds[0])
    material.append(scaled_material[1] * (parameters.alpha_bounds[1] - parameters.alpha_bounds[0]) + parameters.alpha_bounds[0])
    material.append(scaled_material[2] * (parameters.beta_bounds[1] - parameters.beta_bounds[0]) + parameters.beta_bounds[0])
    material.append(scaled_material[3] * (parameters.gamma_bounds[1] - parameters.gamma_bounds[0]) + parameters.gamma_bounds[0])
    material.append(parameters.C10)
    material.append(parameters.D1)

    print('Scaled material parameters in current evaluation:', scaled_material)
    print('Rescaled Material parameters in current evaluation:', material)

    # compute plastic stress values
    plastic_stresses = []
    plastic_stresses = material[2] + material[3] * (1 - np.exp(-material[4] * parameters.plastic_strain)) + material[5] * parameters.plastic_strain
    plastic_data = []
    for i in range(parameters.number_plast_values):
        plastic_data.append(( plastic_stresses[i], parameters.plastic_strain[i],))

    # create hyperelastic material
    for model_name in mdb.models.keys():
        model = mdb.models[model_name]
        del model.materials[parameters.material_name].elastic

        model.materials[parameters.material_name].Hyperelastic(
             materialType=ISOTROPIC, table=((material[6], material[7]), ), testData=OFF, type=
             NEO_HOOKE, volumetricResponse=VOLUMETRIC_DATA)

        model.materials[parameters.material_name].Plastic(
            scaleStress=None,
            table=plastic_data
        )
        print(
            f'Material parameters written for model "{model_name}": E = {material[0]}, '
            f'nu = {material[1]}, yield stress = {material[2]}, alpha = {material[3]}, beta = {material[4]}, gamma = {material[5]}, C10 = {material[6]}, D1 = {material[7]}')

    # check for lock-files
    lck_files = glob.glob('*.lck')
    if lck_files:
        for file in lck_files:
            try:
                os.remove(file)
            except Exception as e:
                print(f'Error when deleting {file}: {e}')
    else:
```

```
        print('No .lck-files found.')

    # Loop over all jobs
    job_names = list(mdb.jobs.keys())

    for job_name in job_names:
        if job_name in mdb.jobs:
            job_to_run = mdb.jobs[job_name]
            job_to_run.submit()
            job_to_run.waitForCompletion()
        else:
            print(f"Job '{job_name}' not found in MDB.")
            continue

        # search and open odb-file
        odb_file_name = f"{job_name}.odb"
        current_directory = os.getcwd()
        full_path = os.path.join(current_directory, odb_file_name)

        if os.path.isfile(full_path):
            odb = openOdb(full_path)

        else:
            print(f"ODB '{odb_file_name}' not found in directory: '{current_directory}'.")
            continue

        # Initialise stress-directory for current job
        if 'E' in job_name:
            stress_directory = {dir_name: [] for dir_name in ['xx', 'yy', 'zz']}
        elif 'G' in job_name:
            stress_directory = {dir_name: [] for dir_name in ['xy', 'xz', 'yz']}
        else:
            print('Unknown kind of job')

        # Loop over directions to fill stress_directory
        for i, dir_name in enumerate(stress_directory.keys()):
            odb_element_set = odb.rootAssembly.elementSets[f'MAX_XX_ELEMENTS']
            last_step_name = odb.steps.keys()[-1]
            last_step = odb.steps[last_step_name]
            for frame in last_step.frames:
                frame_stress = []
                stress_field = frame.fieldOutputs['S']
                stress_values = stress_field.getSubset(region=odb_element_set).values
                for value in stress_values:
                    if 'E' in job_name:
                        frame_stress.append(value.data[i])
                    elif 'G' in job_name:
                        frame_stress.append(value.data[i+3])
                    else:
                        print('Unknown kind of job')

                # store mean stress value from current frame in direction i in stress directory
                stress_directory[dir_name].append(np.mean(frame_stress))

```

```
    print(f"For job {job_name} the following stress values from the odb-data are stored: : {stress_directory}")

    # Initialise displacement difference directory for current job
    if 'E' in job_name:
        strain_directory = {dir_name: [] for dir_name in ['xx', 'yy', 'zz']}
    elif 'G' in job_name:
        strain_directory = {dir_name: [] for dir_name in ['xy', 'xz', 'yz']}
    else:
        print('Unknown kind of job')
        strain_directory = {}

    # Map normal displacement components to node set names
    node_set_map = {
        'xx': ('XX', 0),  # U1
        'yy': ('YY', 1),  # U2
        'zz': ('ZZ', 2)   # U3
    }

    # Map shear components to relevant displacement directions
    shear_components_map = {
        'xy': [('XX', 0), ('YY', 1)],  # Sum of U1 and U2
        'xz': [('XX', 0), ('ZZ', 2)],  # Sum of U1 and U3
        'yz': [('YY', 1), ('ZZ', 2)]   # Sum of U2 and U3
    }

    # Loop over directions to fill strain_directory
    for i,dir_name in enumerate(strain_directory.keys()):
        if dir_name in node_set_map:
            odb_element_set = odb.rootAssembly.elementSets[f'MAX_XX_ELEMENTS']
            last_step_name = odb.steps.keys()[-1]
            last_step = odb.steps[last_step_name]
            for frame in last_step.frames:
                frame_strain = []
                strain_field = frame.fieldOutputs['NE']
                strain_values = strain_field.getSubset(region=odb_element_set).values
                for value in strain_values:
                        frame_strain.append(value.data[i])

                strain_directory[dir_name].append(np.mean(frame_strain))

        elif dir_name in shear_components_map:
            odb_element_set = odb.rootAssembly.elementSets[f'MAX_XX_ELEMENTS']
            last_step_name = odb.steps.keys()[-1]
            last_step = odb.steps[last_step_name]
            for frame in last_step.frames:
                frame_strain = []
                strain_field = frame.fieldOutputs['NE']
                strain_values = strain_field.getSubset(region=odb_element_set).values
                for value in strain_values:
                        frame_strain.append(value.data[i+3])

```

```
                    # store mean stress value from current frame in direction i in strain directory
                    strain_directory[dir_name].append(np.mean(frame_strain))

        print(f"For job {job_name}, the following strain values are stored: {strain_directory}")

        # RMSE-evaluation for current job
        rmse_weighted, mse_dict = calculate_rmse(stress_directory, strain_directory, md_data, job_name, parameters, result_dir, evaluation_count)
        rmse_list.append(rmse_weighted)
        save_mse_array(mse_dict, job_name, result_dir, evaluation_count)
        save_stress_strain(job_name, stress_directory, strain_directory, result_dir, evaluation_count)

        odb.close()

    rmse_array = np.array(rmse_list)
    total_rmse = np.sum(rmse_array)
    save_material_params(material, result_dir, evaluation_count)
    save_rmse(total_rmse, result_dir, evaluation_count)
    evaluation_count[0] += 1
    print(f"RMSE value for all jobs: {total_rmse}")
    return total_rmse

# calculate RMSE
def calculate_rmse(stress_directory, strain_directory, md_file_dict, job_name, parameters, result_dir, evaluation_count):
    # access to md_data for given job
    if job_name not in md_file_dict:
        raise ValueError(f"Job '{job_name}' not found in md-data dictionary.")

    md_data = md_file_dict[job_name]
    mean_squared_differences = {}
    number_of_directions = 0

    # loop over stress directions in stress_directory for given job
    for dir_name, odb_stress_values in stress_directory.items():
        if parameters.stress_analysis_direction[dir_name] == 1:
            stress_key = dir_name
            md_stress_values = md_data.data[f"stress_{stress_key}"]
            print(f'For job {job_name} in {stress_key} - direction the following stress values from the md-data are stored: {md_stress_values}')

            # check whether md data and odb data have the same length
            if len(odb_stress_values) != len(md_stress_values):
                print(f'The length of the odb-data values in {stress_key}-direction is {len(odb_stress_values)}')
                print(f'The length of the md-data in {stress_key}-direction is {len(md_stress_values)}')
                raise ValueError(f"odb-data array and md-data array should have the same length for direction {dir_name}.")

            # calculation of squared differences for stress
            squared_differences_stress = (np.array(odb_stress_values) - np.array(md_stress_values)) ** 2
```

```
            weights_stress = np.ones_like(squared_differences_stress)
            weights_stress[1] = 100     # weight for elastic data point
            weighted_squared_differences = squared_differences_stress * weights_stress
            mean_squared_differences_stress = np.sum(weighted_squared_differences)/np.sum(weights_stress)
            if 'E' in job_name:
                mse_stress_weight = mean_squared_differences_stress * parameters.weights['normalStress']
            elif 'G' in job_name:
                mse_stress_weight = mean_squared_differences_stress * parameters.weights['shearStress']

            mean_squared_differences[f'stress {dir_name}'] = mse_stress_weight
            print('Mean Squared Diff with Stress', mean_squared_differences)
            number_of_directions += 1

    # loop over strain directions in strain_directory for given job
    for dir_name, odb_strain_values in strain_directory.items():
        if parameters.strain_analysis_direction[dir_name] == 1:
            strain_key = dir_name
            md_strain_values = md_data.data[f"strain_{strain_key}"]
            print(f'For job {job_name} in {strain_key} - direction the following strain values from the md-data are stored: {md_strain_values}')

            # check whether md data and odb data have the same length
            if len(odb_strain_values) != len(md_strain_values):
                print(f'The length of the odb-data values in {strain_key}-direction is {len(odb_strain_values)}')
                print(f'The length of the md-data in {strain_key}-direction is {len(md_strain_values)}')
                raise ValueError(f"odb-data array and md-data array should have the same length for direction {dir_name}.")

            print(f'For job {job_name} in {strain_key} - direction the following displacement values from the odb-data are stored: {odb_strain_values}')
            # calculation of squared differences for strain
            squared_differences_strain = (np.array(odb_strain_values) - np.array(md_strain_values)) ** 2

            weights_strain = np.ones_like(squared_differences_strain)
            weights_strain[1] = 100     # weight for elastic data point
            weighted_squared_differences_strain = squared_differences_strain * weights_strain
            mean_squared_differences_strain = np.sum(weighted_squared_differences_strain)/np.sum(weights_strain)
            if 'E' in job_name:
                mse_strain_weight = mean_squared_differences_strain * parameters.weights['normalStrain']
            elif 'G' in job_name:
                mse_strain_weight = mean_squared_differences_strain * parameters.weights['shearStrain']
            mean_squared_differences[f'strain {dir_name}'] = mse_strain_weight
            print('Mean Squared Diff with Strain', mean_squared_differences)
```

```
            print('Weights', parameters.weights)

            number_of_directions += 1

    # check if there are any directions for RMSE calculation
    if number_of_directions == 0:
        raise ValueError("No directions for RMSE calculation given.")

    # calculate RMSE
    mean_squared_diff = np.array(list(mean_squared_differences.values()))
    mean_squared_error = np.sum(mean_squared_diff) / number_of_directions
    rmse = np.sqrt(mean_squared_error)
    print(f"RMSE for job '{job_name}': {rmse}")
    rmse_dir_weight = 10.0

    for key in parameters.prescribed_directions.keys():
        if key in job_name:
            direction_weight = parameters.prescribed_directions[key]["weight"]
            rmse_dir_weight = rmse * direction_weight
            print(direction_weight)
            print(rmse_dir_weight)

    md_data_weight = parameters.md_data_dict.get(md_data.filename)
    print(md_data_weight)
    print(rmse_dir_weight)
    rmse_dir_md_weight = rmse_dir_weight * md_data_weight
    save_rmse_array(job_name, rmse, rmse_dir_weight, rmse_dir_md_weight, result_dir, evaluation_count)

    return rmse_dir_md_weight, mean_squared_differences

# save stress and strain components
def save_stress_strain(job_name, stress_directory, strain_directory, result_directory, evaluation):
    job_directory = os.path.join(str(result_directory), job_name)
    os.makedirs(job_directory, exist_ok=True)

    components = [key.replace('stress_', '') for key in stress_directory.keys()]
    first_stress_key = next(iter(stress_directory))
    num_values = len(stress_directory[first_stress_key])

    # prepare data for saving
    combined_values = []
    for i in range(num_values):
        row = []
        for component in components:
            stress_key = component
            strain_key = component
            row.extend([stress_directory[stress_key][i], strain_directory[strain_key][i]])
        combined_values.append(row)

    headers = []
    for component in components:
        headers.extend([f'Stress_{component}', f'Strain_{component}'])

    # write combined data to CSV for stress and strain
```

```
    combined_file_path = os.path.join(job_directory, f"stress_strain_{evaluation}.csv")
    with open(combined_file_path, mode='a', newline='') as file:
        writer = csv.writer(file)
        if os.path.getsize(combined_file_path) == 0:
            writer.writerow(headers)
        for values in combined_values:
            writer.writerow(values)

    print(f"Stress and strain values for (evaluation {evaluation}) have been saved.")

# save material parameters
def save_material_params(material, result_directory, evaluation):
    material_file_path = os.path.join(result_directory, "material_parameters.csv")
    with open(material_file_path, mode='a', newline='') as file:
        writer = csv.writer(file)
        if os.path.getsize(material_file_path) == 0:
            writer.writerow(['Evaluation', 'YoungsModulus', 'PoissonRatio', 'PlasticYield', 'Alpha', 'Beta', 'Gamma', 'C10', 'D1'])
        # add material parameters of curent evaluation
        writer.writerow([evaluation] + material)

    print(f"Material parameters for (evaluation {evaluation}) have been saved.")

# save RMSE
def save_rmse(rmse, result_directory, evaluation):
    rmse_file_path = os.path.join(result_directory, "rmse.csv")
    with open(rmse_file_path, mode='a', newline='') as file:
        writer = csv.writer(file)
        if os.path.getsize(rmse_file_path) == 0:
            writer.writerow(['Evaluation', 'RMSE'])
        # add RMSE value of current evaluation
        writer.writerow([evaluation, rmse])

    print(f" RMSE for (evaluation {evaluation}) have been saved.")

# save RMSE with weights of all jobs
def save_rmse_array(job_name, rmse, rmse_dir_weight, rmse_dir_md_weight, result_directory, evaluation):
    rmse_file_path = os.path.join(result_directory, "rmse_perJob.csv")
    with open(rmse_file_path, mode='a', newline='') as file:
        writer = csv.writer(file)
        if os.path.getsize(rmse_file_path) == 0:
            headers = ['Evaluation', 'Job', 'RMSE', 'RMSE_DirWeight', 'RMSE_Dir_MD_Weight']
            writer.writerow(headers)
        # write job name, and RMSE values for current evaluation
        writer.writerow([evaluation, job_name, rmse, rmse_dir_weight, rmse_dir_md_weight])

    print(f"RMSEs for evaluation {evaluation} and job {job_name} have been saved.")

# save MSE values
def save_mse_array(mse_dict,job_name, result_directory, evaluation):
    job_directory = os.path.join(str(result_directory), job_name)
    os.makedirs(job_directory, exist_ok=True)
```

```
    mse_file_path = os.path.join(job_directory, "mse.csv")
    mse_keys = sorted(mse_dict.keys())
    mse_values = [mse_dict[key] for key in mse_keys]
    with open(mse_file_path, mode='a', newline='') as file:
        writer = csv.writer(file)
        if os.path.getsize(mse_file_path) == 0:
            writer.writerow(['Evaluation'] + mse_keys)
        # add MES values for current evaluation
        writer.writerow([evaluation] + mse_values)

    print(f"MSE values for evaluation {evaluation} have been saved.")

# copy input ile
def copy_input_parameters(src_filename, params, result_dir, index):
    dest_filename = f"{params.test_name}_input.json"
    full_path = os.path.join(result_dir, dest_filename)
    try:
        os.makedirs(result_dir, exist_ok=True)
        with open(src_filename, 'r') as src_file:
            parameters = json.load(src_file)
            new_data = parameters.copy()
            # update arrays to only include the specified index
            try:
                new_data['PlasticYield']['value'] = parameters['PlasticYield']['value'][index]
                new_data['Alpha']['value'] = parameters['Alpha']['value'][index]
                new_data['Beta']['value'] = parameters['Beta']['value'][index]
                new_data['Gamma']['value'] = parameters['Gamma']['value'][index]
                new_data['YoungsModulus']['value'] = parameters['YoungsModulus']['value'][index]
                new_data['PoissonRatio']['value'] = parameters['PoissonRatio']['value'][index]
            except IndexError:
                print(f"Index {index} is out of range for one of the parameter arrays.")
                return

        with open(full_path, 'w') as dest_file:
            json.dump(parameters, dest_file, indent=4)

        print(f"Copied input parameters in '{full_path}'.")

    except Exception as e:
        print(f"Error when copying input parameters: {e}")

    return None

# copy reference data
def copy_md_file(src_filename, job_name, result_directory, input_directory):
    job_directory = os.path.join(str(result_directory), job_name)
    os.makedirs(job_directory, exist_ok=True)
    dest_filename = os.path.join(job_directory, os.path.basename(src_filename))
    src_filepath = os.path.join(input_directory, src_filename)
    try:
        with open(src_filepath, 'rb') as src_file:
```

```
            with open(dest_filename, 'wb') as dest_file:
                dest_file.write(src_file.read())

        print(f"Die Datei '{src_filename}' wurde erfolgreich nach '{dest_filename}' kopiert.")

    except Exception as e:
        print(f"Fehler beim Kopieren der Datei: {e}")

    return None

# svae ODB
def save_odb():
    # loop over all jobs in job_names
    job_names = list(mdb.jobs.keys())

    for job_name in job_names:
        if job_name in mdb.jobs:
            job_to_run = mdb.jobs[job_name]
            job_to_run.submit()
            job_to_run.waitForCompletion()
        else:
            print(f"Job '{job_name}' not found in MDB.")
            continue

        odb_file_name = f"{job_name}.odb"
        current_directory = os.getcwd()
        full_path = os.path.join(current_directory, odb_file_name)
        if os.path.isfile(full_path):
            odb = openOdb(full_path)
        else:
            print(f"ODB '{odb_file_name}' not found in directory: '{current_directory}'.")
            continue
        odb.save()
        odb.close()
        print(f"Odb for job {job_name} has been saved in {current_directory}.")

    return None

# write scipy.minize message
def write_rmse_message(RMSE, output_filename, directory):
    full_path = os.path.join(directory, output_filename)
    with open(full_path, 'w') as file:
        file.write(f"Optimization result:\n")
        file.write(f"Final RMSE value: {RMSE.fun}\n")
        file.write(f"Optimized parameters: {RMSE.x}\n")
        file.write(f"Optimization success: {RMSE.success}\n")
        file.write(f"Message: {RMSE.message}\n")
        if 'allvecs' in RMSE:
            file.write(f"All optimization paths: {RMSE.allvecs}\n")

    print(f"RMSE results have been saved to {output_filename}")

# main function
def main():

    print('###\n')
    print('NEW SCRIPT RUN\n')
```

```
    print('###\n')

    top_dir = '/calculate/Ziegler/Abaqus_Scripting/MultiJobAnalysis'
    result_directory = '/calculate/Ziegler/Abaqus_Scripting/MultiJobAnalysis/PA_Results'
    input_directory = os.path.join(top_dir, 'PA_Input')
    parameter_input = "AbaqusInput.json"
    parameter_input_path = os.path.join(input_directory, parameter_input)
    cube_parameters = read_parameter_file(parameter_input_path, input_directory)
    print(cube_parameters)

    # loop over all initial value combinations
    for i in range(cube_parameters.number_of_params):
        mdb = Mdb()
        work_directory_name = f"{cube_parameters.test_name}_{i}"
        work_directory = os.path.join(result_directory, work_directory_name)
        os.makedirs(work_directory, exist_ok=True)
        model_directory = os.path.join(work_directory, cube_parameters.model_name)
        os.makedirs(model_directory, exist_ok = True)
        os.chdir(model_directory)
        copy_input_parameters(parameter_input_path, cube_parameters, work_directory, i)
        mdb_cubes_dict = {}
        md_data_dict = {}
        # create Abaqus model for all load parameters and load cases
        for filename in cube_parameters.md_data_dict.keys():
            for direction, properties in cube_parameters.prescribed_directions.items():
                if properties['active']  != 0:
                    md_data_path = os.path.join(input_directory, filename)
                    mdb_cube = MDBCube(cube_parameters,md_data_path, model_directory, direction)
                    mdb_cube.create_cube(i)
                    job_name = mdb_cube.create_job(filename, direction)
                    mdb_cube.update_boundary_condition(direction)
                    mdb_cube.update_increment_size()
                    mdb_cube.save_mdb()
                    mdb_cubes_dict[filename] = mdb_cube
                    md_data = MDData(filename, input_directory)
                    md_data_dict[job_name] = md_data
                    copy_md_file(filename, job_name, work_directory, input_directory)

        print('Result directory', work_directory)

        lck_files = glob.glob('*.lck')
        if lck_files:
            for file in lck_files:
                try:
                    os.remove(file)
                except Exception as e:
                    print(f'Error when deleting {file}: {e}')
        else:
            print('No .lck-file found.')

        # scaled material parameter boundaries
```

```
        bounds =[(0,1), (0,1), (0,1), (0,1)]
        scaled_material = [cube_parameters.plastic_yield_scaled[i],
        cube_parameters.alpha_scaled[i], cube_parameters.beta_scaled[i],
    cube_parameters.gamma_scaled[i]]

        evaluation_count = [0]
        # call scipy.minize()
        RMSE = minimize(optimization_multiple_stresses, scaled_material,
    args=(cube_parameters,  md_data_dict, work_directory, evaluation_count
    ),
            method='nelder-mead', bounds = bounds, options={'disp': True,
     'return_all': True, 'maxiter': cube_parameters.iteration_number})

        print(RMSE)
        write_rmse_message(RMSE, 'rmse_message.txt', work_directory)
        save_odb()
        mdb.save()
        mdb.close()

if __name__=="__main__":
    main()
```