Aalto-yliopisto
Aalto-universitetet
Aalto University

Bachelor's Programme in Computational Engineering

# Crystal Plasticity Parameter Optimization

Automated optimization of material parameters for DAMASK crystal plasticity simulations.

---

**Ilari Pajula, Nam Anh Chu, Lucas Krieger**

Supervisor: Junhe Lian
Advisor: Wenqi Liu

**Bachelor's Major Project**
**2021**

# Table of Contents

# Abstract

Crystal plasticity (CP) modelling has now become a powerful tool which can deliver real value for purpose-built materials. Material simulation methods have industry wide applications for accelerating product development, and increasing efficiency in materials research. In this paper we implement a program-driven control of CP parameter optimization that runs on the CSC computing cluster. We adapt and build on previous implementations of constitutive material parameter optimization processes by applying a genetic algorithm (GA) that optimizes four material parameters with the help of a multi-layer perceptron (MLP) neural network response surface model. Additionally, this paper implements a reliable evaluation criterion for the curve fitting quality using Python. We conclude that our process is capable of arriving at accurate solutions through calidating our approach on a test data with known ground truth labels. This paper closes on mentions of how futher machine learning approaches could be used to futher explore the domain of CP parameter optimization.

# 1 Introduction

Almost everything used in our lives involves materials. Built structures, mobile phones, and medical equipment use specialized materials that have been carefully engineered for their end purpose. Recent advances in materials modelling have improved accuracy, realism, and predictive capabilities for engineers designing materials for new use cases. Predictive modelling has now become a powerful tool which can deliver real value for purpose-built materials. Material simulation methods, such as crystal plasticity modelling, have industry wide applications for accelerating product development, and increasing efficiency in materials research.

Crystal plasticity (CP) modelling refers to a computational technique using crystallographic anisotropy in modelling the mechanical behavior of polycrystalline materials. CP models are crucial for understanding and predicting the evolution of underlying material microstructure and anisotropic stress–strain responses of materials. In CP modelling that mechanical behavior is determined through a set of material parameters, which must be carefully calibrated to model some specific behavior. Parameter calibration refers to the process of optimizing these parameters. In this paper we present an implementation for an automated optimization of CP parameters operating with on DAMASK simulation softaware.

The main tasks of this project include writing a literature review on fitting quality evaluation and parameter optimization approaches, and implementing a program-driven control of CP parameter optimization that runs on the CSC computing cluster. The main objectives are to implement a reliable evaluation criterion for the curve fitting quality using Python, and to implement a CP parameter optimization program. Overall, the work aims to serve as preliminary research for future implementations of other automatic optimization processes for various applications in the materials science field.

This paper will start with a review of existing research on the topic of optimization in machine learning. This section will be dedicated to exploring different optimization methods such as gradient based approaches and derivative-free approaches. Additionally, this section covers response surface methodology, which are statistical approximation methods used to reduce the computational complexity of an

optimization approach. The section will discuss the advantages and disadvantages of these approaches, and evaluate the suitability of each one for CP parameter optimization.

Research on objective functions (eg. loss functions or error functions) is performed to build an understanding of fitting quality evaluation methods. In that section we review popular machine learning objective functions. Namely, root mean squared error (RMSE), mean squared error (MSE), weighted MSE, and $R^2$ for regression problems. We discuss the advantages and disadvantages of the introduced objective functions and discuss which ones could be suitable for CP parameter optimization.

The review extends to existing applications to parameter optimization in materials science. Namely, the literature review on CP parameter optimization revealed previous work on an almost identical topic. Sedighiani et al. [2] details a modern approach to the specific material parameter optimization problem using a genetic algorithm (GA) for parameter optimization. Their approach was a large influence for the optimization process presented in this paper.

Finally we present our methodology and implementation for an automated optimization of CP parameters, along with a evaluation criteria for simulated stress-strain curves. A validation of the optimization approach with ground-truth parameters performed, and the performance of the optimization is discussed in the results section of the report.

# 2 State of the art

## 2.1 Crystal plasticity modeling

Crystal plasticity is a multi-scale process starting at the atomic scale where dislocation cores, the regions near dislocation lines, control local properties such as glide planes, dislocation mobility, cross-slip, and nucleation processes [2]. Crystal plasticity modeling is a computational technique where the objective is to model the relationship between stress and strain of material from the physics at the crystal level. One widely used and simple model is the phenomenological constitutive model [3] for crystal plasticity. The constitutive laws in a CP model relate the kinetics of deformation to physical material behavior, while obeying upper and lower bounds from physics-based models [1]. These constitutive laws depend on several adjustable parameters, which are the subjects for the optimization in this paper.

The phenomenological constitutive model mostly uses a critical resolved shear stress, as state variable for each slip system. In the model, the shear on each slip system occurs according to:

$$\dot{\gamma}^{\alpha} = \dot{\gamma}_0 \left|\frac{\tau^{\alpha}}{\tau_0^{a}}\right|^{n} \operatorname{sgn}(\tau^{\alpha}) \qquad \text{Eq.1}$$

Where $\tau_0$ is the slip resistance, $\dot{\gamma}$ is the reference shear rate, and $n$ determines the strain rate sensitivity of slip. The influence of any set of slip system, $\alpha'$, on the hardening behavior of a slip system $\alpha$ is given by:

$$\dot{\tau}_0^{\alpha} = \sum_{a'=1}^{N_x} h_{a\alpha'} \left|\dot{\gamma}^{\alpha'}\right| \qquad \text{Eq.2}$$

Where $h_{a\alpha'}$ is the hardening matrix, which empirically captures the micromechanical interaction between different slip systems:

$$h_{a\alpha'} = q_{\alpha a'} \left[h_0 \left(1 - \frac{{\tau_0}^{\alpha}}{\tau_{\infty}}\right)^{\alpha}\right] \qquad \text{Eq.3}$$

Where $h_0$, a, $\tau_{\infty}$, are slip hardening parameters, which are assumed to be the same for all 12 slip systems of the {111} <110> type in FCC crystals. The parameter $q_{\alpha a'}$ is a measure for latent hardening, and its values is taken as 1.0 for coplanar slip systems $\alpha$, and $\alpha'$ and 1.4 otherwise. The hardening behavior is anisotropic.

These constitutive equations are in the class of partial differential equations, which often cannot be solved analytically and require numerical methods to find solutions for. Düsseldorf Advanced Material Simulation Kit (DAMASK) is a multi-physics crystal plasticity simulation package that implements a variety of constitutive materials models, including the phenomenological variant [3]. Their simulation kit implements a spectral solver using a Fast-Fourier-Transform (FFT) method to numerically find static solutions for the phenomenological constitutive material model. DAMASK CP simulations can be run given a description for the geometry of a material, loadings that act on the material, and defined constitutive parameters.

## 2.2 Material parameter optimization

Finding optimal constitutive parameters requires an iterative approach that adjusts the parameters until simulation results are similar enough to some experimental target. For small models with few adjustable parameters, it is feasible to find optimal parameter values through the process of trial and error. However, when the complexity of material models grows and multiple parameters need to be optimized, trial-and-error approaches need to be replaced with more advanced optimization methods [2].

For these types of iterative optimization problems gradient-based optimization methods such as gradient descent, and stochastic gradient descent are usually utilized [7,8,9]. However, from other gradient-based material parameter optimization studies [7] some of the drawbacks of this approach become apparent. Gradient-based methods are sensitive to initial parameter guesses which can result in the optimization process converging to some local minima, resulting in less-than-ideal solutions. The tendency to converge to local minima would be exemplified in the complex optimization space the constitutive laws describe. Experimentally obtained strain – stress responses might also introduce significant noise into the objective function, which would further complicate the optimization process.

An optimization approach that is more robust against initial guesses, converging to bad local minima, and noisy objective functions is needed. To these criteria, Sedighiani et al. 2020, [1] proposed a specialized method of using a response surface and a genetic algorithm to optimize parameters of the phenomenological constitutive model. Their

problem was very similar to the research topic in this paper, even using the same DAMASK FFT Spectral Solver for CP simulations.

Their optimization process is centered around the combined use of a response surface and the genetic algorithm to optimize material parameters. In their method, the response surface works as a statistical approximation of the underlying constitutive model, then GA is used to find the optimal solutions based on the response surface.

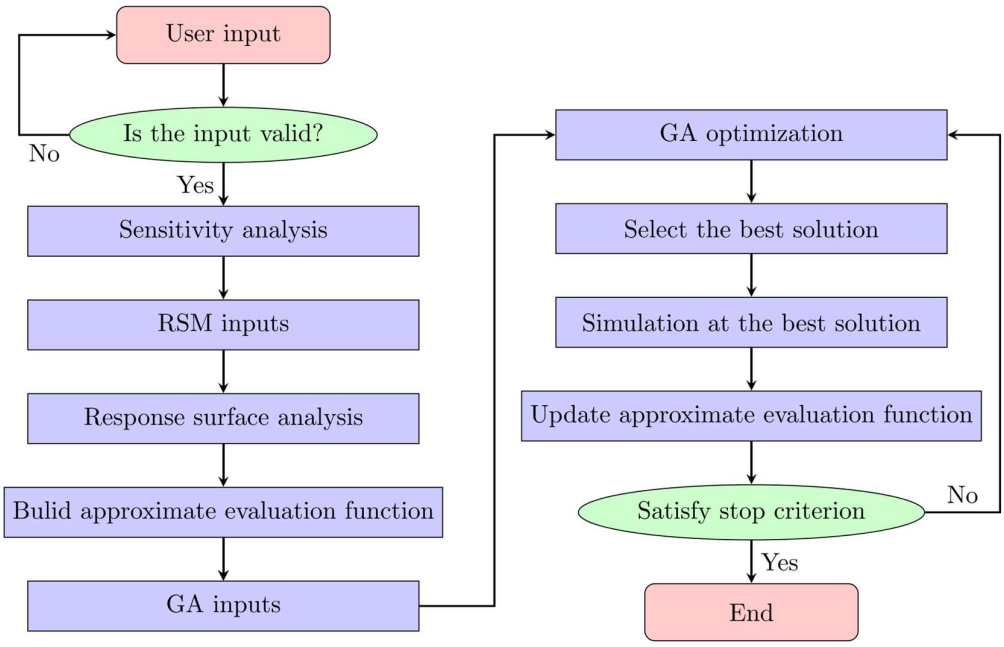



**Figure 1: Workflow of the optimization process introduced in Sedighiani et al. [2]**

**Figure 1** outlines the optimization process utilizing a response surface and a GA. In the first step, user inputs are provided. These consist of an experimentally obtained flow curve that is defined as the target of the optimization, and constraints for each parameter to be optimized. The following steps first involve running a set of initial simulations (each one with unique parameters). The paper also introduces a method for choosing these initial simulations. Next, fitting a response surface regression model to approximate the stress response from the initial simulations. Specifically, a second degree polynomial surface is used for the approximation. The response surface is then used to build an approximate evaluation function. The third step is the optimization process, where a GA is applied on the response surface to find the parameters that

best fit the experimental target flow curve. An additional simulation is then performed with the parameter values of the best solution. If that simulation is a good fit for the experimental target, then the optimal parameters have been found and the process terminates. If the additional simulation has a poor fit, it is used to update the response surface and the process restarts.

Sedighiani et al. were fundamental to understanding how an efficient material parameter optimization process needs to be formulated. As such, the approach presented in this paper adopts a significant amount of the methods, tools, and prior knowledge from their work. Specifically, this paper adopts the optimization method using a response surface coupled with a GA for which the workflow is presented in Figure 1.

## 2.3 Objective functions selection

Objective functions (sometimes called loss functions, cost functions, or error functions) are functions that return a scalar value that represents a difference between estimated and true values in a model. In curve-fitting optimization problems, the aim is to minimize the objective function. With the context of the research topic, the optimization of crystal plasticity parameters, the objective function should provide a numerical measure for the goodness of fit for a simulated stress–strain curve (response) to an experimentally measured stress–strain curve. For this problem, the most popular regression objective functions were investigated.

A machine learning survey paper [5] comparing a vast variety of different regression models for a large amount of different test datasets used as its primary performance metrics Root Mean Squared Error (RMSE), $R^2$, and Mean Squared Error (MSE). Of these, $R^2$ is only applicable to linear models where it provides a relative measure of fit between variables, in other words $R^2$ is the fraction of the total sum of squares that is 'explained by' the regression. RMSE and MSE have direct relationships to the $R^2$ but instead provides an absolute measure of fit. Both RMSE and MSE are suitable for non-linear models, and will be discussed in this paper.

Materials science specific implementations for loss functions were scarce, and there were not many relevant research articles. However, one paper [6] about estimating material parameters using inverse modelling and their application to sheet metal forming simulations discussed using MSE and a weighted MSE for their approaches. RMSE, MSE, and weighted MSE are objective function measures that are popular and are safe to use as a general approach to a problem like measuring the quality of fit. These three measures have been successfully tested on similar problems, and could be applied to the problem of optimizing crystal plasticity parameters.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2} \qquad \text{Eq.4}$$

$$MSE = \frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \qquad \text{Eq.5}$$

$$Weighted\ MSE = \frac{1}{n}\sum_{i=1}^{n} w_i(y_i - \hat{y}_i)^2 \qquad \text{Eq.6}$$

When constructing a objective function an important thing to consider is how complex or noisy to the function will be to minimize. Of the objective functions in equations 4,5, and 6 all are similar, with a few key differences. MSE and weighted MSE are more sensitive to outliers and larger deviations than RMSE due to the squared term, and thus could cause additional noisy and complexity during optimization. The weighted MSE also requires the additional weights, $w_i$, to be defined for each datapoint. The weights allow for a customized loss function in specialized problems, but come with the disadvantage of needing to find suitable weights. The RMSE is a generally more stable objective function due the square root. Its only disadvantage is that the square root is computationally intensive to calculate when many objective function evaluations are needed.

## 2.4 Response surface methodology - multi-layer perceptron

The fitness of a chromosome can simply be estimated by performing a simulation and comparing the output with the experimental results through the above objective functions. However, when the evaluation function is computationally expensive, such as the case is for CP simulations, a GA becomes unreasonable; because of their time-

consuming nature induced by the substantial number of evaluations. To evaluate the fitness of the chromosomes, a cost-effective approximate evaluation function based on the response surface methodology (RSM) is employed. This response can be described as a statistical technique that presents the relationship between explanatory variables with one or more response variables. [1]

This method involves fitting a polynomial to approximate the responses variables and it is especially useful when “the effects of an explanatory variable are dependent on the level of the other explanatory variables” [1]. In this paper, the response variable, explanatory variables, and the independent input variables are stress, adjustable constitutive parameters are applied strain.

This research paper proposes using a multi-layer perceptron (MLP) as the response surface function. An MLP is a class of feedforward artificial neural networks consisting of an input layer, one or more hidden layers, and an output layer. With the reasoning that MLPs can model complex, non-linear, data one is implemented for modeling the strain-strain response given 4 material parameters. At a higher level, this could be mathematically formulated as.

$$MLP(\tau, \tau_s, \mathrm{h}_0, \alpha) = \tilde{\boldsymbol{\sigma}} \qquad \text{Eq.7}$$

Where $\tau, \tau_s, \mathrm{h}_0, \alpha$ are the parameters to be optimized, and $\tilde{\boldsymbol{\sigma}}$ is the MLPs prediction for the stress response at those specfic strain values.

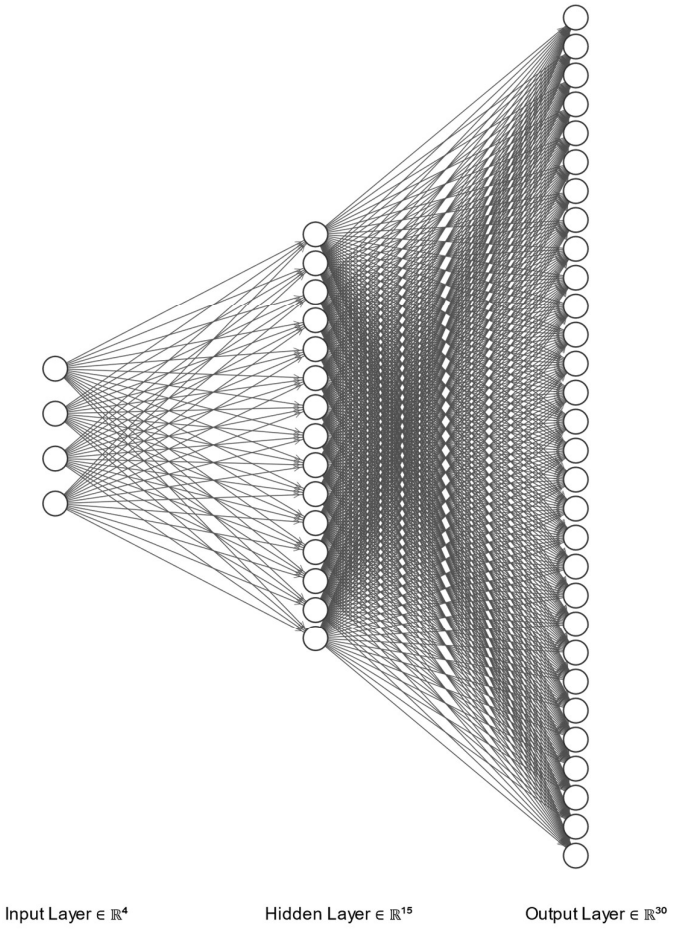



**Figure 2: Structure of the MLP model.**

Each circle in Figure 2 is a neuron. The leftmost layer is the input layer that consist of the neurons for the input data, in the context of the problem they represent the 4 CP in the optimization problem. Each neuron in the hidden layer is a weighted linear summation of the neurons in the previous layer followed by a transformation by applying an activation function (ReLU in this paper).

An activation function is a function which outputs a value depending on if the inputs pass a given threshold. The output value is small if the inputs do not pass that threshold, and large if the inputs exceed that threshold. If the inputs of the function exceeds the threshold the activation function “activates”. ReLU is a non-linear activation function that is commonly used in multi-layered neural networks. It outputs

the maximum value between zero and the input value. The output value is negative if the input value is less than zero and positive if larger or equal to zero.

$$ReLU(x) = \max(0, x) \quad \text{Eq.8}$$

The output layer is the response, in this context each output neuron represents a stress value ($\tilde{\boldsymbol{\sigma}}$) at a corresponding strain. Figure 3 has 30 neurons in the output layer which each represent a stress value at some strain level. The output receives the values from the hidden layer and transforms the values using the linear summation. For the MLP regression model used in this paper, the activation function in the output layer is an identity function, i.e. no activation function is applied in the last step.

The general mathematical formulation of the layers of the MLP regression model would be described as.

$$\mathbf{f}^{(1)}(\mathring{\mathbf{x}}) = \mathbf{ReLU}\left(\mathbf{W}_1^T \mathring{\mathbf{x}}\right) \quad \text{Eq.9}$$

$$\hat{\mathbf{y}} = \mathbf{f}^{(2)}(\mathbf{f}^{(1)}(\mathring{\mathbf{x}})) = \mathbf{W}_2^T \mathbf{f}^{(1)} \quad \text{Eq.10}$$

Where $\mathring{\mathbf{x}}$ are the inputs into the model (parameters $\tau, \tau_s, \mathrm{h}_0, \alpha$ in this study), $\mathbf{W}_1^T$ and $\mathbf{W}_2^T$ are the edge weights of the corresponding layers, $\mathbf{f}^{(1)}$ is the transformed outputs at the hidden layer neurons, and $\hat{\mathbf{y}}$ (equivalently $\mathbf{f}^{(2)}$) is the transformed outputs at the output layer. ReLU is the hidden layer's activation function.

Similar to advanced linear models, MLP uses the sum of the squared error loss and an additional L2-norm regularization term for a loss function. The optimization problem to fit the MLP is to identify the the edge weights $\boldsymbol{\beta}$ that minimizes:

$$\mathrm{argmin}_\beta \overbrace{\frac{1}{2} \| \hat{\mathbf{y}} - \mathbf{y} \|_2^2 + \underbrace{\frac{\alpha}{2} \| \boldsymbol{\beta} \|_2^2}_{\text{regulization term}}}^{\text{loss function}}, where\ \boldsymbol{\beta} = [\mathbf{W}_1^T, \mathbf{W}_2^T] \quad \text{Eq.11}$$

Where $\alpha$ is the learning rate for optimization, since the MLP is fit with some implementation of backpropagation with stochastic gradient descent:

$$\boldsymbol{\beta} \leftarrow \beta - \gamma \left(\alpha \frac{\partial R(\beta)}{\partial \beta} + \frac{\partial \mathrm{lo}}{\partial \beta}\right) \quad \text{Eq.12}$$

Where $R(\beta)$ is the regulization term.

## 2.5 Genetic algorithm

The underlying principle is a randomized search technique based on the survival of the fittest. Genetic algorithms are inspired by Darwin's theory of evolution and because of this it uses the following biological terminology [1]:

Genes – adjustable parameters of the objective function.
Chromosome – A solution generated by the genetic algorithm, composed of genes.
Population – A collection of chromosomes

A brief overview of the genetic algorithm steps is given below:
First, several chromosomes are randomly selected to produce an initial population. These chromosomes will be evaluated by the previously defined objective functions, some chromosomes in the population will then mate (crossover) resulting in the production of offspring whose genes are formed by the combination of the parents' genes. Moreover, mutations can arise at any moment, the rate at which crossovers and mutations occur is defined by the crossover rate and the mutation rate. The chromosomes that have the highest probability of being selected for the next generation are the ones that have the highest fitness value. After multiple generations, the chromosomes will converge on the solution to the problem. [4]

Furthermore, each step has more detail, given below:
Step 1: Initial Population – Each gene in the population is selected randomly based on selected ranges, which are based on the physical bounds of parameters. If the range of a parameter is larger than one order of magnitude, the parameter value is selected from numbers spaced evenly on a logarithmic scale. [4]

Step 2: Objective Functions and Fitness – The fitness of the chromones in the population is determined using the fitness function, the weighted sum of the objective functions. [4]

Step 3: Selection – Chromosomes with higher fitness have a higher chance of being selected to be a parent for the next generation. These are selected using a rank-based

wheel approach where chromosomes are first ranked by their fitness and then each chromosome gets a weight based on its rank among the population. [4]

Step 4: Crossover – Crossover is the operation in which parent chromosomes mate and combine genes to form offspring. According to Sedighiani et. All, the single-point crossover method was used, where the division chromosomes of both parents are divided into two parts. The division is done at the same point for both parents which is selected randomly as long as each parent contains one gene. The genes are then combined to form offspring. If the parents' genes are too similar, then no action will be taken. [4]

Step 5: Mutation – Mutation will randomly change one or more genes in a chromosome, determined by the mutation rate. The aim of mutations is to make the population diverse, thus helping it avoid getting stuck at local minima's; however, too high of a mutation rate and the algorithm becomes a random search algorithm. [4]

Step 6: Elitist selection – Elitist selection ensures a small proportion of the best chromosomes will be selected for the next generation without any changes. They are, however, eligible for parent selection. Elitist selection will rank the chromogens for each objective function and then select the ninety-ninth percentile of each objective function. [4]

Step 7: New generation – The offspring generation will consist of chromosomes that are Elitist chromosome, which is less than or equal to 5%; offspring with mutations, which is roughly 1%; and the rest, which are offspring without mutations. [4]

Step 8: Stop criterion – The GA will stop when one of the following criteria are met: 70% of the chromosomes have become equal, no improvement has been made in the best solution after a set amount of generations; or if the maximal number of generations has been reached, in this paper that number is 100. [4]

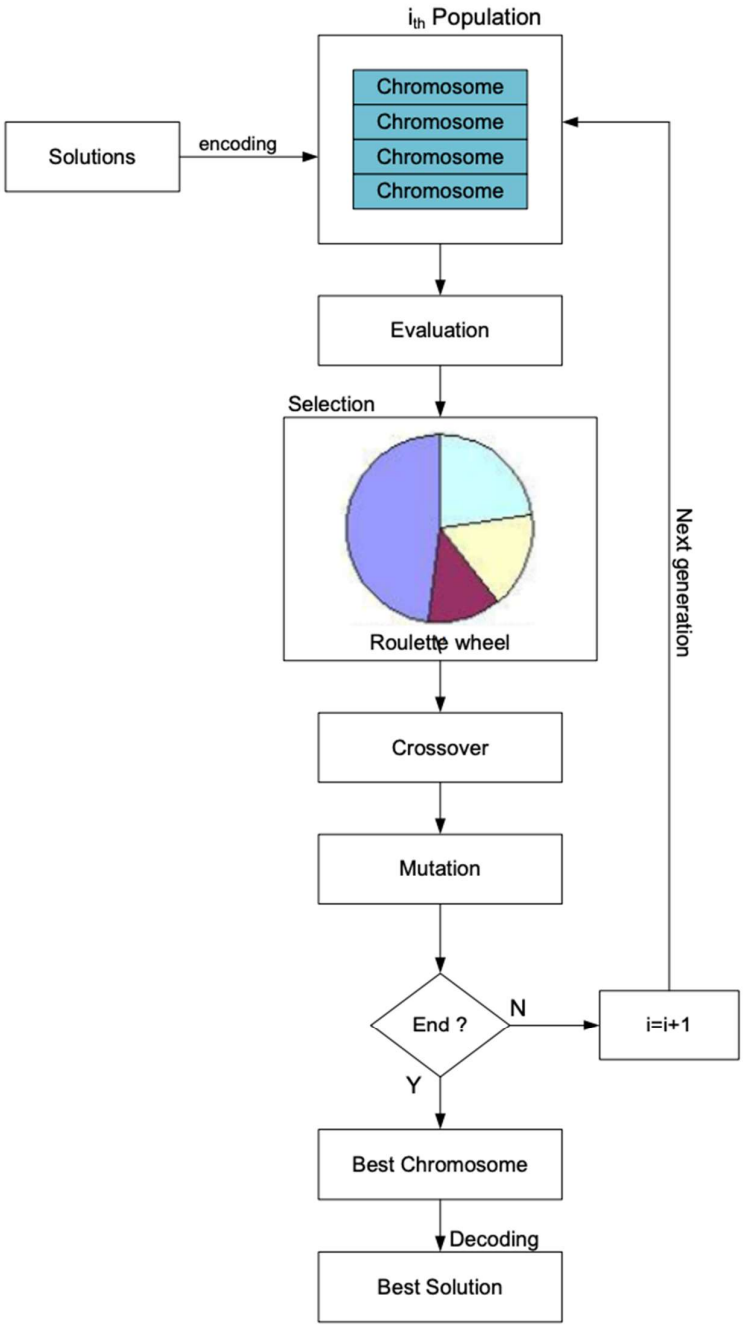



**Figure 3: Genetic algorithm flowchart [4]**

Figure 3 showcases the process that a simple genetic algorithm undergoes. The solutions to population segment is what was discussed in Step 1. After that the Evaluation is what was discussed in Step 2. This is followed by selection, which is Step 3. Next we have Crossover and Mutation, which are Steps 4 and 5. Next the program

checks if the stop criterion (Step 8) has been met, if it has been met the alrogithm then finds which chromosome is the best solution and decodes that, if the stop criterion has not been met the program goes back to Step 1, but with a population that is different to the initial, as it has undergone crossovers and mutations. In Figure 3, i=i+1 is used to count the number of generations that occurred if it reaches 100, that is a stop crietion and hence the algorithm will end.

## 2.6 Hardware, tools, and environment

For the optimization process presented in this paper, the Finnish CSC computing cluster was used to run DAMASK CP simulations. Specifically, Puhti, a general-purpose computing cluster. To run the large amounts of computationally demanding CP simulations required for the optimization process the optimization program presented in this paper is built upon Slurm, a Linux-based job-scheduler, and Python.

With Slurm, simulation jobs are submitted to run in parallel on separate computing nodes in Puhti. This allows us to run up to 35 CP simulations simultaneously, saving a significant amount of time. Slurm jobs are submitted by running Bash scripts.

Python is a high-level programming language that has good support for scientific computing libraries. The optimization program presented in this paper makes heavy use of the PyGAD which is used for GA optimization; Sci-Kit Learn, which is used for response surface modeling; and Numpy, which is used for data processing libraries. Additionally, Python is used to automatically submit new simulations from within our optimization program by running the Bash scripts for Slurm.

# 3 Methodology

This section introduces an iterative optimization process for CP parameters in a phenomenological model. This section of the paper will describe the overall workflow, different modules, and tools utilized in the optimization process presented in this paper.

First, the modules of the optimization process are introduced at a higher level. This is to introduce the different processes in the optimization process, and to explain how the optimization process converges. Second, the workflow of the optimization is discussed and explored in detail. This section will discuss the inputs and outputs of each module specifically and introduce the inner workings of each module individually. In the third section, the third-party tools and libraries used are discussed.

## 3.1 Modules

Since the optimization process is quite complex, this paper has divided the workflow into multiple modules. The term, module, is used to represent a significant section of the optimization process. This paper has spilt the optimization process into three separate modules.

1. The simulation module
2. The response surface module
3. The genetic algorithm module

The simulation module runs DAMASK crystal plasticity simulations. In this paper the phenomenological material model is used from DAMASK simulation software. The phenomenological model constructs a set of partial differential equations (PDEs) given material parameters, since very few PDEs can be solved analytically, numerical methods are used to find solutions. DAMASK employs a fast Fourier transform based spectral solver to find solutions.

The response surface module provides a function mapping from material parameter values to material stress values. Essentially, a response surface is a regression function that approximates a material parameter space to stress values. The purpose of using a response surface is to provide a continuous approximation of the

optimization space. The merit in using a response surface with optimization problems is the ability to minimize the amount of computationally intensive DAMASK simulations. Polynomial regression is a classic example of a response surface and is frequently used in literature [1,6]. This research uses an MLP as our response surface function as described in Section 2. An MLP is a class of feedforward artificial neural networks consisting of an input layer, one or more hidden layers, and an output layer. They have the ability to model complex, non-linear, data [6].

The genetic algorithm module is used to optimize material parameters. GA is a stochastic algorithm used in bounded optimization problems. As introduced in Section 2 it is an iterative optimization process that begins by initializing a set of bounded parameters, and then gradually converges to the optimal parameters through a repetitive application of mutation, crossover, inversion, and selection of parameters at each iteration. At each iteration, which is referred to as a generation in GAs, a fitness function is evaluated. The fitness function is a measure of fit between the current solution, and the target solution. A GA will run until the fitness function is good enough, or a specified number of generations is reached.

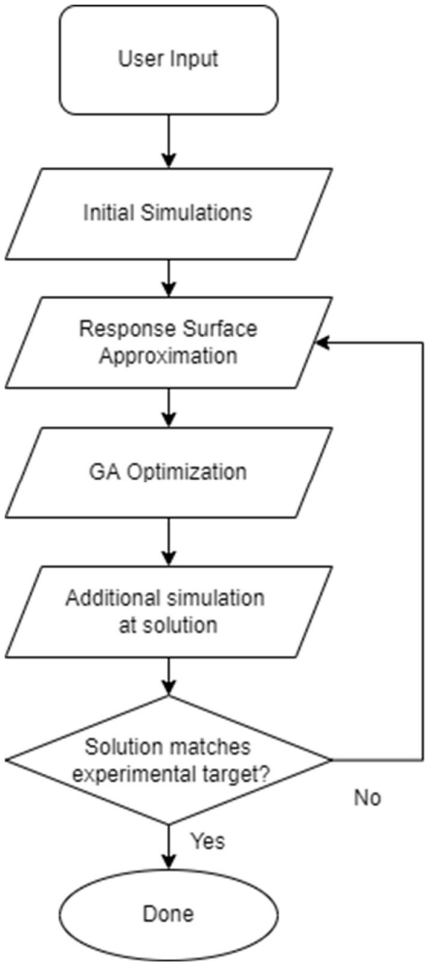



**Figure 4: General workflow of the optimization process.**

## 3.2 Workflow

Together, the modules of the optimization process work to provide an efficient way of optimizing material parameters. Efficiency in the workflow is essential as CP simulations are quite computationally intensive to run, and even with four parameters the optimization space can get extremely large.

At a higher level, the optimization process works as follows. The setup before the iterative optimization consists of two steps. In the first step, initial simulations are performed at select sets of material parameters and the corresponding stress values are obtained. In the second step, an MLP response surface is fitted to map those parameter values to the corresponding stress values.

The iterative process begins after the initial response surface has been fit. In the next step, the fitted MLP response surface is optimized with the GA module the set of parameters with the best fitness function is obtained. Next, a new CP simulation is performed at those optimal parameters and new stress values are obtained. That new parameter and corresponding stress pair is used to update the MLP response surface with the new data. This process repeats until the MSE from the new CP simulation is smaller than a predetermined value. The predetermined value should be adjusted for each optimization run, keeping in mind the required accuracy of the optimization. For the tests in this paper, we used MSE < 10.0 as a stopping criterion.

This process converges because with more simulations the response surface begins to accurately approximate the underlying relationship between the parameters and stress values. The GA algorithm can find the best solution from the response surface.

Figure 5 is a workflow diagram of the optimization process which highlights the inputs and outputs of the optimization process.

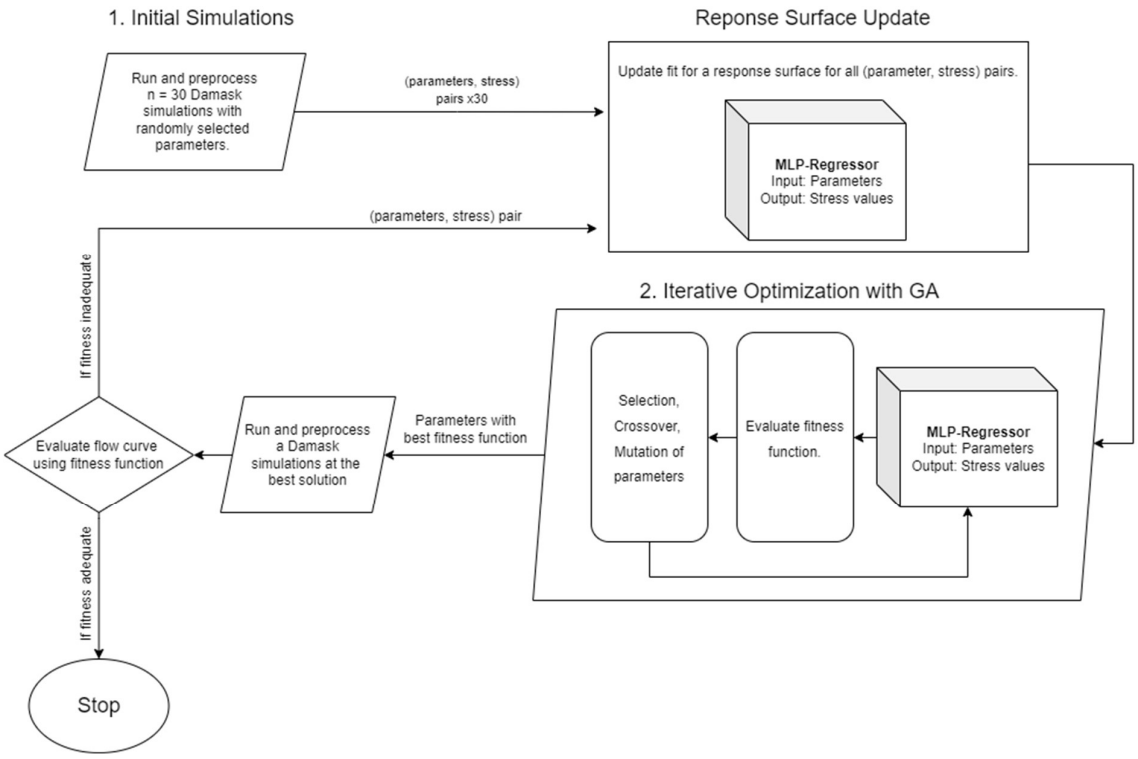



**Figure 5. Optimization Process Workflow**

## 3.3 Tools and hardware

The pipeline based on the workflow mentioned above is primarily implemented with Python 3.6 scripts. The PyGAD library is used to implement the GA module, Sci-Kit Learn library is used to implement the MLP response surface, and NumPy is used for all data processing. Python's OS library is used to automatically submit new simulations from within our optimization program by running Slurm BASH scripts. The DAMASK version 2.0.2 spectral FFT solver is used to run the simulations.

# 4 Evaluation criteria for curve fitting quality

This paper showcases the written formulas in the modules and processes of the methodology section. It is more useful to visualize the results of these functions and the simulated stress curves. This section will be spill into four sections, first it discusses what changes were made to the flow curves to get them ready to be used by the objective functions. Second, a discussion on which objective functions were selected. Third, a section which discusses how the objective functions were set-up. Fourth, and last, a disuccusion on the objective functions were validated.

## 4.1 Data preprocessing for flow curve

Running the DAMASK simulations on the CSC platform returns multiple .txt files with the true stress–true strain curves. In order to process these files and obtain the stress–true strain curve for the optimization pipeline, a data processing pipeline is necessary. For the data preprocessing pipeline, three important functions are implemented using the Pandas, NumPy, os, and glob libraries for data processing; and Sci-Kit Learn and SciPy for r-squared evaluation. Some of the most important functions are:

The “preprocess” function, where the path .txt file from running DAMASK simulations is obtained as input, extracts the stress–strain curve, calculate the plastic strain, evaluate, and trim stress - strain and stress - plastic strain curve based on R-squared. Finally, it returns the stress - plastic strain curve as NumPy arrays for faster computation.

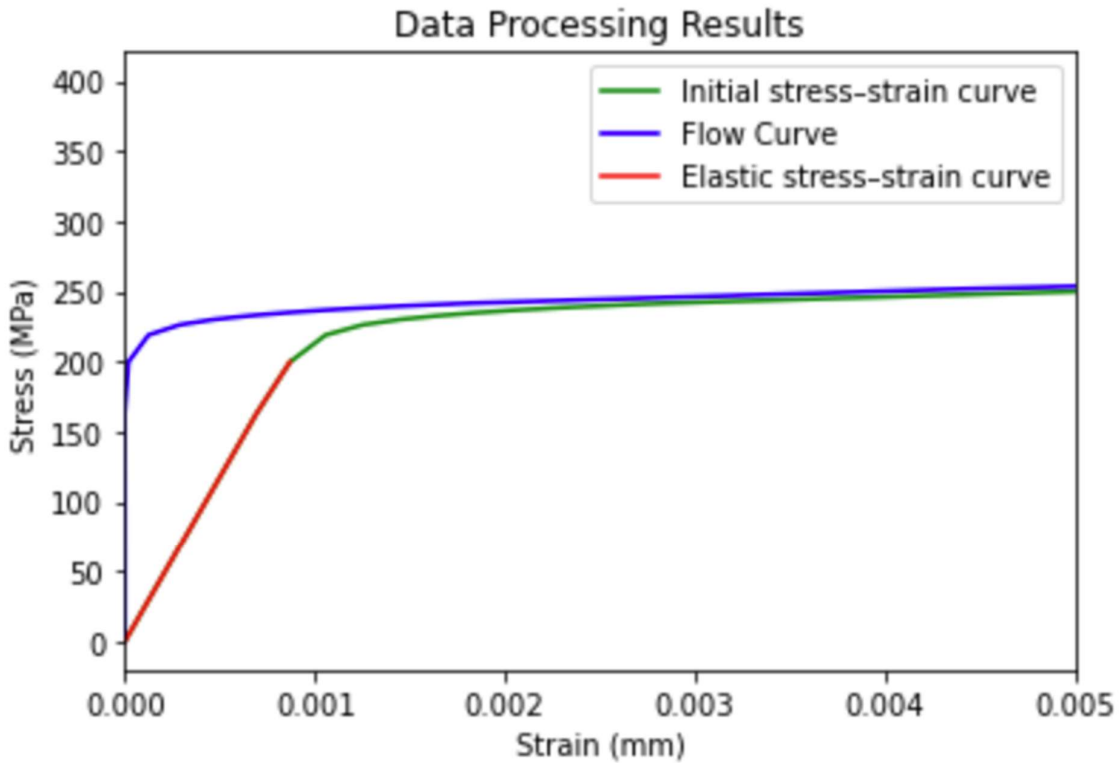
The “getTruePlasticStrain” function, which calculates the true plastic strain of a given stress–strain curve, which is getting the slope of the stress–strain curve, then minus the strain with the stress divided by the slope. The formula is denoted as followed:

$$\varepsilon^{pt} = \varepsilon^{t} - \varepsilon^{et} = \varepsilon^{t} - \frac{\sigma}{E} \qquad \text{Eq.9}$$

Where $\varepsilon^{pt}$ is true plastic strain, $\varepsilon^{t}$ is true total strain, $\varepsilon^{et}$ is true elastic strain, $\sigma$ is true stress, and E is Young’s modulus.

The “elastic” function, also the most interesting one in the pipeline. This function is implemented only to obtain the elastic part of any stress–strain curve using r-squared, which is not necessary for the entire project, but may prove useful in future implementations. The function initializes a trim value array, from 0.0005 to 0.02, with

0.00001 incrementations. Then it is flipped. After that, a while loop is constructed for continuously evaluating the trimmed (or elastic) stress–strain curve using the trim value array using r-squared. This is possible using a separate sixth function called “adjR” to fit the trimmed stress–strain curve into a one-degree linear line, then calculates the r-squared of the model. Also, the function can obtain the stress–strain curve with the elastic part removed.




R-squared = 0.9995460424322485

**Figure 6: Data Processing Results**

Figure 6 illustrates the result of the data processing pipeline, where the “SingleCrystalTest” stands for the true stress–true strain curve, the “Elastic” plot stands for the elastic portion of the stress – strain curve calculated using the “Elastic” function, which works really well with the r-squared of the elastic portion fitted into a linear first-degree model being 0.9996, and it covers nearly all of the elastic portion of the simulated stress–strain curve.

## 4.2 Objective functions selection

To evaluate the fit between an experimentally obtained and a simulated strain–stress response a set of objective functions that numerically measure the difference between the two curves needs to be defined. The aim of the optimization process is to minimize these objective functions. Four objective functions are implemented which are combined in a weighted sum to form an overall fitness function.

The first objective function used is the L2-norm, which gives the Euclidean distance between the experimental and simulated curves. In the equations below $\sigma_i$ represent experimental stress at strain i and $\widehat{\sigma_i}$ represents the simulated stress at strain i.

$$\mathrm{D1}(\boldsymbol{\sigma}, \widehat{\boldsymbol{\sigma}}) = \sqrt{\frac{\sum_{\mathrm{i}}^{\mathrm{n}} (\sigma_{\mathrm{i}} - \widehat{\sigma_{\mathrm{i}}})^2}{\sum_{\mathrm{i}}^{\mathrm{n}} \sigma_{\mathrm{i}}^2}} \qquad \text{Eq.10}$$

The first objective function only gives the absolute distance between the two curves, it does not consider positive (simulated curve above the experimental result) or negative (simulated curve below the experimental one) distances. Therefore, D2 takes the difference in slopes at a given point to calculate the loss.

$$\mathrm{D2}(\boldsymbol{\sigma}, \widehat{\boldsymbol{\sigma}}) = \sqrt{\frac{\sum_{i}^{n} \left(\sigma_{\mathrm{i}}' - \widehat{\sigma_{\mathrm{i}}'}\right)^2}{\sum_{i}^{n} \sigma_{\mathrm{i}}'^2}} \qquad \text{Eq.11}$$

To penalize the maximum difference in the two above loss functions D3 and D4 were implemented. They represent the maximum loss in the Euclidean distance and the maximum loss in the difference of slopes respectively.

$$\mathrm{D3}(\boldsymbol{\sigma}, \widehat{\boldsymbol{\sigma}}) = \max\left(\sqrt{\frac{(\sigma_{\mathrm{i}} - \widehat{\sigma_{\mathrm{i}}})^2}{\sum_{\mathrm{i}}^{\mathrm{n}} \sigma_{\mathrm{i}}^2}}\right) \qquad \text{Eq.12}$$

$$\mathrm{D4}(\boldsymbol{\sigma}, \widehat{\boldsymbol{\sigma}}) = \max\left(\sqrt{\frac{\left(\sigma_{\mathrm{i}}' - \widehat{\sigma_{\mathrm{i}}'}\right)^2}{\sum_{\mathrm{i}}^{\mathrm{n}} \sigma_{\mathrm{i}}'^2}}\right) \qquad \text{Eq.13}$$

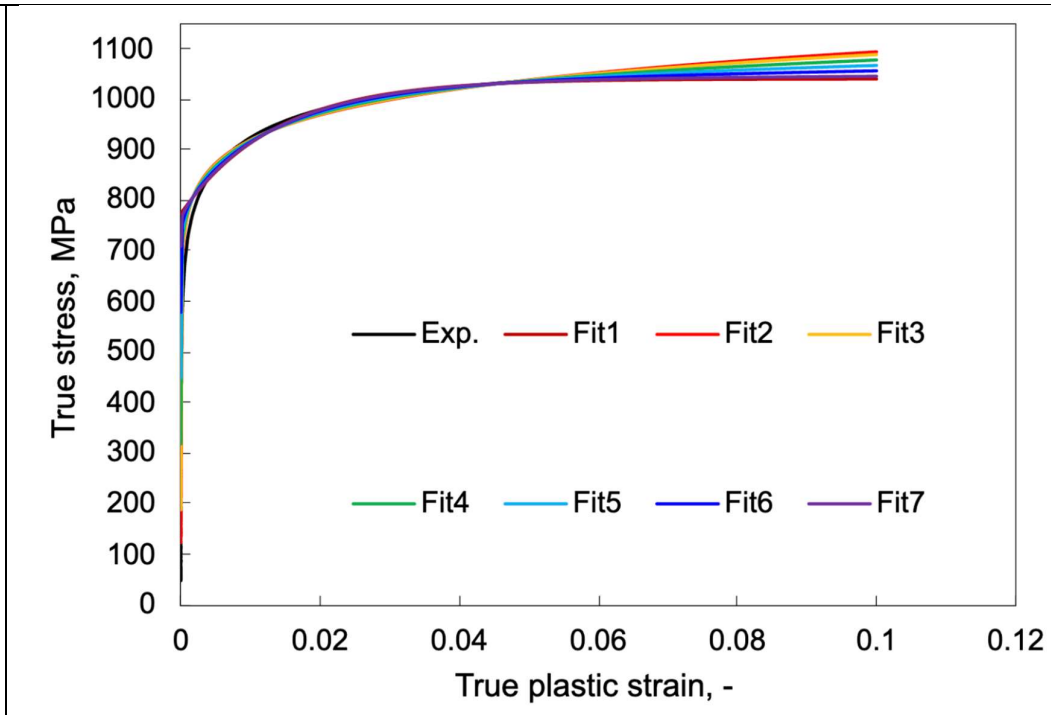
The fitness is then calculated by the weighted sum of the four objective functions. Where $w_i$ represents a chosen weight and $D_i$ represents an objective function. The fitness function is the criteria used to choose the best optimized simulated stress–strain curves.

$$\mathrm{Fitness}(\mathbf{w}, \boldsymbol{\sigma}, \widehat{\boldsymbol{\sigma}}) = \sum_{\mathrm{i}}^{4} w_i \cdot \mathrm{D_i}(\boldsymbol{\sigma}, \widehat{\boldsymbol{\sigma}}) \qquad \text{Eq.14}$$

## 4.3 Evaluation criteria setup

This section will go over how the evaluation criteria was implemented. Figure 7 showcases the data that is being worked with in this section. Figure 7 (a) shoecases the entirerity of the data; however, the data behaves rather inconsistently before the 0.02 mark and some flow curves have missing values. Thereforem the data was trimmed to start at the 0.02 mark, as can be seen in Figure 7 (b). Exp. can be seen in Figure 7, this is what the other flow curves are being compared to in the objective functions to give a fitness value. The objective functions were implemented in python and were based off of Eqs. 10-13, whilst the fitness equation was based on Eq. 14.

(a)



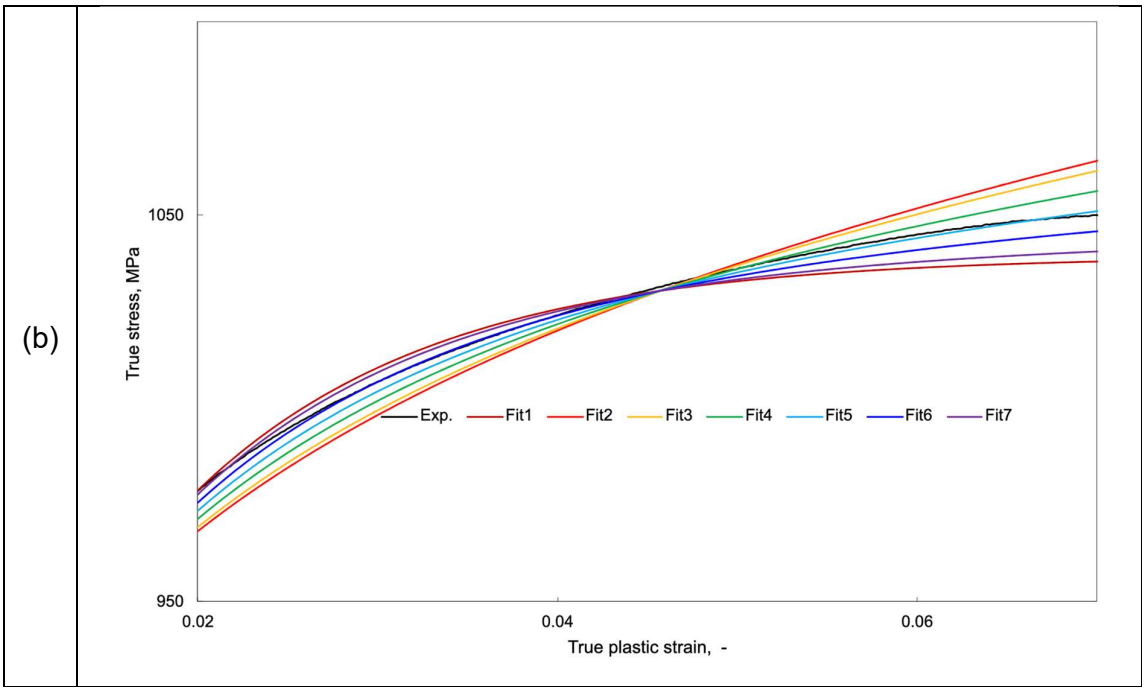



**Figure 7: Flow curves to set up the evlaution cireteria.**

**(a) Entire flow cuves (b) Zoom in with true plastic strain in the range of 0.02-0.07**

The results of the all objective functions are shown in Figure 8-11. Furthermore, a weighted sum of all four deviation functions were used for the fitness function, as shown in Figure 12.

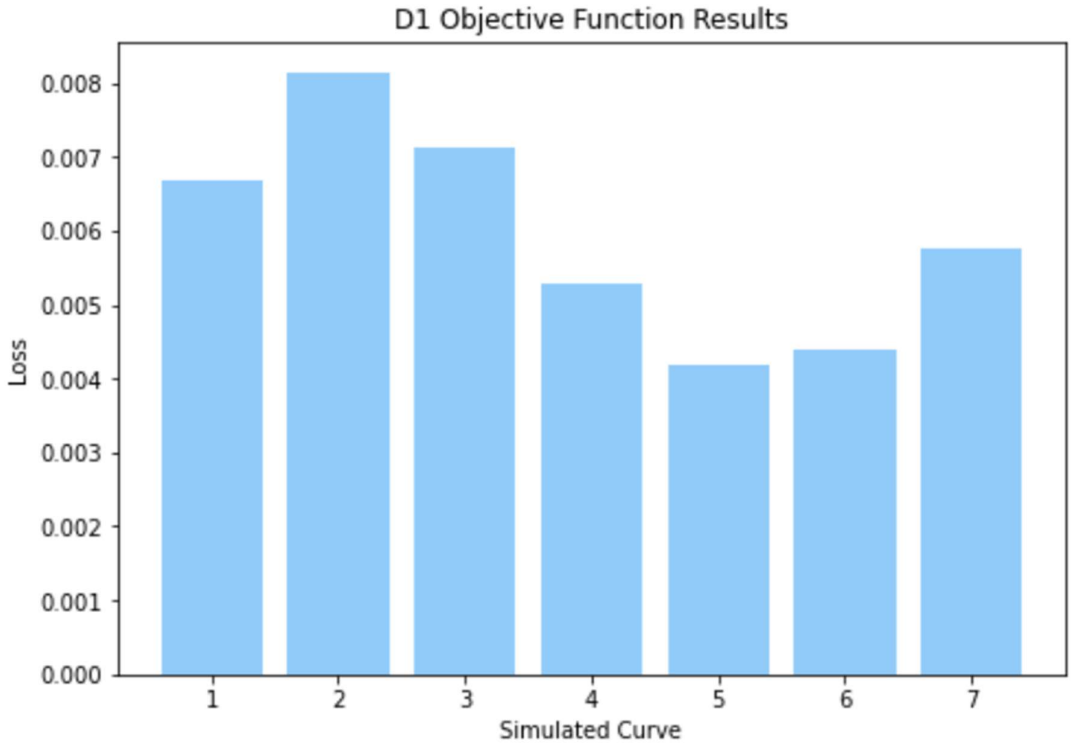


**Figure 8: D1 Objective Function Results**

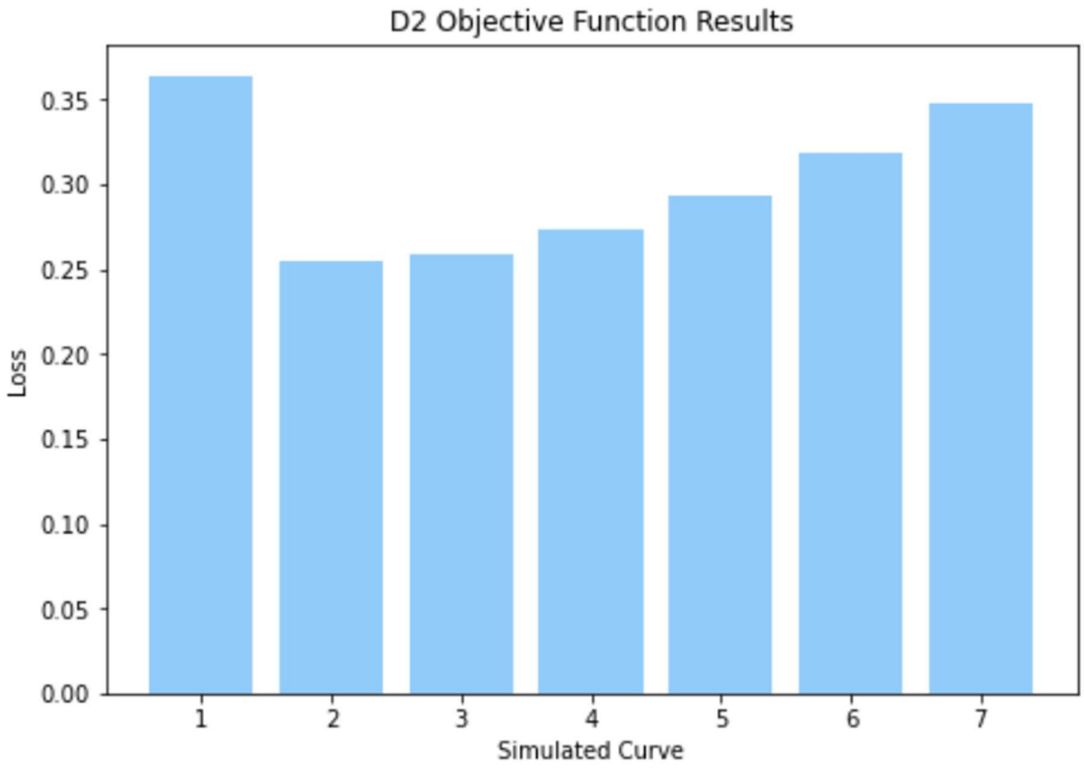



**Figure 9: D2 Objective Function Results**

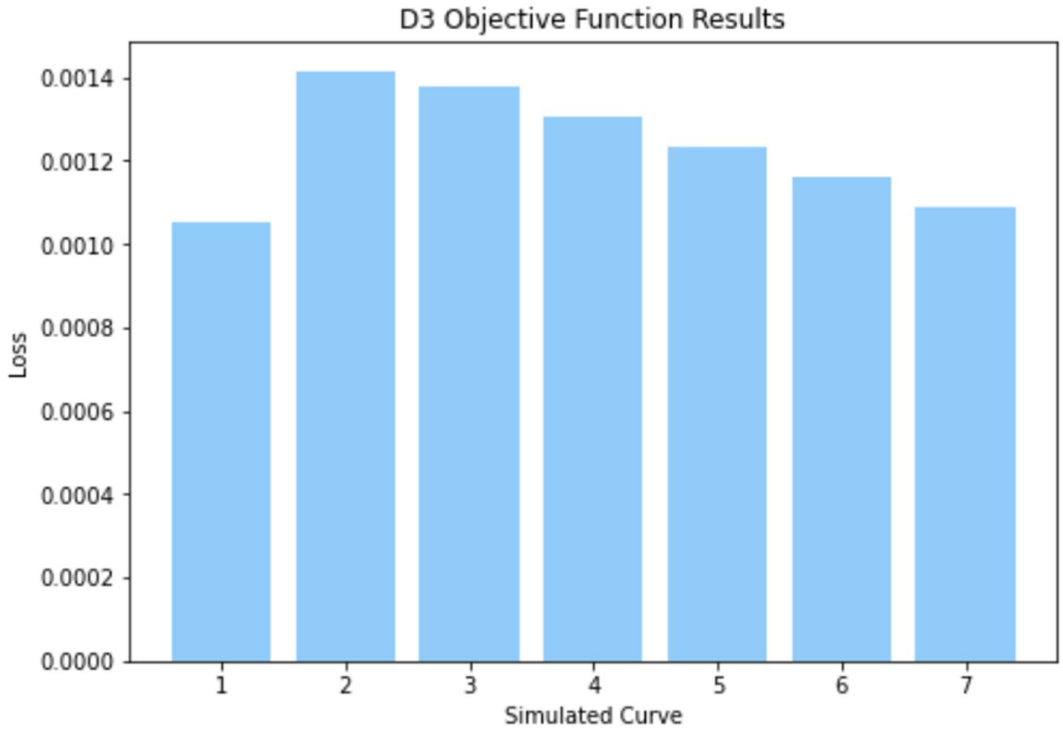



**Figure 10: D3 Objective Function Results**

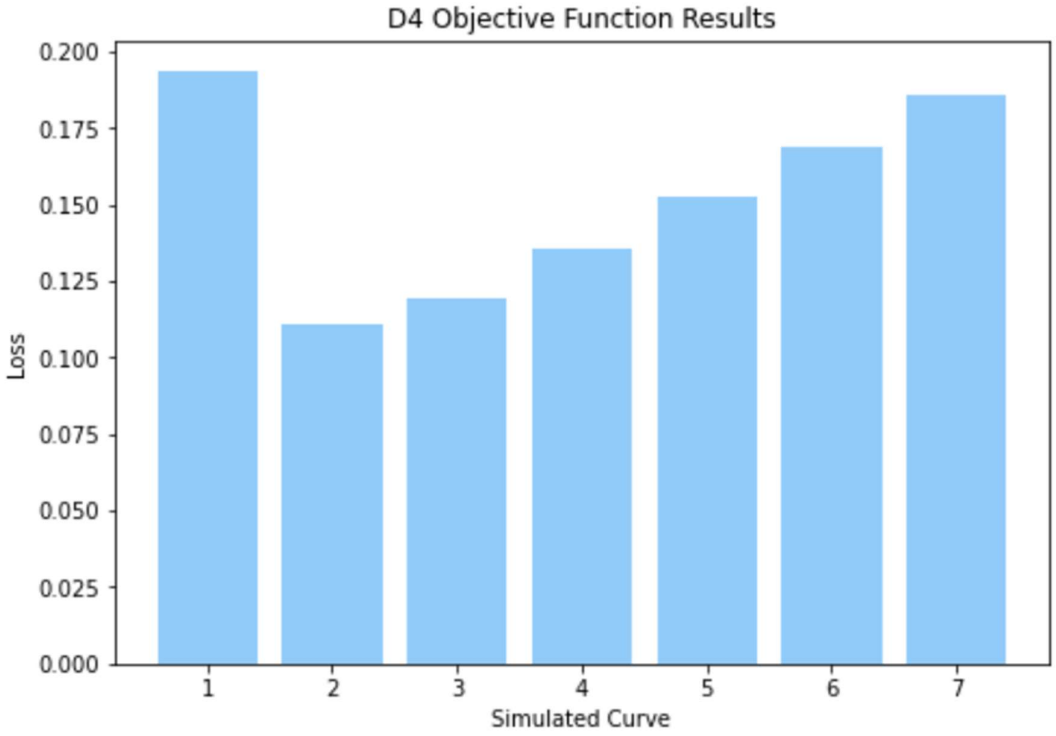



**Figure 11: D4 Objective Function Results**

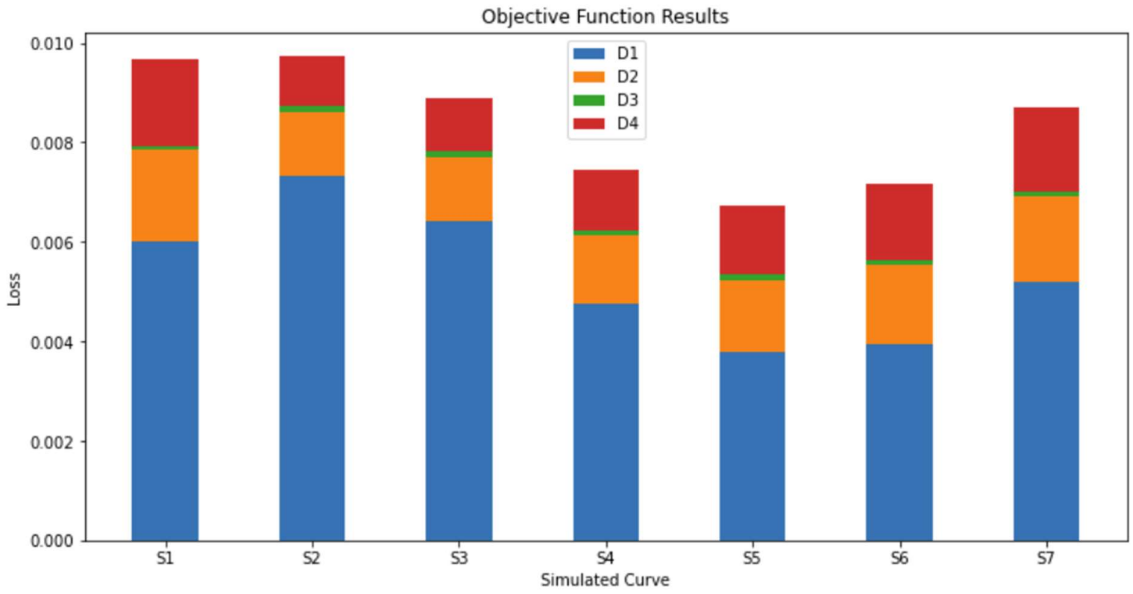



**Figure 12: Fitness Function Results**

For Figure 12, $Fitness = 0.9 * D1 + 0.005 * D2 + 0.086 * D3 + 0.009 * D4$. D2 and D4 resulted in very large loss values, because these objection functions take the difference of the derivates between the experimental curve and the simulated curve. Thus, their designated weights were chosen to be small. It is though that the first objective function D1 is the most significant and thus carries the largest weight. D4 is chosen as the least significant due to it only representing one value (the maximum), the same it thought of D3.

## 4.4 Validation and discussion

In order to validate the evaluation criteria, the objections functions were run on a different set of data, which had the following flow curves in Figure 13.

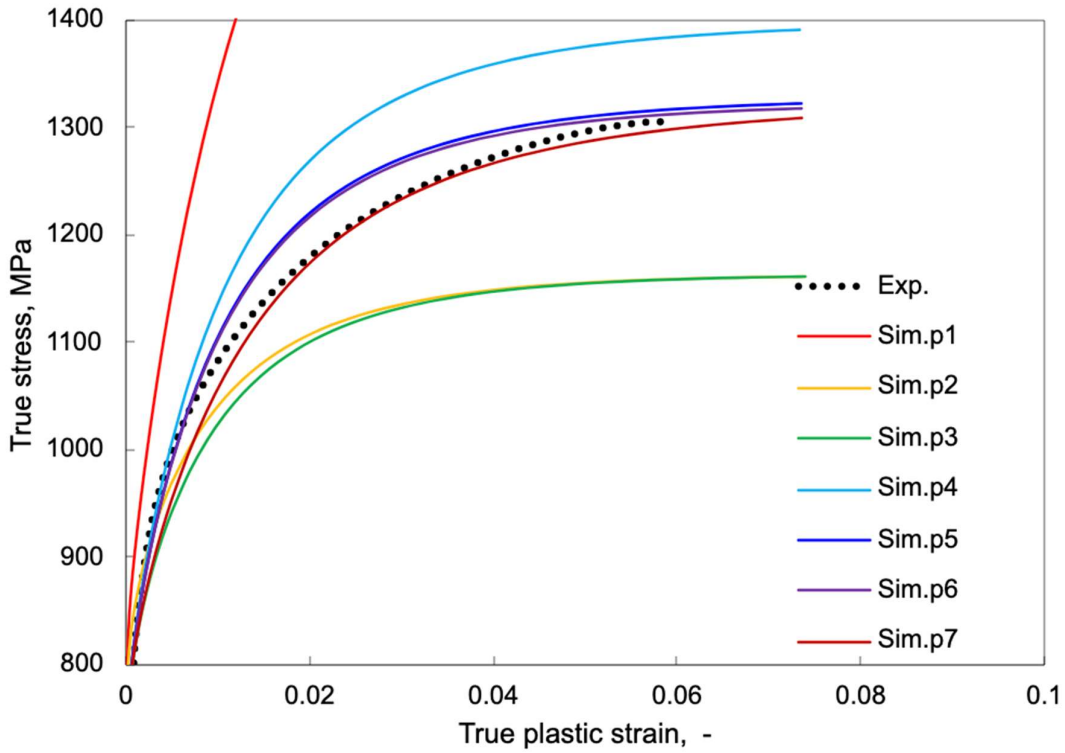



**Figure 13: Flow Curves Used in Validation**

From Figure 13, Sim.p5, Sim.p6 and Sim.p7 should return the lowest values as they appear to best match the experimental data and Sim.p1 should return the highest value. Figures 14-17 are the results for D1, D2, D3 and D4 for the validation dataset.

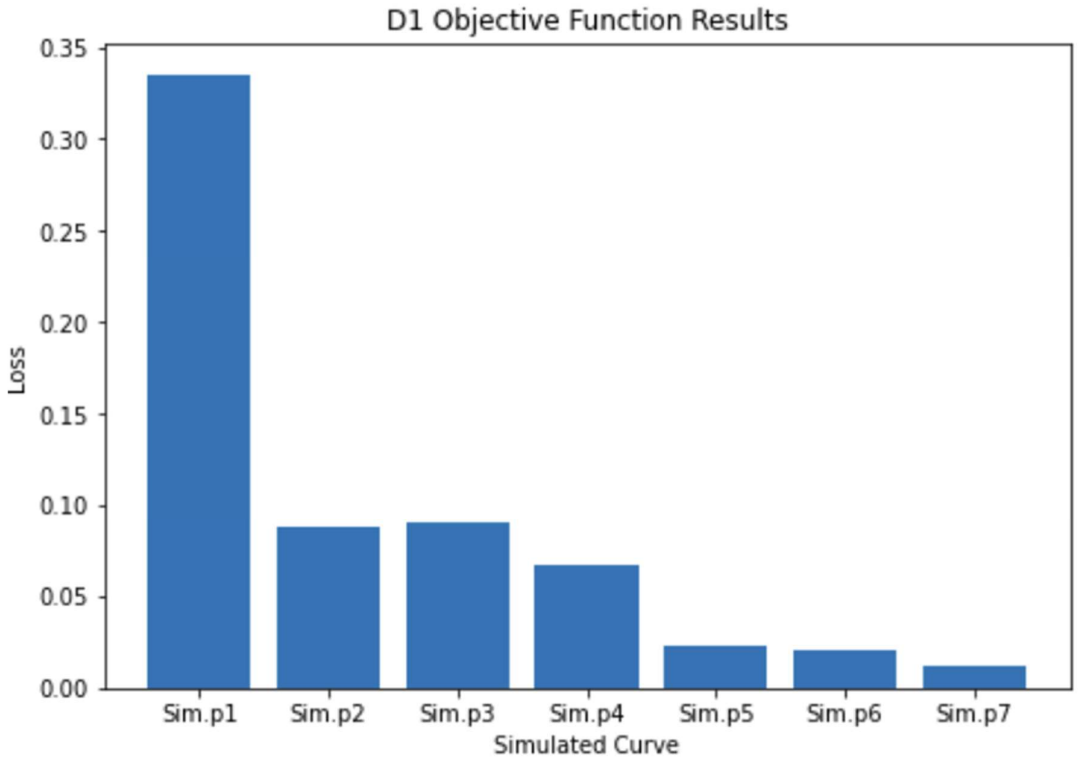



**Figure 14: D1 Validation Objective Function Results**

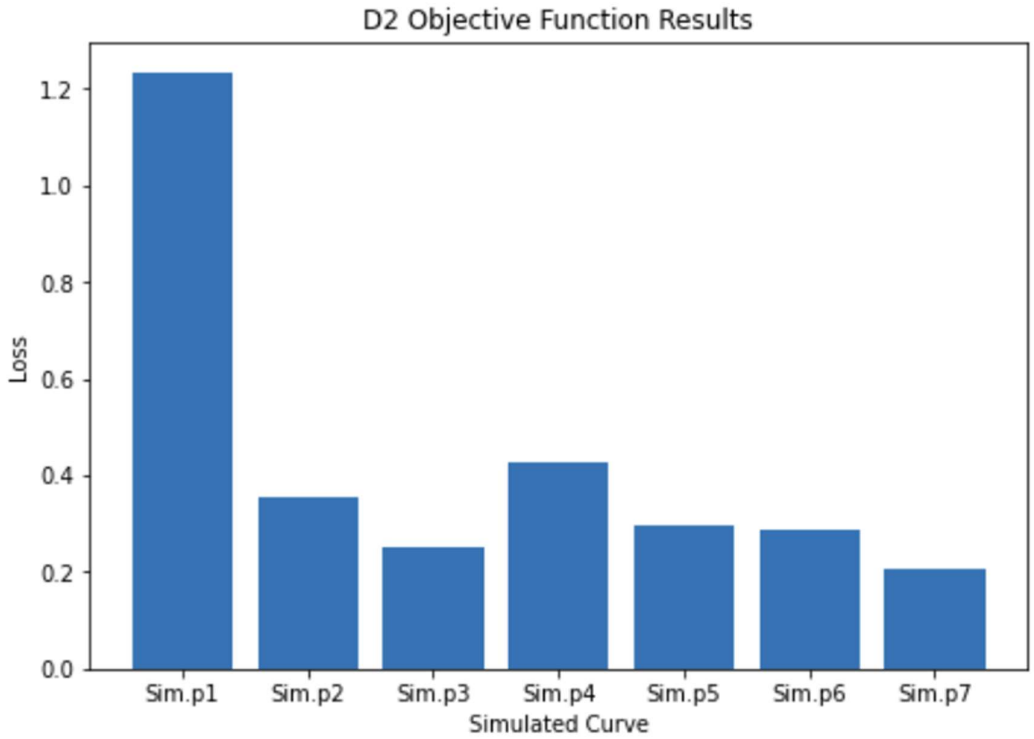



**Figure 15: D2 Validation Objective Function Results**

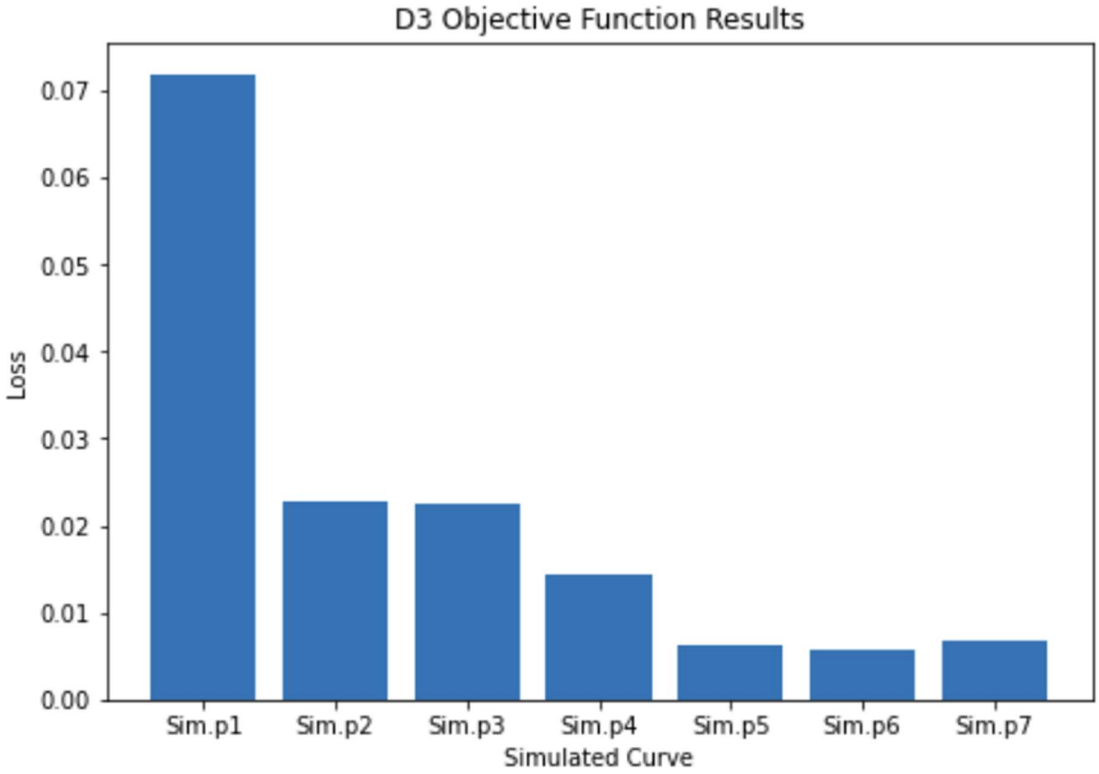



**Figure 16: D3 Validation Objective Function Results**

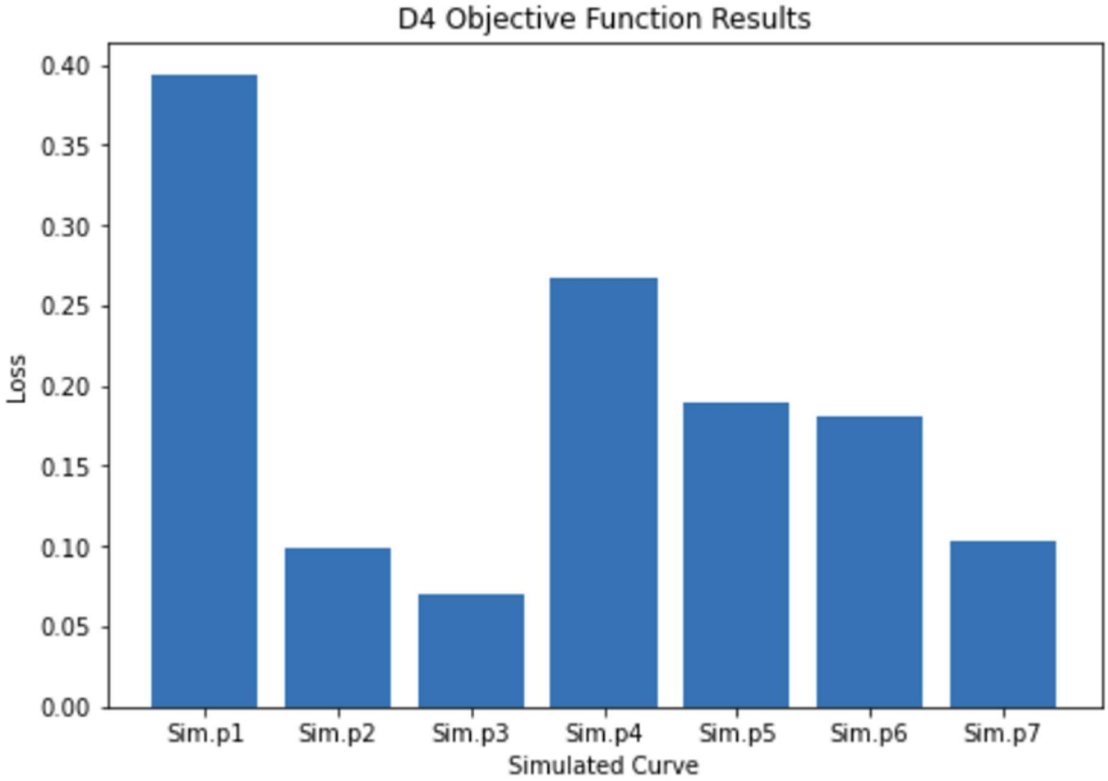
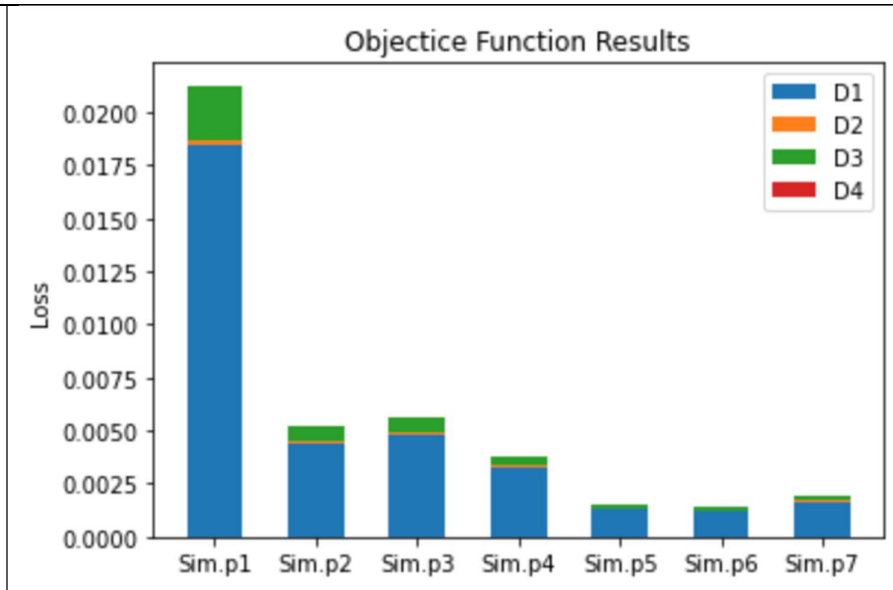



**Figure 17: D4 Validation Objective Function Results**

(a)




(b)

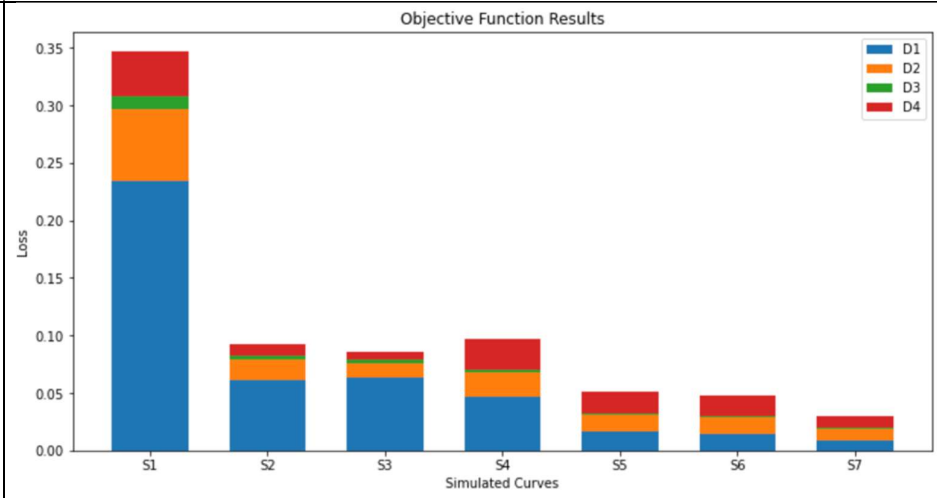



**Figure 18: Validation Fitness Function Results (a) Results from 0.002 (b) Results from 0.005**

Figure 18 (a) showcases the results for the fitness function after being run on data from 0.002, as this is before the yielding point the results where not what was anticipated. Sim.p5 appeared to be the best fit, but this is untrue as can be seen in Figure 13, thus the data was further trimmed to start at 0.005, which is seen in Figure 18 (b).

For Figure 18 (b) the equation used was: $Fitness = 0.7 * D1 + 0.05 * D2 + 0.15 * D3 + 0.1 * D4$. The weights selected were slightly different than prioviously, this was because when the same weights were selected the Objective functions did not fill the bars proportionally, having the different weights preserved these proportions. Overall, Sim.p5, Sim.p6 and Sim.p7 are the best fit data and Sim.p1 is the worst fit. The evaluation criteria passes the validation check.

# 5 Parameter optimization

## 5.1 Response surface method

This section will explain how the response surface works as an approximation to the consititutive model, and how the response surface model is set up and updated in the optimization process.

The response surface is implemented with the Sci-Kit Learn MLPRegressor class. Its structure is set up exactly as is shown in figure 2, and its code implementation is in the optimize.py script in Section 9. The MLPRegressor is initially fit using the 30 initial DAMASK simulations, and learns to approximate the relationship of the parameters to the output stress values during the fitting process. Since the strain values for the simulations and the target curve stress values are all identical, the strain range is omitted from the MLP model as it can be assumed constant.

During each iteration the MLPRegressor is re-fit with the additional simulation values. As the iterative optimization approach runs the MLPRegressor will fit an incrementally better approximation each time.

## 5.2 Genetic algorithm and convergence

This section will explain how the GA is set up in our process, how the GA works alongside the response surface to converge to incrementally better solutions. The code implementation and use in presented in optimize.py in Section 10.

In this section, the optimization process proposed presented in Section 5 is tested using multiple example scenarios. To run, the parameter optimization program requires a target stress–strain curve and bounded constraints for each of the parameters. To test the performance and accuracy of the optimization two different types of target curves are used.

1. Experimentally measured stress–strain curves, for which constitutive parameters are unknown.
2. Simulated stress–strain curves, for which the constitutive parameters are known.

The GA implemented in our process runs for 100 generations with a 25% gene mutation rate.

For evaluation metrics, the fitness function and the individual objective functions proposed in Section 2 will be used to determine a good optimization result, but the GA uses the MSE as a fitness function for simplicity.

## 5.3 Parameter calibration with an automatic optimization approach

### 5.3.1 Setup

This section will cover the setup for the optimization process. Including the setup for the RVE used in our CP simulations, and the setup required to run an optimization program.

The RVE setup we used is a 512 element, ($8^3$) RVE. This small volume was chosen for the faster time needed to run simulations.

In this test our target curve is presented in Figure 19. The unprocessed experimental curve is shown alongside its (automatically) preprocessed version.

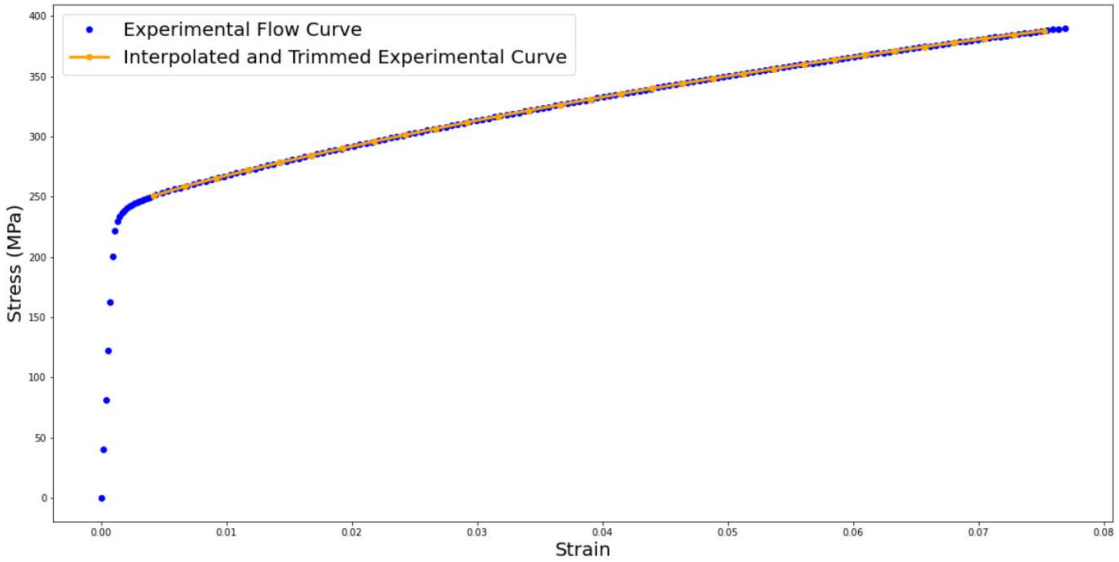



**Figure 19: Experimenally measured target stress–strain curve.**

The preprocessing step for the experimental curve consists of first interpolating it with linear interpolation and then pruning the points that are outside of the strain range in

the DAMASK simulations. This preprocessing step is done automatically in the optimization program.

Next the constraints for the parameters in our optimization process are defined. The constraints tell the GA algorithm what are the feasible lower and upper bounds for each parameter, and also define a step size.

**Table 1 Intial parameter constrains and steps for response surface calculation**

| Parameter | Bounds | Step |
|---|---|---|
| $\tau$, MPa | $[120, 150]$ | 1 |
| $\tau_s$, MPa | $[150, 300]$ | 1 |
| $h_0$, MPa | $[600, 1000]$ | 5 |
| $\alpha$ | $[1, 3]$ | 0.2 |

### 5.3.2 Results

When running the optimization process with these inputs the program converged to a solution after 5 iterations. Including the initial simulations used to initialize the response surface, a total of 35 CP simulations were run.

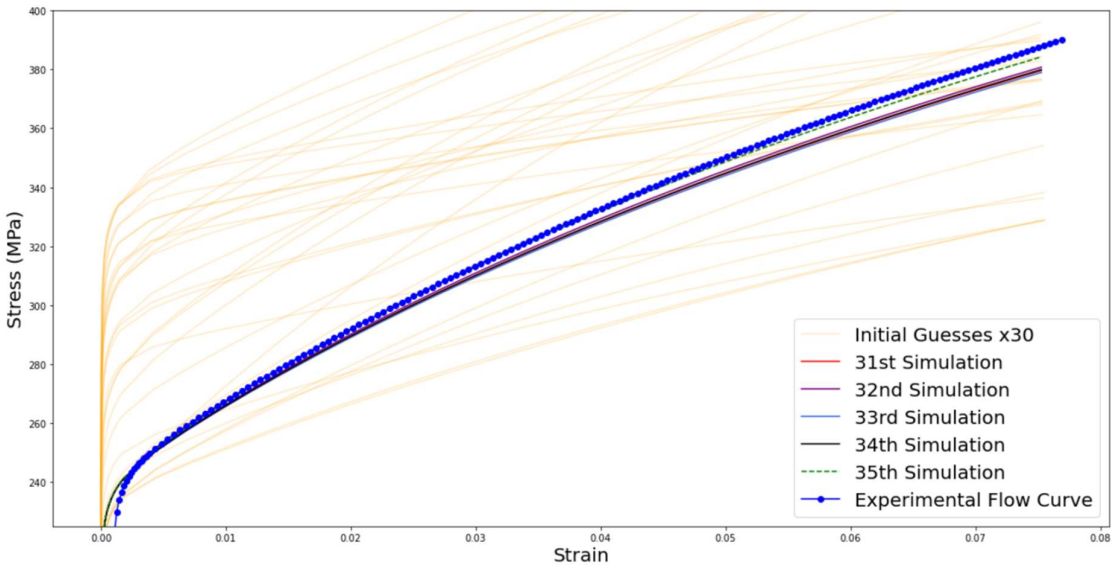



**Figure 20: Optimization results for experimental target curve.**

While it can be hard to visually asses the best stress–strain curve here. From examining the values of fitness function introduced in Section 4, the values for the last

5 simulations it is evident that the 35$^{th}$ simulation has the best fit out of all the curves. This is displayed visually in figure 21.

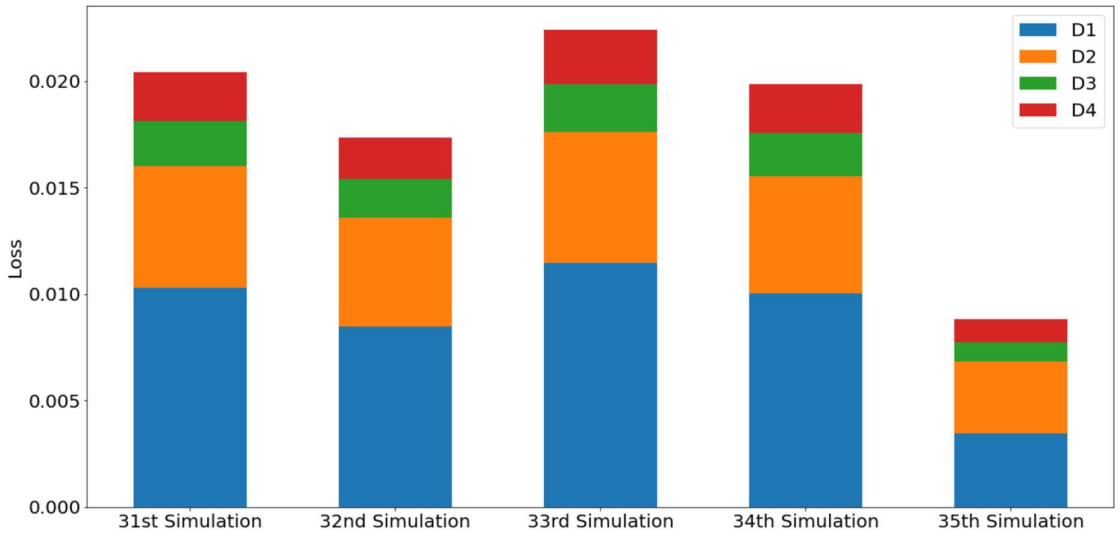



**Figure 21: Fitness values for the experimental target test.**

## 5.4 Validation of parameter optimization approach

For the validation of our parameter optimization process a simulated target curve, for which we know the ground truth parameters for is used. A successful optimization result should be able to converge to similar ground-truth parameters. An evaluation of the variance in the optimized and ground-truth parameters can then be performed to assess the performance of the model.

### 5.4.1 Setup

For the second test the preprocessed (interpolated and trimmed) version of the flow curve is shown in figure n. The interpolated region is the region for which we will optimize for.

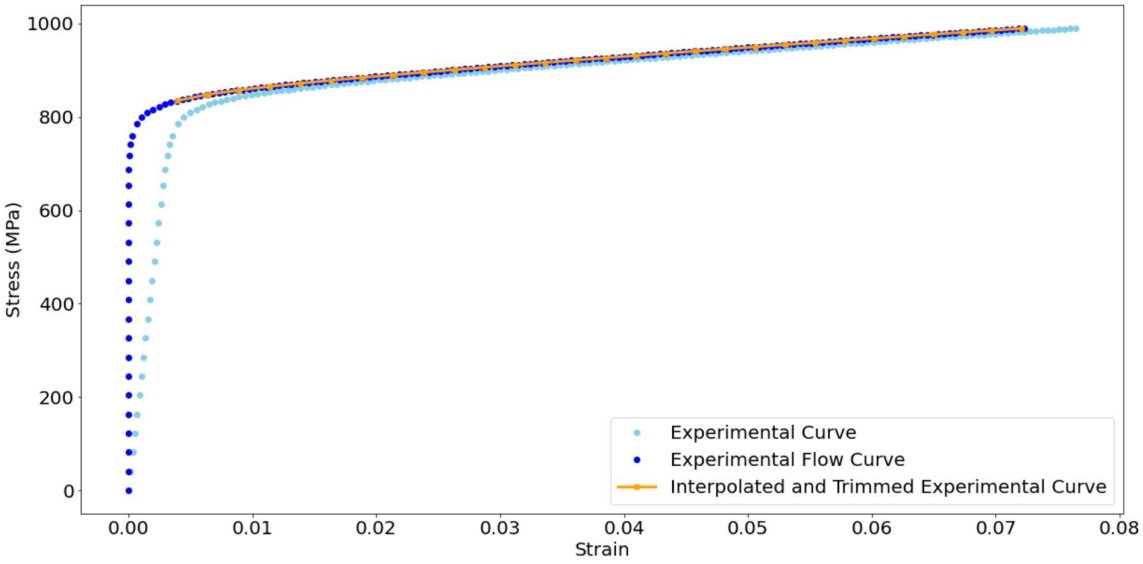



**Figure 22: Initial true stress–true strain curve, true stress–true plastic strain curve, interpolated true stress–true plastic strain cure**

Next the constraints for the parameters in our optimization process are defined. **Table 2** shows the ranges and steps for the simulated target.

**Table 2: CP parameters for the validation data set**

| Parameter | Bounds | Steps | Optimized parameters | Ground-truth parameters | Deviation |
|---|---|---|---|---|---|
| $\tau$, MPa | [350-380] | 1 | 360 | 362 | $-2$ |
| $\tau_s$, MPa | [1150-1250] | 1 | 1215 | 1200 | 15 |
| $h_0$, MPa | [1100-1300] | 5 | 1195 | 1200 | $-5$ |
| aslip | [3-4] | 0.1 | 3.6 | 3.7 | $-0.1$ |

### 5.4.2 Results

In this test the best solution is found on the 35$^{th}$ simulation. In these results optimization process was stopped manually after the loss kept increasing gradually. The calibrated and the targeted parameters are summarized in **Table 2**. From a comparison of the optimized parameters and the ground-truth parameters we can see that the deviations are quite small. The differences are shown in the deviation column, the largest difference is in the value of $\tau_s$, which is 15 MPa more in the optimized value. For the

other parameters, and with respect to the steps, the optimized parameters are very close to the ground truth values indicating that the optimized values are indeed significant.

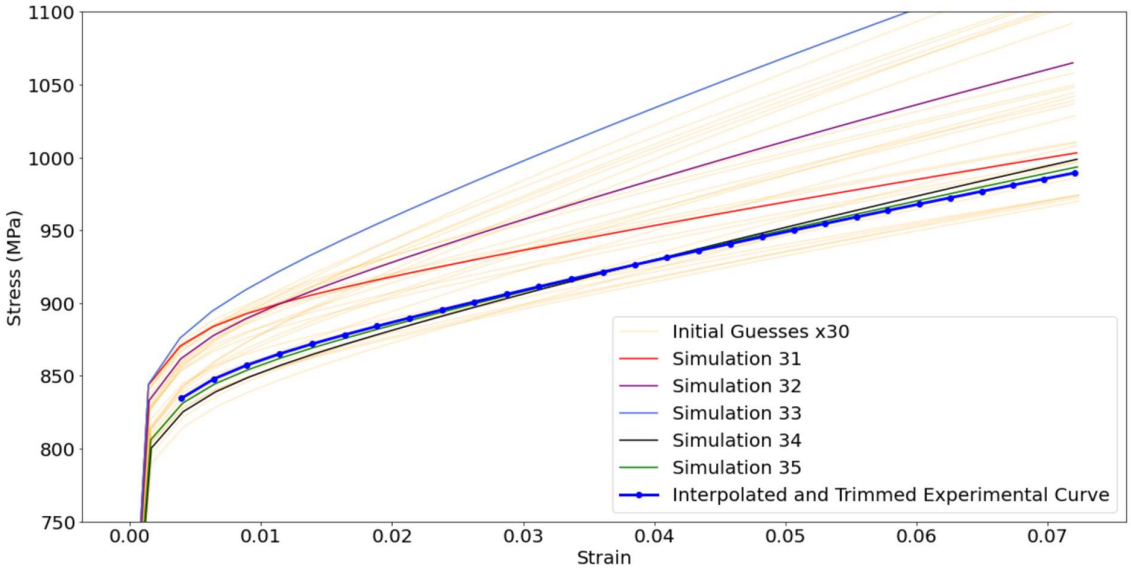



**Figure 23: Flow curves for optimization of the simulated target**

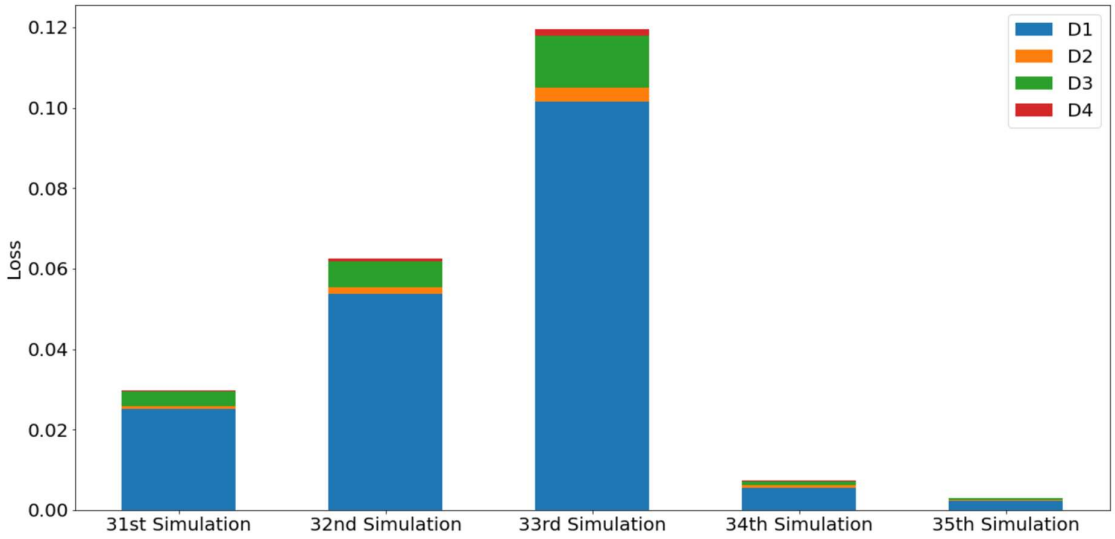



**Figure 24: Fitness values of the best solutions to the simulated target**

## 5.5 Discussion

During any optimization process it is possible that a model fails to converge to a good solution. This is also true for the optimization process presented in this paper. This

subsection will be dedicated to showcasing some of the issues and characteristics of the convergence encountered during test trials.

### 5.5.1 Unsuitable parameter constraints

One issue that could lead to poor solutions is unsuitable parameter constraints. If the constraints for the parameters are too small the optimization process can get stuck at the 'border' of a parameter's range. When this happens the iterative process will repeat simulations at the same parameter values and gets stuck.

Figure n is a visualization of a tight parameter range where the 33rd and 32nd simulations have gotten stuck and are both identical. In this particular case the problem was a poorly defined $\alpha$ range from $[3, 5]$ instead of $[1, 3]$.

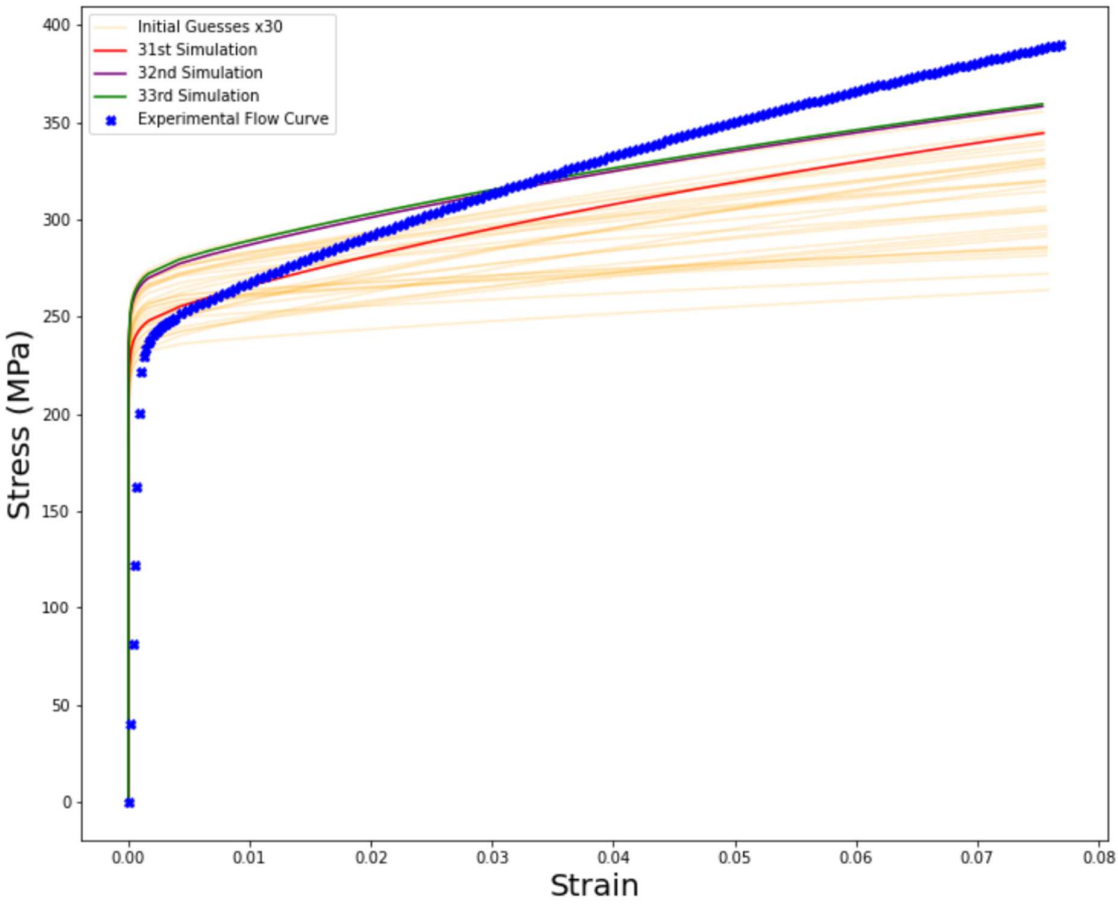



**Figure 25: Tight parameter constraints causing a bad fit.**

There are multiple reasons for how poor convergence could happen in this optimization process. We can hypothesize that a poor convergence should happen because of two main reasons, the response surface and the optimization (MLP and GA).

The first risk for poor convergence the response surface is either underfitting or overfitting, resulting in a poor approximation of the constitutive model. For a reliable fit the response surface needs to be appropriately tuned for the underlying optimization problem through cross-validation methods. Implementing a good cross-validation method is one of the improvement points for this paper.

A bad response surface will consequently result in a poor optimization space for the GA. But another concern is that the GA will also be unable to optimize properly on even an accurate response surface. Similarly to the cross-validation proposed for the response surface, additional testing to validate the performance of the GA is also left as future work.

# 6 Conclusions and outlooks

Overall, the objective of the paper which is to build a program-driven control of crystal plasticity parameter optimization have been accomplished. First of all, the data processing pipeline for obtaining the stress - strain curves and the stress – plastic strain curves from any given amount of .txt files from running DAMASK simulations.
Next, four objective functions were implemented for the evaluation criteria, which all passed the validation check. The results show that the objective function $D1$ has the smallest weighted error loss and therefore is the most significant and carries the largest weight. On the contrary, the objective function $D4$ has the largest weighted loss, thus it is the least significant and carries the smallest weight.

The additional deliverable program-driven control of the CP simulation with bash scripts using Slurm was constructed successfully, with the program submitting multiple CP simulations jobs and returning .txt file for each of them.

The optimization pipeline was implemented using the genetic algorithm with the response surface methodology. Although the initial results were promising, there are a lot of room for performance improvements in terms of computational time and accuracy of the algorithm.

Future work could include hyperparameter tuning of the GA algorithm and RSM, which is finding better ranges for the starting parameters using either manual input or searching algorithms like grid search, random search or Bayesian. Also, tuning the weights for the GA algorithm using said algorithms can also make a big difference in performance. In addition, implementing more optimization or ML algorithms provides more insights as to compare how each different algorithm interacts and performs with CP simulation.

In general, this paper has been successful in implementing and presenting a potentially powerful framework and pipeline for performing CP parameter calibration. The prospects of applying this framework for future applications in not only the materials science field, but also other engineering fields as well are extremely bright. This paper proves to be an extremely strong bridge between computer science and engineering,

and hopefully paves the way for more upcoming similar interactions between these two cornerstones of modern science and society.

There were quite many issues that were encountered during the making of this paper. First of all, knowledge gaps were substantial throughout the majority of the project, as not many members were well-versed with the topic of optimization, programming with Python, or implementing ML algorithms with Python. Also, writing bash scripts was picked up on the project to implement the program-driven control of CP. As such, literature review were consistently happening during the course, though it hindered the time that was available to complete the tasks of the project, the learning process was precious to the members of this paper.

# 7 Personal evaluation

Overall this project was a great interdisciplinary project that showed how machine learning methods and materials science can be used together. The key takeaways were the principles and techniques for an effective optimization experiment.

As this paper has outlined, there were quiete a few knowledge gaps in both materials sicnece and optimization. However, the knowledge that was gained will undoubtedly be helpful in the long run.

This project also required a substantial amount of programming experience to build many parts of the project. Some members did not have the programming background required to build parts of the project, thus they spent a lot of time learning this required background. Nonetheless, the experience was greatly treasured among us. In addition, effective program design and efficient execution were among the key takeaways from this project. Also, certain health issues were remarked for a member of this project, which greatly limits the work output for that member during the second-half of the project.

The members of this project would like to give special thanks to our advisor Wenqi Liu for all the support she has given. Which included reviewing the weekly progress and answering any questions and doubts that arised. As the members of this project were not well-acquanted with this subject, the detailed and extensive feedback given by Wenqi guided the team through the project.

# 8 References

[1] W. Liu, J. Lian, N. Aravas, S. Münstermann, A strategy for synthetic microstructure generation and crystal plasticity parameter calibration of fine-grain-structured dual-phase steel,
International Journal of Plasticity, 2020, Pages 12-13 126: 102614.
https://doi.org/10.1016/j.ijplas.2019.10.002

[2] K. Sedighiani, M. Diehl, K. Traka, F. Roters, J. Sietsma, D. Raabe, An efficient and robust approach to determine material parameters of crystal plasticity constitutive laws from macro-scale stress–strain curves,
International Journal of Plasticity, Volume 134, 2020, 102779, ISSN 0749-6419,
https://doi.org/10.1016/j.ijplas.2020.102779.
(https://www.sciencedirect.com/science/article/pii/S0749641919308769)

[3] D. Rodney, L. Ventelon, E. Clouet, L. Pizzagalli, F. Willaime, Ab initio modeling of dislocation core properties in metals and semiconductors, Acta Materialia, Volume 124, 2017,Pages 633-659, ISSN 1359-6454,
(https://www.sciencedirect.com/science/article/pii/S1359645416307492)

[4] F. Roters, M. Diehl, P. Shanthraj, P. Eisenlohr, C. Reuber, S.L. Wong, T. Maiti, A. Ebrahimi, T. Hochrainer, H.-O. Fabritius, S. Nikolov, M. Friák, N. Fujita, N. Grilli, K.G.F. Janssens, N. Jia, P.J.J. Kok, D. Ma, F. Meier, E. Werner, M. Stricker, D. Weygand, D. Raabe, DAMASK – The Düsseldorf Advanced Material Simulation Kit for modeling multi-physics crystal plasticity, thermal, and damage phenomena from the single crystal up to the component scale, Computational Materials Science, Volume 158, 2019, Pages 420-478, ISSN 0927-0256,
(https://www.sciencedirect.com/science/article/pii/S0927025618302714)

[5] D. Hermawanto, Indonesian Institute of Sciences (LIPI), Genetic Algorithm for Solving Simple Mathematical Equality Problem

[6] A. Colin Cameron, Frank A.G. Windmeijer, An R-squared measure of goodness of fit for some common nonlinear regression models, Journal of Econometrics, Volume

77, Issue 2, 1997, Pages 329-342, ISSN 0304-4076, https://doi.org/10.1016/S0304-4076(96)01818-0.
(https://www.sciencedirect.com/science/article/pii/S0304407696018180)

[7] M. Fernández-Delgado, M.S. Sirsat, E. Cernadas, S. Alawadi, S. Barro, M. Febrero-Bande, An extensive experimental survey of regression methods,Neural Networks, Volume 111, 2019, Pages 11-34,
(https://www.sciencedirect.com/science/article/pii/S0893608018303411)

[8] Robin M. Schmidt, Frank Schneider, Philipp Hennig, Descending through a Crowded Valley - Benchmarking Deep Learning Optimizers., 2021, https://arxiv.org/abs/2007.01547

[9] Jiaxuan Wang, Jenna Wiens, [2006.16541] AdaSGD: Bridging the gap between SGD and Adam (arxiv.org), 2020, https://arxiv.org/abs/2007.01547

[10] Diederik P. Kingma, Jimmy Ba, Adam: A Method for Stochastic Optimization (arxiv.org), 2014, https://arxiv.org/abs/2007.01547

# 9 Appendix

## 9.1 Optimization script (optimize.py)

```
# External libraries
import pandas as pd
import numpy as np
import pygad
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from scipy.interpolate import interp1d
# Our classes
from classes.SIM import *
from classes.preprocessing import *

# ------------------------------------------------------------------
#   Run initial simulations (used to initialize optimization process)
# ------------------------------------------------------------------

# Initialize SIM object.
sim = SIM()
print("Running Initial Simulations...")
sim.run_initial_simulations()
print(f"Done. {len(sim.simulations)} simulations completed.")
np.save('initial_simulations.npy', sim.simulations)

# -------------------------------------
#   Set up experimental curve (target)
# -------------------------------------

exp = pd.read_csv('sim_curve.csv')
exp_stress = exp["True stress, MPa"]
exp_strain = exp["True strain, -"]
#sim_strain = exp['Simulated strain'].dropna()
f = interp1d(exp_strain, exp_stress)
x_min, x_max = 0.002, exp_strain.max()
```

```
prune = np.logical_and(sim.strain > x_min, sim.strain < x_max)
sim.strain = sim.strain[prune]
exp_target = f(sim.strain).reshape(1,len(sim.strain))

# ----------------------------------------
#  Initialize Response Surface Module (MLP)
# ----------------------------------------

mlp = MLPRegressor(hidden_layer_sizes=[15], solver='adam', max_iter=100000,
shuffle=True)
print("Fitting response surface...")
y = np.array([stress[prune] for (_, stress) in sim.simulations.values()])
X = np.array(list(sim.simulations.keys()))
mlp.fit(X,y)

# ------------------------------
#      Initialize GA
# ------------------------------

# Initialize fitness function
def fitness(solution, solution_idx):
    sol = solution.reshape((1,4))
    y_pred = mlp.predict(sol)
    fitness = 1 / mean_squared_error(y_pred, exp_target)
    return fitness

# Initialize GA Optimizer
num_generations = 100 # Number of generations.
num_parents_mating = 500 # Number of solutions to be selected as parents in
the mating pool.
sol_per_pop = 1000 # Number of solutions in the population.
num_genes = 4
gene_space = [
    {'low': 350, 'high': 380, 'step': 1},  # tau
    {'low': 1150, 'high': 1250, 'step': 1},  # taucs
    {'low': 1100, 'high': 1300, 'step': 5},  # h0
    {'low': 3, 'high': 4, 'step': 0.1}]      # alpha
```

```
last_fitness = 0
keep_parents = 1
crossover_type = "single_point"
mutation_type = "random"
mutation_percent_genes = 25

def on_generation(ga_instance):
    global last_fitness
    generation = ga_instance.generations_completed
    fitness                                                                 =
ga_instance.best_solution(pop_fitness=ga_instance.last_generation_fitness)[
1]
    change                                                                  =
ga_instance.best_solution(pop_fitness=ga_instance.last_generation_fitness)[
1] - last_fitness
    last_fitness                                                            =
ga_instance.best_solution(pop_fitness=ga_instance.last_generation_fitness)[
1]

ga_instance = pygad.GA(num_generations=num_generations,
                       num_parents_mating=num_parents_mating,
                       sol_per_pop=sol_per_pop,
                       num_genes=num_genes,
                       fitness_func=fitness,
                       on_generation=on_generation,
                       gene_space=gene_space,
                       crossover_type=crossover_type,
                       mutation_type=mutation_type,
                       mutation_percent_genes=mutation_percent_genes)


# ------------------------------
#      Helper Functions
# ------------------------------

def output_results(ga_instance):
    # Returning the details of the best solution in a dictionary.
```

```
    solution,           solution_fitness,           solution_idx           =
ga_instance.best_solution(ga_instance.last_generation_fitness)
    best_solution_generation = ga_instance.best_solution_generation
    loss = 1/solution_fitness
    values      =      (solution,      solution_fitness,      solution_idx,
best_solution_generation, loss)
    keys     =     ("solution",     "solution_fitness",     "solution_idx",
"best_solution_generation", "loss")
    output = dict(zip(keys, values))
    return output

def print_results(results):
    print(f"Parameters of the best solution : {results['solution']}")
    print(f"Fitness     value     of     the     best     solution     =
{results['solution_fitness']}")
    print(f"Index of the best solution : {results['solution_idx']}")
    print(f"MSE given by the MLP estimate: {results['loss']}")
    print(f"Best         fitness         value         reached         after
{results['best_solution_generation']} generations.")

# -------------------------------
#      Optimization Loop
# -------------------------------
epsilon = 10.0 # Target MSE value.

# Optimization on initial simulations.
print("Optimizing using GA...")
ga_instance.run()
results = output_results(ga_instance)
print_results(results)
loss = 1e6
# Iterative optimization.
print("--------------------------------")
print("Starting iterative optimization:")
while loss > epsilon:
    sim.run_single_test(tuple(results['solution']))
    np.save('simulations.npy', sim.simulations)
```

```
    y = np.array([stress[prune] for (_, stress) in
sim.simulations.values()])
    X = np.array(list(sim.simulations.keys()))
    mlp.fit(X,y)
    ga_instance.run()
    results = output_results(ga_instance)
    print_results(results)
    loss = mean_squared_error(y[-1].reshape((1,-1)), exp_target)
    print(f"LOSS = {loss}")

print("Optimization Complete")
print("--------------------------------")
print("Final Parameters: ", results['solution'])
print(f"MSE of the final solution : {loss}")
```

## 9.2 Simulator Class (SIM.py)

```
import os
import numpy as np
import random
import shutil
from .preprocessing import *


class SIM:
    def __init__(
        self,
        tau_range = (100, 120, 1),
        taucs_range = (230, 250, 1),
        h0_range = (600, 800, 50),
        alpha_range = (3, 5, 1)
    ):
        # Suggested Parameter ranges.
        random.seed(16)
        self.tau_range = tau_range
        self.taucs_range = taucs_range
        self.h0_range = h0_range
```

```
        self.alpha_range = alpha_range
        self.filename = 0
        self.filename2params = {}
        self.simulations = {}
        self.strain = None

    def submit_job(self):
        """
        Runs the simulation and postprocessing with a shell script.
        code = 0 if success
        code = 1 if errors occurred
        """
        code = os.system(f'sh runsim.sh {self.filename}')
        return code

    def submit_array_jobs(self, start=None):
        """
        Run the simulation and postprocessing.
        Array jobs will submit multiple simulations up until filename:int.
        code = 0 if success
        code = 1 if errors occurred
        """
        if start:
            code = os.system(f'sh array_runsim.sh {self.filename} {start}')
        else:
            code = os.system(f'sh array_runsim.sh {self.filename} {1}')
        return code



    def make_new_job(self, params, path):
        shutil.copytree('./TEMPLATE/', path)
        self.filename2params[path] = params
        self.edit_material_parameters(params, path)

    def edit_material_parameters(self, params, job_path):
        # Edit the material.config file.
        def tau0_edit(num):
```

```
            return f'tau0_slip                    {num} {num}          # per
family\n'

        def tausat_edit(num):
            return f'tausat_slip            {num} {num}      # per family\n'

        def h0_edit(num):
            return f'h0_slipslip              {num}\n'

        def a_edit(num):
            return f'a_slip                   {num}\n'

        path = f'{job_path}/material.config'
        with open(path) as f:
            lines = f.readlines()

        lines[36] = tau0_edit(params[0])
        lines[37] = tausat_edit(params[1])
        lines[46] = h0_edit(params[2])
        lines[54] = a_edit(params[3])

        with open(f'{job_path}/material.config', 'w') as f:
            f.writelines(lines)

    def run_initial_simulations(self):
        """
        Runs N simulations according to get_grid().
        Used when initializing a response surface.
        """
        n_params = self.get_grid()
        for params in n_params:
            self.filename += 1
            path = f'./simulations/{str(self.filename)}'
            self.make_new_job(params, path)
        self.submit_array_jobs()
        self.strain = self.save_outputs()
```

```
    def run_single_test(self, params):
        """
        Runs a single simulation with 'params'.
        Used during optimization process.
        """
        self.filename += 1
        path = f'./simulations/{str(self.filename)}'
        self.make_new_job(params, path)
        self.submit_array_jobs(start=self.filename)
        self.save_single_output(path, params)

    def get_grid(self):
        points = []
        np.random.seed(16)
        for _ in range(30):
            tau = np.random.randint(low=350, high=380)
            taucs = np.random.randint(low=1150, high=1250)
            h0 = np.random.randint(low=1100, high=1300)
            #alpha = np.random.randint(low=1, high=3)
            alpha = np.round(np.random.uniform(low=3, high=4),1)
            points.append((tau, taucs, h0,alpha))
        return points

    def save_outputs(self):
        true_strains = []
        for (path, params) in self.filename2params.items():
            path2txt = f'{path}/postProc/'
            files    =    [f    for    f    in    os.listdir(path2txt)    if
os.path.isfile(os.path.join(path2txt, f))]
            processed = preprocess(f'{path2txt}/{files[0]}')
            true_strains.append(processed[0])
            self.simulations[params] = processed
        return    np.array(true_strains).mean(axis=0).round(decimals=3)    #
Outputs the strain values for the simulations

    def save_single_output(self, path, params):
        path2txt = f'{path}/postProc/'
```

```
        files = os.listdir(path2txt)
        processed = preprocess(f'{path2txt}/{files[0]}')
        self.simulations[params] = processed
```

## 9.3 Preprocessing Functions (preprocessing.py)

```
import pandas as pd
import numpy as np
import os
import glob
import csv
from sklearn.metrics import r2_score
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
from scipy import stats

## Data preprocessing
def preprocess(path, to_csv=False):
    df = pd.read_csv(path, skiprows = 6, delimiter = "\t")
    #Main function that calls all other functions.
    strain = df["Mises(ln(V))"]
    stress = df["Mises(Cauchy)"]
    true_plastic_strain = getTruePlasticStrain(strain,stress)

    if to_csv:
        data = {
           "Trimmed Stress" : trimmedStress,
           "Trimmed Strain" : trimmedTrueStrain
        }
        newdf = pd.DataFrame(data)
        newdf.to_csv(f"{path}/flowcurve.csv", index=False)

    return (true_plastic_strain.to_numpy(), stress.to_numpy())

def getTruePlasticStrain(strain, stress):
    # Getting the slope
    slope = (stress[1] - stress[0]) / (strain[1] - strain[0])
```

```
    true_plastic_strain = strain - stress / slope
    return true_plastic_strain

def elastic(stress, strain, trueStrain):
    # Obtain the elastic stress and strain based on r-squared
    trimValue = np.arange(0.0005, 0.02, 0.00001)
    trimValue = np.flip(trimValue)
    r2 = 0.00
    elasticTrueStrain = trueStrain
    elasticStrain = strain
    elasticStress = stress
    while(r2 <= 0.985):
        for x in trimValue:
            val = np.argmax(strain >= x)
            elasticTrueStrain = trueStrain[1:val]
            elasticStrain = strain[1:val]
            elasticStress = stress[1:val]
            r2 = adjR(elasticStrain, elasticStress, 1)
    trimmedTrueStrain = trueStrain.drop(range(1, val))
    trimmedStrain = strain.drop(range(1, val))
    trimmedStress = stress.drop(range(1, val))
    return elasticStress, elasticStrain, elasticTrueStrain, trimmedStress,
trimmedStrain, trimmedTrueStrain

def   plot(elasticStress,   elasticStrain,   trimmedStress,   trimmedStrain,
trimmedTrueStrain):
    plt.plot(trimmedStrain, trimmedStress, 'g', label = "SingleCrystalTest")
    plt.plot(trimmedTrueStrain, trimmedStress, 'b', label = "FlowCurve")
    plt.plot(elasticStrain, elasticStress, 'r', label = "Elastic")
    leg = plt.legend(loc = "upper right")
    plt.xlabel(xlabel = "Strain (mm)")
    plt.ylabel(ylabel = "Stress (MPa)")
    plt.xlim([0, 0.005])
    plt.figure(figsize = (6,6))
    plt.show()

def adjR(x, y, degree):
```

```
    results = []
    coeffs = np.polyfit(x, y, degree)
    p = np.poly1d(coeffs)
    yhat = p(x)
    ybar = np.sum(y)/len(y)
    ssreg = np.sum((yhat-ybar)**2)
    sstot = np.sum((y - ybar)**2)
    results = 1- (((1-(ssreg/sstot))*(len(y)-1))/(len(y)-degree-1))
    return results
```

## 9.4 Implementation of objective functions in Python:

```
# D1 loss function: L2 loss
def D1(exp_stress, sim_stress):
    return np.sqrt(np.sum(np.square(exp_stress - sim_stress))/np.sum(np.square(exp_stress)))

# D2 function, the difference of the average slope between the response and experimental max
def D2(exp_stress, sim_stress):
    exp_stress_d1 = np.diff(exp_stress)/np.diff(strainobj)
    sim_stress_d1 = np.diff(sim_stress)/np.diff(strainobj)
    return np.sqrt(np.sum(np.square(sim_stress_d1 - exp_stress_d1))/np.sum(np.square(exp_stress_d1)))

# D3 Function, max(Derivation of stress at each strain between exp. and sim. )
def D3(exp_stress, sim_stress):
    return np.max(np.sqrt(np.square(exp_stress - sim_stress)/sum(np.square(exp_stress))))

# D4 function, max(Derivation of slope at each strain between exp. and sim. )
def D4(exp_stress, sim_stress):
    exp_stress_d1 = np.diff(exp_stress)/np.diff(strainobj)
    sim_stress_d1 = np.diff(sim_stress)/np.diff(strainobj)

    return max(np.sqrt(np.square(sim_stress_d1 - exp_stress_d1))/np.sum(np.square(exp_stress_d1)))

#Fitness, weighted sum of loss functions
def fitness(exp_stress, sim_stress, w1, w2, w3, w4):

    return ( w1*D1(exp_stress, sim_stress) + w2*D2(exp_stress, sim_stress)
           + w3*D3(exp_stress, sim_stress) + w4*D4(exp_stress, sim_stress))
```