Homework 2

BMEGEMMNWCM, Continuum Mechanics

We use the 2nd-order incompressible Ogden's isotropic hyperelastic constitutive model to represent the mechanical behavior of an incompressible rubber-like material. The strain energy potential of the model is defined as

$$W = \sum_{k=1}^{2} \frac{2\mu_k}{\alpha_k^2} \left(\lambda_1^{\alpha_k} + \lambda_2^{\alpha_k} + \lambda_3^{\alpha_k} - 3 \right).$$

The model contains four independent material parameters $(\mu_1, \mu_2, \alpha_1, \alpha_2)$. We have performed the following experimental tests on the material: Uniaxial extension, Equibiaxial extension, Planar Extension. The measured data are given in tabular form.

TASKS:

General: Determine the material parameters of the 2nd-order Ogden's model by performing a parameter-fitting task. Use the "*Root Mean Squared Relative Error*" with the engineering stress to define the quality function for all experiments. Use *constrained minimization* with constraint $\mu_1 + \mu_2 > 0$ in order to ensure positive ground-state Young's modulus.

• **Task 1.** Plot the experimental data points for each test in the same coordinate system. Use stretch for the horizontal axis and the nominal stress for the vertical axis. The plot range for stretch must be $\lambda = 1 \dots 4$. The plot range for the nominal stress must be $P = 0 \dots P_{\max}$, where $P_{\max}$ is the maximum measured nominal stress in the equibiaxial test. You can connect the data point with lines, but use dots to indicate the particular data points.

• **Task 2.** Obtain the material parameters by fitting the model only to the uniaxial data. Consequently, in the calculation of the quality function use only the uniaxial data. The parameter-fitting is acceptable if $Q_U < 5$ %. Report the values of the fitted material parameters and Q_U.

• **Task 3.** Compute the quality functions (numerical values) of the fitted model (in Task 2) for the equibiaxial and the planar extensions. Report the values!

• **Task 4.** Visualize the model accuracy by plotting the model solutions for the stresses in each test using the fitted material parameters. Use separate figures for the tests. Use the plot range defined in Task 1. Write a few sentences (your personal conclusions and observation) about the accuracy of the fitted model.

Task 5. Obtain the material parameters by fitting the model simultaneously to all measurements. Consequently, the quality function is the mean of the individual quality values: $Q = (Q_U + Q_B + Q_P)/3$. The parameter-fitting is acceptable if $Q < 5$ %. Report the values of the fitted parameters and Q, Q_U, Q_B, Q_P.

• **Task 6.** Visualize the model accuracy by plotting the model solutions for the stresses in each test using the fitted material parameters in Task 5. Use separate figures for the tests. Use the plot range defined in Task 1. Write a few sentences (your personal conclusions and observation) about the accuracy of the fitted model.

• **Task 7.** The homogeneous displacement field in the material is given with the components of the material displacement field. The displacement components are given in [mm] if the coordinates are substituted in [mm]. The material is in a plane stress state, i.e. $\sigma_z = \tau_{xz} = \tau_{yz} = 0$. Determine the unknown scalar k included in the displacement field using the incompressibility constraint!

• **Task 8.** Determine the principal stretches and the normalized principal Eulerian directions.

• **Task 9.** Compute the components of the matrix of the Cauchy stress tensor using the fitted material parameters in Task 5. Determine the Mises equivalent stress.

Your data can be downloaded from the following link:

https://www.mm.bme.hu/~kossa/NEPTUN-HW2.pdf

The string **NEPTUN** has to be replaced with the Student's NEPTUN code. The link above is case-sensitive! Use uppercase letters for the file-name! Students must solve the problem corresponding to their NEPTUN's code!

Budapest University of Technology and Economics
Faculty of Mechanical Engineering

Continuum Mechanics
II. Homework

Gergő Kelle

November 29, 2023

Contents

1 Task - Experimental data

In this task I've created a visual representation of the given dataset of Table 1.

Table 1: Mechanical Test Results

	Uniaxial Test	**Equibiaxial Test**	**Planar Test**
Eng. Strain [%]	Eng. Stress [MPa]	Eng. Stress [MPa]	Eng. Stress [MPa]
0	0	0	0
30	8	18	12
60	15	30	18
90	20	47	26
120	28	59	35
150	43	85	50
180	65	105	69
210	92	140	95
240	116	176	132
270	171	231	169
300	220	307	227

Before the visualization I've calculated stretch from engineering strain with the following equation

$$\lambda = \frac{\varepsilon_{eng}}{100} + 1 \ . \tag{1}$$

The resulting solution is plotted at Figure 1.

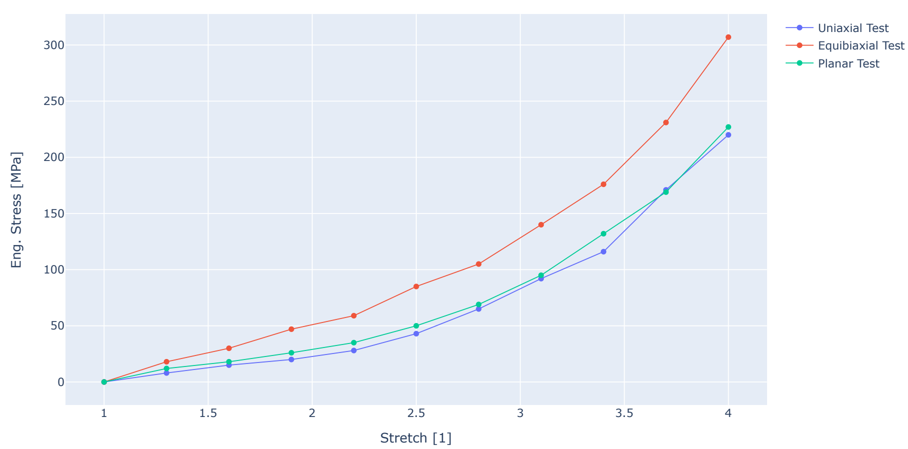

Figure 1: Stress-Stretch Curves for Different Tests

2 Task - Model fitting to the uniaxial data

As the Homework determined the quality function that is used to describe the parameter fitting correctness is the root mean squared relative error which should be implemented. In general it has the following form:

$$Q = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{P_i^{exp} - P_i^{sim}}{P_i^{exp}}\right)^2} \tag{2}$$

Where P^{sim} simulated nominal stress is determined based on the type of the experimental test. Therefore the different nominal stresses are calculated the following ways:

$$P_U = \sum_{i=1}^{2} \frac{2\mu_i}{\alpha_i}\left(\lambda^{\alpha_i - 1} - \lambda^{-0.5\alpha_i - 1}\right) \tag{3}$$

$$P_B = \sum_{i=1}^{2} \frac{2\mu_i}{\alpha_i}\left(\lambda^{\alpha_i - 1} - \lambda^{-2\alpha_i - 1}\right) \tag{4}$$

$$P_P = \sum_{i=1}^{2} \frac{2\mu_i}{\alpha_i}\left(\lambda^{\alpha_i - 1} - \lambda^{-\alpha_i - 1}\right) \tag{5}$$

As for the model fitting I've used only the uniaxial data in this task. To calculate the parameters of $\mu_1, \mu_2, \alpha_1, \alpha_2$ I've used *Python* programming language in which I've used the *minimize* from *scipy.optimize*. For the best solution I've tried different methods simultaniously to achieve the best quality result for the paramters. The resulting parameters are showed in Table 2.

Table 2: Resulting Parameters for Uniaxial Data Model Fitting

Material Parameters	
μ_1	10.5544
μ_2	2.1012
α_1	-0.7732
α_2	4.9987

The visualization of the solution is displayed in Figure 2.

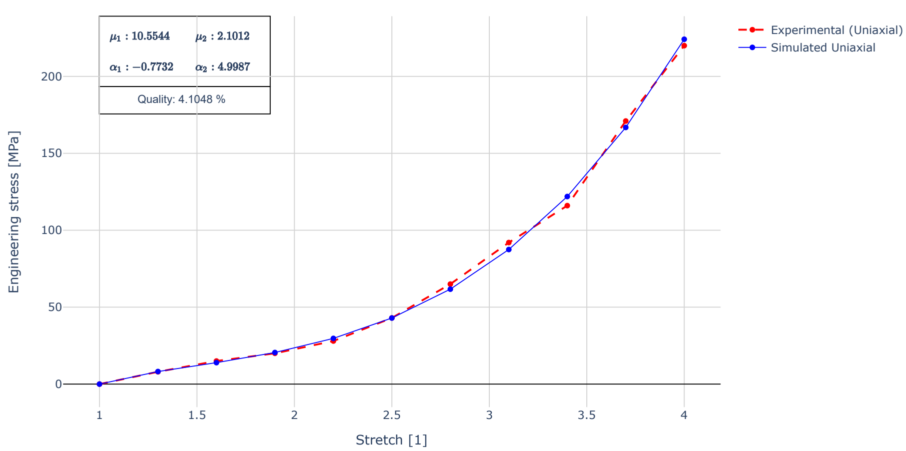

Figure 2: Uniaxial-Only Model and Uniaxial Data Set Comparison

3 Task - Quality in case of uniaxial data model fitting

In the second task, parameter selection was optimized to minimize the uniaxial quality function (Q_U), leading to the exclusion of other datasets. Consequently, higher anticipated values for Q_B and Q_P were expected. However, unexpectedly, the planar quality function exhibited a superior result compared to the uniaxial.

Table 3: Quality Function Results for uniaxial data model fitted parameters

	Uniaxial	**Equibiaxial**	**Planar**
Quality [%]	4.1048	9.4568	3.3923

4 Task - Model accuracy and conclusions

In general, it can be stated that the optimally tuned parameters based solely on uniaxial measurement results are insufficient to cover the values of engineering stresses in case of equibiaxial or planar extensions. Consequently, a quality value below 5% cannot be guaranteed for the model. To achieve improved results, it is more prudent to fit the parameters $\mu_1, \mu_2, \alpha_1, \alpha_2$ based on the entire set of measurement data which will be calculated and proved later on.

Figure 3 and 4 visualize the resulting solution.

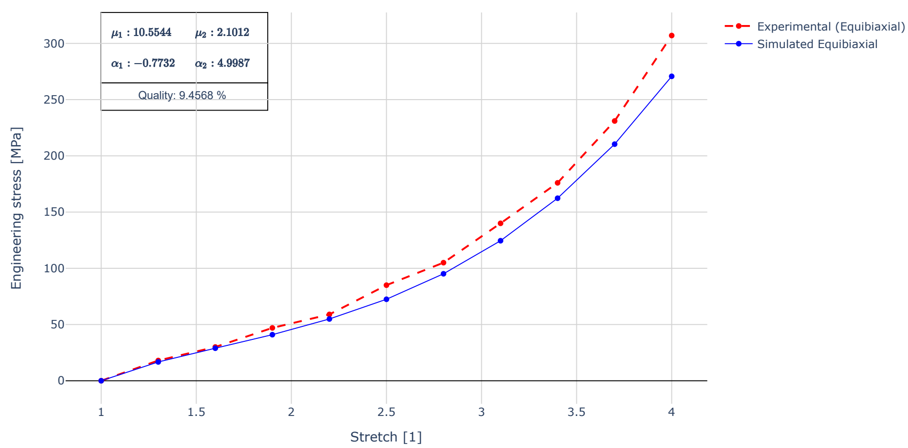

Figure 3: Uniaxial-Only Model and Equibiaxial Data Set Comparison

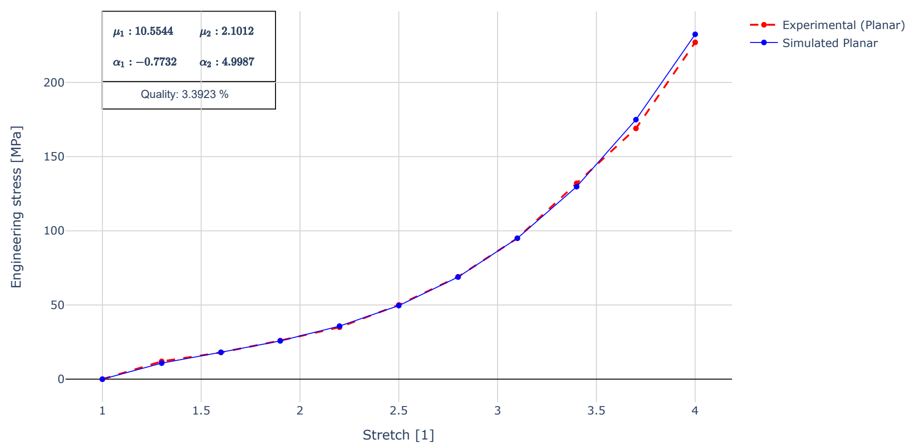

Figure 4: Uniaxial-Only Model and Planar Data Set Comparison

5 Task - Model Fitting to All Experimental Data

When fitting to all experimental - measured data a summarization of quality function should be used. The new used quality function value is the mean of the uniaxial, quibiaxial and planar quality functiuns value.

$$Q = \frac{Q_U + Q_B + Q_P}{3} \tag{6}$$

Running the same algorith that was used before on all data resulting different parameter set that is listed in Table 4.

Table 4: Resulting Parameters for All Data Model Fitting

Material Parameters		Difference [%]
μ_1	11.0175	4.39
μ_2	1.9718	-6.16
α_1	-0.9609	24.28
α_2	5.0453	0.93

It is intresting to point out that the parameters (on average) for fitting to every data is changed only for 8.94 % compared to the Only-Uniaxial Model.

Table 5: Quality Function Results to All Data Model Fitted Parameters

	All Data	**Uniaxial**	**Equibiaxial**	**Planar**
Quality [%]	3.4626	4.2447	3.0041	3.1389

6 Task - All Data Fitted Model Accuracy and Conclusions

The accuracy can be described the best with the mean quality function value which is 3.4626 %, in comparison when fitting only to the uniaxial extension engineer stress data a mean quality function value of 5.6513 % was achived. This shows that unsuprisingly fitting to all data is more accurate when speaking of the whole material model.

The visualization of the solution is displayed in Figure 5 - 7.

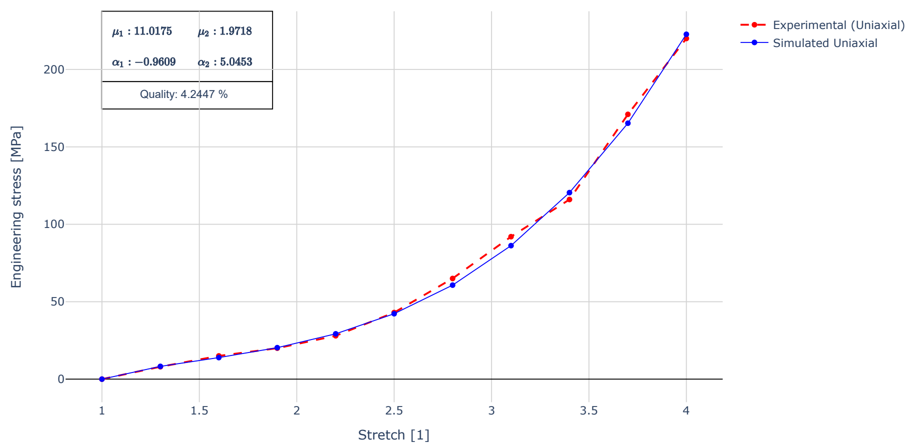

Figure 5: Comprehensive Model and Uniaxial Data Set Comparison

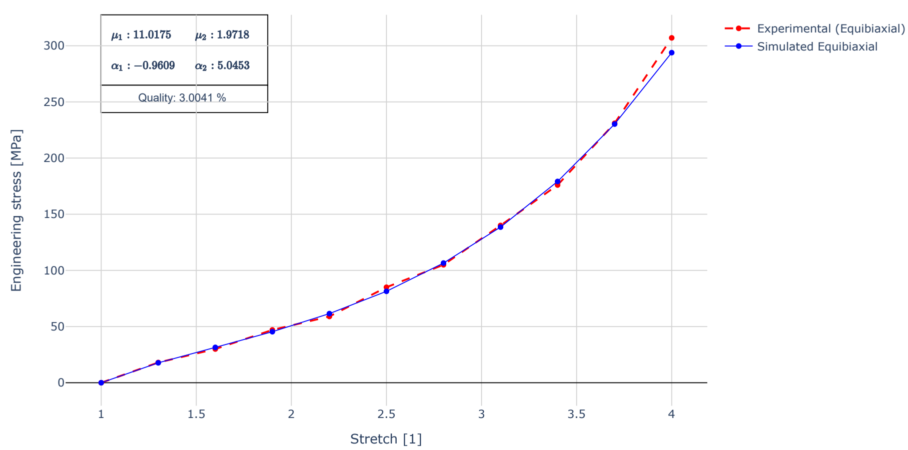

Figure 6: Comprehensive Model and Equibiaxial Data Set Comparison

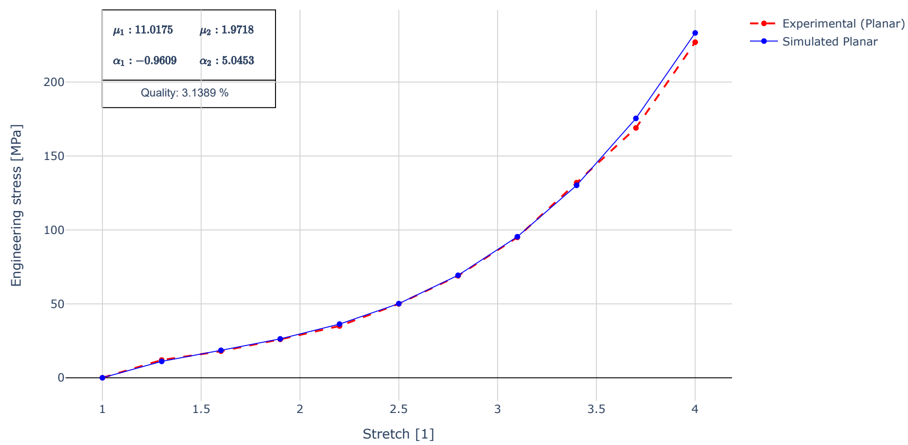

Figure 7: Comprehensive Model and Planar Data Set Comparison

7 Task - Calculation of the unknown scalar k

In this task I need to determine the unknown scalar k found in the given displacement field $\mathbf{U}$. Since the homework is about an incompressible rubber-like material thus the volume ratio $J = 1$ therefore k can be obtained if the deformation gradient ($\mathbf{F}$) is known.

Based on my data the displacement field $\mathbf{U}$ is the following:

$$\mathbf{U} = \begin{bmatrix} 2 - k \cdot Y \\ -0.6X + 0.1Y \\ -2 \end{bmatrix} \tag{7}$$

The deformation gradient ($\mathbf{F}$) is determined as

$$\mathbf{F} = \text{Grad}\,\mathbf{U} + \mathbf{I} = \begin{bmatrix} \frac{\partial \mathbf{U}_1}{\partial X} & \frac{\partial \mathbf{U}_1}{\partial Y} & \frac{\partial \mathbf{U}_1}{\partial Z} \\ \frac{\partial \mathbf{U}_2}{\partial X} & \frac{\partial \mathbf{U}_2}{\partial Y} & \frac{\partial \mathbf{U}_2}{\partial Z} \\ \frac{\partial \mathbf{U}_3}{\partial X} & \frac{\partial \mathbf{U}_3}{\partial Y} & \frac{\partial \mathbf{U}_3}{\partial Z} \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & -k & 0 \\ -0.6 & 1.1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{8}$$

$$J = 1 = \det \mathbf{F} \tag{9}$$

The resulting equation can be solved for k unknown scalar since it is the only unkown value.

$$k = 0.1667 \tag{10}$$

Substituting the k value into the deformation gradient gives it's numerical value of

$$\mathbf{F} = \begin{bmatrix} 1 & -0.1667 & 0 \\ -0.6 & 1.1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{11}$$

8 Task - Determine the principal stretches

In order to determine the principal stretches and the corresponding principal Eulerian directions obtaining the left Cauchy-Green deformation tensor is the efficient way (since we know the deformation tensor from the task before).

$$\mathbf{b} = \mathbf{F} \cdot \mathbf{F}^T = \begin{bmatrix} 1.028 & -0.7833 & 0 \\ -0.7833 & 1.57 & 0 \\ 0 & 0 & 1.0 \end{bmatrix} \tag{12}$$

Table 6: Principal Stretch and Eulerian Directions

i	Principal Stretch (λ_i)	Principal Eulerian Direction ($\mathbf{n}_i^T$)
1	0.6855	$\begin{bmatrix} -0.8146 & -0.5801 & 0 \end{bmatrix}$
2	1.4587	$\begin{bmatrix} 0.5801 & -0.8146 & 0 \end{bmatrix}$
3	1	$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$

9 Task - Cauchy stress and Mises Equivalent Stress

In this task I'll use the parameters determined in the fifth Task which is present in the Table 4. According to the homework description the used strain potential function (W) is the 2^{nd}-order incompressible Ogden's isotropic hyperelastic constitutive model which is given in the following form:

$$W = \sum_{k=1}^{2} \frac{2\mu_k}{\alpha_k^2} \left(\lambda_1^{\alpha_k} + \lambda_2^{\alpha_k} + \lambda_3^{\alpha_k} - 3\right). \tag{13}$$

Cauchy stress tensor ($\boldsymbol{\sigma}$) can be determined with the help of the strain potential function and the Left Cauchy-Green def. tensor ($\mathbf{b}$).

$$\boldsymbol{\sigma} = \frac{2}{J} \left[\left(\frac{\partial W}{\partial I_1} + I_1 \frac{\partial W}{\partial I_2} \right) \mathbf{b} - \frac{\partial W}{\partial I_2} \mathbf{b}^2 + I_3 \frac{\partial W}{\partial \lambda_3} \mathbf{I} \right] \tag{14}$$

As for the partial derivatives of the strain potential function the following equations can be obtained:

$$\frac{\partial W}{\partial \lambda_1} = 2 \left(\frac{\mu_1}{\alpha_1} \lambda_1^{\alpha_1 - 1} + \frac{\mu_2}{\alpha_2} \lambda_1^{\alpha_2 - 1} \right), \tag{15}$$

$$\frac{\partial W}{\partial \lambda_2} = 2 \left(\frac{\mu_1}{\alpha_1} \lambda_2^{\alpha_1 - 1} + \frac{\mu_2}{\alpha_2} \lambda_2^{\alpha_2 - 1} \right), \tag{16}$$

$$\frac{\partial W}{\partial \lambda_3} = 2 \left(\frac{\mu_1}{\alpha_1} \lambda_3^{\alpha_1 - 1} + \frac{\mu_2}{\alpha_2} \lambda_3^{\alpha_2 - 1} \right). \tag{17}$$

Since we are talking about incompressible isotropic case $J = 1$ and $I_3 = 1$ thus

$$W = W\left(I_1, I_2, I_3\right) \quad \rightarrow \quad W = W\left(I_1, I_2\right) \tag{18}$$

In this case only the deviatoric stress can be calculated from the constitutive equation.

$$\boldsymbol{\sigma} = \operatorname{dev}[\boldsymbol{\sigma}] + \operatorname{sph}[\boldsymbol{\sigma}] = \mathbf{s} + \mathbf{p} \tag{19}$$

$$\mathbf{s} = \operatorname{dev}\left[\sum_{i=1}^{3} \lambda_i \frac{\partial W}{\partial \lambda_i} \mathbf{n}_i \otimes \mathbf{n}_i\right] = \begin{bmatrix} -3.4953 & -10.4615 & 0 \\ -10.4615 & 3.7462 & 0 \\ 0 & 0 & -0.2509 \end{bmatrix} \text{ [MPa]} \tag{20}$$

In the context of examining a plain stress state, the determination of the pressure field ($\mathbf{p}$) is achieved through the imposition of a boundary condition, specifically requiring the stress in the z-direction to be zero ($\sigma_z = \tau_{xz} = \tau_{yz} = 0$), in accordance with the planar stress nature of the problem.

$$\mathbf{p} = -\mathbf{s}_{33}\, \mathbf{I} = \begin{bmatrix} 0.2509 & 0 & 0 \\ 0 & 0.2509 & 0 \\ 0 & 0 & 0.2509 \end{bmatrix} \text{ [MPa]} \tag{21}$$

Since we know the deviatoric and the spherical part of the tensor, the Cauchy stress tensor can be obtained as

$$\boldsymbol{\sigma} = \mathbf{s} + \mathbf{p} = \begin{bmatrix} -3.2443 & -10.4615 & 0 \\ -10.4615 & 3.9972 & 0 \\ 0 & 0 & 0 \end{bmatrix} \text{ [MPa].} \tag{22}$$

Regarding the Mises equivalent stress, with the help of the deviatoric part of the Cauchy stress tensor it can be calculated as follows:

$$\sigma_{\mathrm{VM}} = \sqrt{\frac{3}{2} \operatorname{tr}\left(\mathbf{s} \otimes \mathbf{s}\right)} = 19.1782 \text{ [MPa].} \tag{23}$$

10 Appendix

Continuum Mechanics 2 HW

November 29, 2023

Reset stored data

[6]:
```
%reset -f
```

Imports

[7]:
```
import plotly.graph_objects as go
import numpy as np
import os
import plotly.io as pio
import sympy as sp

from sympy import symbols, Matrix, eye, det, Eq, solve, latex
from IPython.display import display, Markdown
from scipy.optimize import minimize
from lmfit import Model
```

1 Data

[104]:
```
# Data for uniaxial test
uniaxial_test_data = [
    ["Eng. Strain [%]", "Eng. Stress [MPa]"],
    [0, 0],
    [30, 8],
    [60, 15],
    [90, 20],
    [120, 28],
    [150, 43],
    [180, 65],
    [210, 92],
    [240, 116],
    [270, 171],
    [300, 220]
]

# Data for equibiaxial test
equibiaxial_test_data = [
    ["Eng. Strain [%]", "Eng. Stress [MPa]"],
```

```
    [0, 0],
    [30, 18],
    [60, 30],
    [90, 47],
    [120, 59],
    [150, 85],
    [180, 105],
    [210, 140],
    [240, 176],
    [270, 231],
    [300, 307]
]

# Data for planar test
planar_test_data = [
    ["Eng. Strain [%]", "Eng. Stress [MPa]"],
    [0, 0],
    [30, 12],
    [60, 18],
    [90, 26],
    [120, 35],
    [150, 50],
    [180, 69],
    [210, 95],
    [240, 132],
    [270, 169],
    [300, 227]
]
```

2 Tasks

2.1 Task 1

Plot for epsilon

[105]:
```
# Extract data for each test
uniaxial_x = [row[0] for row in uniaxial_test_data[1:]]
uniaxial_y = [row[1] for row in uniaxial_test_data[1:]]

equibiaxial_x = [row[0] for row in equibiaxial_test_data[1:]]
equibiaxial_y = [row[1] for row in equibiaxial_test_data[1:]]

planar_x = [row[0] for row in planar_test_data[1:]]
planar_y = [row[1] for row in planar_test_data[1:]]

# Create traces for each test with lines and markers
```

```
trace_uniaxial = go.Scatter(x=uniaxial_x, y=uniaxial_y, mode='lines+markers',␣
↪name='Uniaxial Test')
trace_equibiaxial = go.Scatter(x=equibiaxial_x, y=equibiaxial_y,␣
↪mode='lines+markers', name='Equibiaxial Test')
trace_planar = go.Scatter(x=planar_x, y=planar_y, mode='lines+markers',␣
↪name='Planar Test')

# Create layout
layout = go.Layout(
    title='Stress-Strain Curves for Different Tests',
    xaxis=dict(title='Eng. Strain [%]'),
    yaxis=dict(title='Eng. Stress [MPa]')
)

# Create figure
fig = go.Figure(data=[trace_uniaxial, trace_equibiaxial, trace_planar],␣
↪layout=layout)

# Show the plot
fig.show()
```

Plot for Lambda

[106]:
```
# Calculate lambda values
calculate_lambda = lambda strain: strain / 100 + 1

# Calculate lambda values for each test
lambda_uniaxial = [calculate_lambda(row[0]) for row in uniaxial_test_data[1:]]
lambda_equibiaxial = [calculate_lambda(row[0]) for row in␣
↪equibiaxial_test_data[1:]]
lambda_planar = [calculate_lambda(row[0]) for row in planar_test_data[1:]]

# Extract y values for each test
uniaxial_y = [row[1] for row in uniaxial_test_data[1:]]
equibiaxial_y = [row[1] for row in equibiaxial_test_data[1:]]
planar_y = [row[1] for row in planar_test_data[1:]]

# Create traces for each test with lines and markers
trace_uniaxial = go.Scatter(x=lambda_uniaxial, y=uniaxial_y,␣
↪mode='lines+markers', name='Uniaxial Test')
trace_equibiaxial = go.Scatter(x=lambda_equibiaxial, y=equibiaxial_y,␣
↪mode='lines+markers', name='Equibiaxial Test')
trace_planar = go.Scatter(x=lambda_planar, y=planar_y, mode='lines+markers',␣
↪name='Planar Test')

# Create layout
layout = go.Layout(
```

```
    xaxis=dict(title='Stretch [1]', tickmode='array', tickvals=[1, 1.5, 2, 2.5,␣
↪3, 3.5, 4]),
    yaxis=dict(title='Eng. Stress [MPa]')
)

# Create figure
fig = go.Figure(data=[trace_uniaxial, trace_equibiaxial, trace_planar],␣
↪layout=layout)

# Show the plot
fig.show()

fig.write_html("plot_task1.html")
pio.write_image(fig, "plot_task1.pdf", width=1400, height=600)
fig.write_image("plot_task1.jpg", width=1400, height=600)
```

2.2 Task 2

[107]:
```
# Experimental data
lambda_exp_uniaxial = np.array(lambda_uniaxial)
lambda_exp_equibiaxial = np.array(lambda_equibiaxial)
lambda_exp_planar = np.array(lambda_planar)

P_exp_uniaxial = np.array(uniaxial_y)
P_exp_equibiaxial = np.array(equibiaxial_y)
P_exp_planar = np.array(planar_y)

# Define the W function
def W(lambdas, mu1, mu2, alpha1, alpha2):
    term1 = 2 * mu1 / alpha1**2 * (lambdas[0]**alpha1 + lambdas[1]**alpha1 +␣
↪lambdas[2]**alpha1 - 3)
    term2 = 2 * mu2 / alpha2**2 * (lambdas[0]**alpha2 + lambdas[1]**alpha2 +␣
↪lambdas[2]**alpha2 - 3)
    return term1 + term2

# Define the partial derivative functions
def P_U(lambdas, mu1, mu2, alpha1, alpha2):
    term1 = 2 * mu1 / alpha1 * (lambdas**(alpha1-1) - lambdas**((-alpha1/2) -␣
↪1))
    term2 = 2 * mu2 / alpha2 * (lambdas**(alpha2-1) - lambdas**((-alpha2/2) -␣
↪1))
    return term1 + term2

def P_B(lambdas, mu1, mu2, alpha1, alpha2):
    term1 = 2 * mu1 / alpha1 * (lambdas**(alpha1-1) - lambdas**((-2*alpha1) -␣
↪1))
```

```python
    term2 = 2 * mu2 / alpha2 * (lambdas**(alpha2-1) - lambdas**((-2*alpha2) -
 ↪1))
    return term1 + term2

def P_P(lambdas, mu1, mu2, alpha1, alpha2):
    term1 = 2 * mu1 / alpha1 * (lambdas**(alpha1-1) - lambdas**((-alpha1) - 1))
    term2 = 2 * mu2 / alpha2 * (lambdas**(alpha2-1) - lambdas**((-alpha2) - 1))
    return term1 + term2

def quality_function(params, lambdas, P_exp, P_function):
    mu1, mu2, alpha1, alpha2 = params
    P_sim = P_function(lambdas, mu1, mu2, alpha1, alpha2)
    error = (P_exp - P_sim) / (P_exp + 1e-10)
    return np.sqrt(np.mean(error**2))

# Function to calculate stress using optimal parameters
def calculate_stress_uniaxial(lambdas, mu1, mu2, alpha1, alpha2):
    P_sim = P_U(lambdas, mu1, mu2, alpha1, alpha2)
    return P_sim

def calculate_stress_equibiaxial(lambdas, mu1, mu2, alpha1, alpha2):
    P_sim = P_B(lambdas, mu1, mu2, alpha1, alpha2)
    return P_sim

def calculate_stress_planar(lambdas, mu1, mu2, alpha1, alpha2):
    P_sim = P_P(lambdas, mu1, mu2, alpha1, alpha2)
    return P_sim

# Fitting function
def fit_function(lambdas, mu1, mu2, alpha1, alpha2):
    return calculate_stress_uniaxial(lambdas, mu1, mu2, alpha1, alpha2)
```

[109]:
```python
min_error_value = float('inf')
best_method = None
best_params = None

# Define the optimization methods
methods = ['Nelder-Mead', 'BFGS', 'L-BFGS-B', 'Powell', 'COBYLA']

# Initial guess
mu1 = 0.1
mu2 = 0.1
alpha1 = -2
alpha2 = 1

# Initial parameters for optimization
initial_params = [mu1, mu2, alpha1, alpha2]
```

```
model = Model(fit_function)
model.set_param_hint('mu1', value=mu1)
model.set_param_hint('mu2', value=mu2)
model.set_param_hint('alpha1', value=alpha1)
model.set_param_hint('alpha2', value=alpha2)

# Loop through each optimization method
for method in methods:
    max_iter = 15000

    # Perform optimization with increased iterations using P_U
    result_optimize = minimize(quality_function, initial_params,␣
↪args=(lambda_exp_uniaxial, P_exp_uniaxial, P_U), method=method,␣
↪options={'maxiter': max_iter})
    optimal_params_P_U = result_optimize.x

    # Calculate error value for P_U using quality_function
    error_value_P_U = quality_function(optimal_params_P_U, lambda_exp_uniaxial,␣
↪P_exp_uniaxial, P_U) * 100

    # Update minimum error value and best method for P_U
    if error_value_P_U < min_error_value:
        min_error_value = error_value_P_U
        best_method = method
        best_params = optimal_params_P_U

    # Print error value for P_U
    print(f"\nOptimal Parameters ({method}, P_U): {optimal_params_P_U}")
    print(f"Error Value ({method}, P_U): {error_value_P_U:.4f} % with␣
↪{max_iter} iterations")

# Set the maximum number of iterations for lmfit
max_iter_lmfit = 15000

# Fit the model to uniaxial data using lmfit with increased iterations
result_lmfit = model.fit(P_exp_uniaxial, lambdas=lambda_exp_uniaxial,␣
↪method='leastsq', iter_cb=None, max_nfev=max_iter_lmfit)
optimal_params_fit_lmfit = [result_lmfit.best_values[f] for f in ['mu1', 'mu2',␣
↪'alpha1', 'alpha2']]

# Calculate error value for lmfit method using quality_function
error_value_lmfit = quality_function(optimal_params_fit_lmfit,␣
↪lambda_exp_uniaxial, P_exp_uniaxial, P_U) * 100
if error_value_lmfit < min_error_value:
    min_error_value = error_value_lmfit
```

```
    best_method = 'lmfit'
    best_params = optimal_params_fit_lmfit

print(f"\nOptimal Parameters (lmfit, P_U): {optimal_params_fit_lmfit}")
print(f"Error Value (lmfit, P_U): {error_value_lmfit:.4f} % with␣
↪{max_iter_lmfit} iterations")

# Print the best method
print(f"\nBest Method: {best_method} (Error Value: {min_error_value:.4f} %)")
print(f"Optimal Parameters for the Best Method (P_U): {best_params}")
```

```
Optimal Parameters (Nelder-Mead, P_U): [10.55439559  2.1012538  -0.77321806
4.99867097]
Error Value (Nelder-Mead, P_U): 4.0832 % with 15000 iterations

Optimal Parameters (BFGS, P_U): [10.55441842  2.10122789 -0.77316348
4.99868234]
Error Value (BFGS, P_U): 4.0832 % with 15000 iterations

Optimal Parameters (L-BFGS-B, P_U): [10.55445786  2.10124775 -0.77327501
4.99867407]
Error Value (L-BFGS-B, P_U): 4.0832 % with 15000 iterations

Optimal Parameters (Powell, P_U): [15.66362192 -2.79230196 -7.8670914
-7.86616575]
Error Value (Powell, P_U): 10.7635 % with 15000 iterations

Optimal Parameters (COBYLA, P_U): [10.17868332  2.08433767 -0.35084183
5.00342576]
Error Value (COBYLA, P_U): 4.1048 % with 15000 iterations

Optimal Parameters (lmfit, P_U): [5.057573082318104, 9.432191162038052,
-9.66295228421433, -1.8570238702943205]
Error Value (lmfit, P_U): 4.4795 % with 15000 iterations

Best Method: BFGS (Error Value: 4.0832 %)
Optimal Parameters for the Best Method (P_U): [10.55441842  2.10122789
-0.77316348  4.99868234]
```

[110]:

```
# Plot the experimental and simulated uniaxial stress
fig = go.Figure()

# Experimental data
fig.add_trace(go.Scatter(x=lambda_exp_uniaxial, y=P_exp_uniaxial,␣
↪mode='lines+markers', name='Experimental (Uniaxial)', line=dict(color='red',␣
↪dash='dash')))
```

```python
# Simulated data using the best method
simulated_uniaxial_stress = calculate_stress_uniaxial(lambda_exp_uniaxial,␣
↪*best_params)
fig.add_trace(go.Scatter(x=lambda_exp_uniaxial, y=simulated_uniaxial_stress,␣
↪mode='lines+markers', name=f'Simulated Uniaxial', line=dict(color='blue')))

# Latex annotations for parameters and quality function
latex_annotations_mu1 = f"$\\mu_1: {best_params[0]:.4f}$"
latex_annotations_mu2 = f"$\\mu_2: {best_params[1]:.4f}$"
latex_annotations_alpha1 = f"$\\alpha_1: {best_params[2]:.4f}$"
latex_annotations_alpha2 = f"$\\alpha_2: {best_params[3]:.4f}$"

fig.update_layout(
    showlegend=True,
    plot_bgcolor='white',  # Set background color to white
    xaxis=dict(title='Stretch [1]', gridcolor='lightgrey', zeroline=True,␣
↪zerolinewidth=1, zerolinecolor='black'),  # Set grid color to lightgrey,␣
↪x-axis line to black
    yaxis=dict(title='Engineering stress [MPa]', gridcolor='lightgrey',␣
↪zeroline=True, zerolinewidth=1, zerolinecolor='black'),  # Set grid color to␣
↪lightgrey, y-axis line to black
    annotations=[
        dict(text=latex_annotations_mu1, x=0.07, y=0.97, xref='paper',␣
↪yref='paper', showarrow=False, align='left', font=dict(family='Arial,␣
↪sans-serif')),
        dict(text=latex_annotations_mu2, x=0.20, y=0.97, xref='paper',␣
↪yref='paper', showarrow=False, align='left', font=dict(family='Arial,␣
↪sans-serif')),
        dict(text=latex_annotations_alpha1, x=0.07, y=0.89, xref='paper',␣
↪yref='paper', showarrow=False, align='left', font=dict(family='Arial,␣
↪sans-serif')),
        dict(text=latex_annotations_alpha2, x=0.20, y=0.89, xref='paper',␣
↪yref='paper', showarrow=False, align='left', font=dict(family='Arial,␣
↪sans-serif')),
    ],
    shapes=[
        dict(
            type='rect',
            xref='paper',
            yref='paper',
            x0=0.055,
            y0=0.82,
            x1=0.315,
            y1=1,
            fillcolor='white',
```

```
            opacity=1,
            layer='below',
            line=dict(color='black', width=2)
        ),
    dict(
            type='rect',
            xref='paper',
            yref='paper',
            x0=0.055,
            y0=0.75,
            x1=0.315,
            y1=0.82,
            fillcolor='white',
            opacity=1,
            layer='below',
            line=dict(color='black', width=2)
        ),
    ]
)

fig.add_annotation(
    text=f"Quality: {error_value_P_U:.4f} %",
    x=0.11, y=0.81,
    xref='paper', yref='paper',
    showarrow=False,
    align='left',
    font=dict(family='Arial, sans-serif')
)

# Show the plot
fig.show()
```

[111]:
```
output_folder = "Cont_mecha_2HW"

# Save the equibiaxial plot as HTML, PDF, and JPG
uniaxial_html_path = os.path.join(output_folder, "uniaxial_plot.html")
uniaxial_pdf_path = os.path.join(output_folder, "uniaxial_plot.pdf")
uniaxial_jpg_path = os.path.join(output_folder, "uniaxial_plot.jpg")

fig.write_html(uniaxial_html_path)
pio.write_image(fig, uniaxial_pdf_path, width=1000, height=600)
fig.write_image(uniaxial_jpg_path, width=1200, height=600)
```

2.3 Task 3

[113]:
```
error_value_P_B = quality_function(best_params, lambda_exp_equibiaxial,␣
↪P_exp_equibiaxial, P_B) * 100
print(f"\nQuality for P_B: {error_value_P_B:.4f} %")

# Calculate the quality for P_P
error_value_P_P = quality_function(best_params, lambda_exp_planar,␣
↪P_exp_planar, P_P) * 100
print(f"Quality for P_P: {error_value_P_P:.4f} %")
```

```
Quality for P_B: 9.4568 %
Quality for P_P: 3.3923 %
```

2.4 Task 4

[114]:
```
# Calculate the quality for P_B
error_value_P_B = quality_function(best_params, lambda_exp_equibiaxial,␣
↪P_exp_equibiaxial, P_B) * 100
print(f"\nQuality for P_B: {error_value_P_B:.4f} %")

# Calculate the quality for P_P
error_value_P_P = quality_function(best_params, lambda_exp_planar,␣
↪P_exp_planar, P_P) * 100
print(f"Quality for P_P: {error_value_P_P:.4f} %")

# Plot the equibiaxial stress
fig_equibiaxial = go.Figure()

# Experimental data
fig_equibiaxial.add_trace(go.Scatter(x=lambda_exp_equibiaxial,␣
↪y=P_exp_equibiaxial, mode='lines+markers', name='Experimental␣
↪(Equibiaxial)', line=dict(color='red', dash='dash')))

# Simulated data using the best method
simulated_equibiaxial_stress =␣
↪calculate_stress_equibiaxial(lambda_exp_equibiaxial, *best_params)
fig_equibiaxial.add_trace(go.Scatter(x=lambda_exp_equibiaxial,␣
↪y=simulated_equibiaxial_stress, mode='lines+markers', name=f'Simulated␣
↪Equibiaxial', line=dict(color='blue')))

fig_equibiaxial.update_layout(
    showlegend=True,
    plot_bgcolor='white',
    xaxis=dict(title='Stretch [1]', gridcolor='lightgrey', zeroline=True,␣
↪zerolinewidth=1, zerolinecolor='black'),
```

```
    yaxis=dict(title='Engineering stress [MPa]', gridcolor='lightgrey',␣
↪zeroline=True, zerolinewidth=1, zerolinecolor='black'),
    annotations=[
        dict(text=latex_annotations_mu1, x=0.07, y=0.97, xref='paper',␣
↪yref='paper', showarrow=False, align='left', font=dict(family='Arial,␣
↪sans-serif')),
        dict(text=latex_annotations_mu2, x=0.20, y=0.97, xref='paper',␣
↪yref='paper', showarrow=False, align='left', font=dict(family='Arial,␣
↪sans-serif')),
        dict(text=latex_annotations_alpha1, x=0.07, y=0.89, xref='paper',␣
↪yref='paper', showarrow=False, align='left', font=dict(family='Arial,␣
↪sans-serif')),
        dict(text=latex_annotations_alpha2, x=0.20, y=0.89, xref='paper',␣
↪yref='paper', showarrow=False, align='left', font=dict(family='Arial,␣
↪sans-serif')),
    ],
    shapes=[
        dict(
            type='rect',
            xref='paper',
            yref='paper',
            x0=0.055,
            y0=0.82,
            x1=0.315,
            y1=1,
            fillcolor='white',
            opacity=1,
            layer='below',
            line=dict(color='black', width=2)
        ),
        dict(
            type='rect',
            xref='paper',
            yref='paper',
            x0=0.055,
            y0=0.75,
            x1=0.315,
            y1=0.82,
            fillcolor='white',
            opacity=1,
            layer='below',
            line=dict(color='black', width=2)
        ),
    ]
)

fig_equibiaxial.add_annotation(
```

```
    text=f"Quality: {error_value_P_B:.4f} %",
    x=0.11, y=0.81,
    xref='paper', yref='paper',
    showarrow=False,
    align='left',
    font=dict(family='Arial, sans-serif')
)

fig_equibiaxial.show()

fig_planar = go.Figure()

fig_planar.add_trace(go.Scatter(x=lambda_exp_planar, y=P_exp_planar,␣
↪mode='lines+markers', name='Experimental (Planar)', line=dict(color='red',␣
↪dash='dash')))

# Simulated data using the best method
simulated_planar_stress = calculate_stress_planar(lambda_exp_planar,␣
↪*best_params)
fig_planar.add_trace(go.Scatter(x=lambda_exp_planar, y=simulated_planar_stress,␣
↪mode='lines+markers', name=f'Simulated Planar', line=dict(color='blue')))

# Layout and annotations
fig_planar.update_layout(
    #title='Experimental vs. Simulated Planar Stress',
    showlegend=True,
    plot_bgcolor='white',
    xaxis=dict(title='Stretch [1]', gridcolor='lightgrey', zeroline=True,␣
↪zerolinewidth=1, zerolinecolor='black'),
    yaxis=dict(title='Engineering stress [MPa]', gridcolor='lightgrey',␣
↪zeroline=True, zerolinewidth=1, zerolinecolor='black'),
    annotations=[
        dict(text=latex_annotations_mu1, x=0.07, y=0.97, xref='paper',␣
↪yref='paper', showarrow=False, align='left', font=dict(family='Arial,␣
↪sans-serif')),
        dict(text=latex_annotations_mu2, x=0.20, y=0.97, xref='paper',␣
↪yref='paper', showarrow=False, align='left', font=dict(family='Arial,␣
↪sans-serif')),
        dict(text=latex_annotations_alpha1, x=0.07, y=0.89, xref='paper',␣
↪yref='paper', showarrow=False, align='left', font=dict(family='Arial,␣
↪sans-serif')),
        dict(text=latex_annotations_alpha2, x=0.20, y=0.89, xref='paper',␣
↪yref='paper', showarrow=False, align='left', font=dict(family='Arial,␣
↪sans-serif')),
    ],
    shapes=[
```

```
        dict(
            type='rect',
            xref='paper',
            yref='paper',
            x0=0.055,
            y0=0.82,
            x1=0.315,
            y1=1,
            fillcolor='white',
            opacity=1,
            layer='below',
            line=dict(color='black', width=2)
        ),
        dict(
            type='rect',
            xref='paper',
            yref='paper',
            x0=0.055,
            y0=0.75,
            x1=0.315,
            y1=0.82,
            fillcolor='white',
            opacity=1,
            layer='below',
            line=dict(color='black', width=2)
        ),
    ]
)

fig_planar.add_annotation(
    text=f"Quality: {error_value_P_P:.4f} %",
    x=0.11, y=0.81,
    xref='paper', yref='paper',
    showarrow=False,
    align='left',
    font=dict(family='Arial, sans-serif')
)

fig_planar.show()
```

```
Quality for P_B: 9.4568 %
Quality for P_P: 3.3923 %
```

[115]:
```
# Save the equibiaxial plot as HTML, PDF, and JPG
equibiaxial_html_path = os.path.join(output_folder, "equibiaxial_plot.html")
equibiaxial_pdf_path = os.path.join(output_folder, "equibiaxial_plot.pdf")
```

```python
equibiaxial_jpg_path = os.path.join(output_folder, "equibiaxial_plot.jpg")

fig_equibiaxial.write_html(equibiaxial_html_path)
pio.write_image(fig_equibiaxial, equibiaxial_pdf_path, width=1000, height=600)
fig_equibiaxial.write_image(equibiaxial_jpg_path, width=900, height=600)

# Save the planar plot as HTML, PDF, and JPG
planar_html_path = os.path.join(output_folder, "planar_plot.html")
planar_pdf_path = os.path.join(output_folder, "planar_plot.pdf")
planar_jpg_path = os.path.join(output_folder, "planar_plot.jpg")

fig_planar.write_html(planar_html_path)
pio.write_image(fig_planar, planar_pdf_path, width=1000, height=600)
fig_planar.write_image(planar_jpg_path, width=900, height=600)
```

[116]:

```python
# Print the values in LaTeX format using IPython.display
display(Markdown(f"Uniaxial fit parameters"))
display(Markdown(f"\n$\\mu_1: {best_params[0]:.4f}$"))
display(Markdown(f"$\\mu_2: {best_params[1]:.4f}$"))
display(Markdown(f"$\\alpha_1: {best_params[2]:.4f}$"))
display(Markdown(f"$\\alpha_2: {best_params[3]:.4f}$"))
```

Uniaxial fit parameters

$\mu_1 : 10.5544$

$\mu_2 : 2.1012$

$\alpha_1 : -0.7732$

$\alpha_2 : 4.9987$

2.5 Task 5

[117]:

```python
# Define the quality function for all measurements
def total_quality_function(params, lambdas, P_exp_uniaxial, P_exp_equibiaxial,␣
↪P_exp_planar):
    # Extract parameters
    mu1, mu2, alpha1, alpha2 = params

    # Calculate individual quality values
    Q_U = quality_function([mu1, mu2, alpha1, alpha2], lambda_exp_uniaxial,␣
↪P_exp_uniaxial, P_U)
    Q_B = quality_function([mu1, mu2, alpha1, alpha2], lambda_exp_equibiaxial,␣
↪P_exp_equibiaxial, P_B)
    Q_P = quality_function([mu1, mu2, alpha1, alpha2], lambda_exp_planar,␣
↪P_exp_planar, P_P)

    # Calculate total quality as the mean of individual qualities
```

```
    Q_total = (Q_U + Q_B + Q_P) / 3.0

    return Q_total

# Initial guess
mu1 = 0.1
mu2 = 0.1
alpha1 = -2
alpha2 = 1

# Initial parameters for optimization
initial_params = [mu1, mu2, alpha1, alpha2]

# Set the optimization methods
methods = ['Nelder-Mead', 'BFGS', 'L-BFGS-B', 'Powell', 'COBYLA']

# Initialize variables to store the minimum error value and the corresponding␣
↪method
min_error_value_total = float('inf')
best_method_total = None
best_params_total = None

# Loop through each optimization method
for method in methods:
    # Set the maximum number of iterations
    max_iter = 15000

    # Perform optimization
    result_optimize = minimize(total_quality_function, initial_params,␣
↪args=(lambda_exp_uniaxial, P_exp_uniaxial, P_exp_equibiaxial, P_exp_planar),␣
↪method=method, options={'maxiter': max_iter})
    optimal_params = result_optimize.x

    # Calculate total error value
    error_value_total = total_quality_function(optimal_params,␣
↪lambda_exp_uniaxial, P_exp_uniaxial, P_exp_equibiaxial, P_exp_planar) * 100

    # Update minimum error value and best method
    if error_value_total < min_error_value_total:
        min_error_value_total = error_value_total
        best_method_total = method
        best_params_total = optimal_params

    # Print error value
    print(f"\nOptimal Parameters ({method}, Total): {optimal_params}")
    print(f"Total Error Value ({method}, Total): {error_value_total:.4f} % with␣
↪{max_iter} iterations")
```

```
# Print the best method
print(f"\nBest Method: {best_method_total} (Error Value: {min_error_value_total:
 ↪.4f} %)")
print(f"Optimal Parameters for the Best Method (total): {best_params_total}")
```

```
Optimal Parameters (Nelder-Mead, Total): [ 1.97175181 11.01756888  5.045279
-0.96092251]
Total Error Value (Nelder-Mead, Total): 3.4626 % with 15000 iterations

Optimal Parameters (BFGS, Total): [11.01752602  1.9717665  -0.9609219
5.04527311]
Total Error Value (BFGS, Total): 3.4626 % with 15000 iterations

Optimal Parameters (L-BFGS-B, Total): [11.01743523  1.97180688 -0.96093718
5.04525032]
Total Error Value (L-BFGS-B, Total): 3.4626 % with 15000 iterations

Optimal Parameters (Powell, Total): [11.00988509  1.96443403 -0.96619812
5.04837469]
Total Error Value (Powell, Total): 3.4653 % with 15000 iterations

Optimal Parameters (COBYLA, Total): [10.86969811  2.03146934 -0.97021645
5.01789347]
Total Error Value (COBYLA, Total): 3.4724 % with 15000 iterations

Best Method: BFGS (Error Value: 3.4626 %)
Optimal Parameters for the Best Method (total): [11.01752602  1.9717665
-0.9609219   5.04527311]
```

[118]:
```
# Calculate quality values for the best method
Q_U_best = quality_function(best_params_total, lambda_exp_uniaxial,␣
 ↪P_exp_uniaxial, P_U) * 100
Q_B_best = quality_function(best_params_total, lambda_exp_equibiaxial,␣
 ↪P_exp_equibiaxial, P_B) * 100
Q_P_best = quality_function(best_params_total, lambda_exp_planar, P_exp_planar,␣
 ↪P_P) * 100

# Report quality values for each measurement
print(f"\nQuality for P_U (Best Method - Total): {Q_U_best:.4f} %")
print(f"Quality for P_B (Best Method - Total): {Q_B_best:.4f} %")
print(f"Quality for P_P (Best Method - Total): {Q_P_best:.4f} %")
```

```
Quality for P_U (Best Method - Total): 4.2447 %
Quality for P_B (Best Method - Total): 3.0041 %
Quality for P_P (Best Method - Total): 3.1389 %
```

[119]:
```
# Print the values in LaTeX format using IPython.display
display(Markdown("Total fit parameters"))
display(Markdown(f"\n$\\mu_1: {best_params_total[0]:.4f}$"))
display(Markdown(f"$\\mu_2: {best_params_total[1]:.4f}$"))
display(Markdown(f"$\\alpha_1: {best_params_total[2]:.4f}$"))
display(Markdown(f"$\\alpha_2: {best_params_total[3]:.4f}$"))
```

Total fit parameters

$\mu_1 : 11.0175$

$\mu_2 : 1.9718$

$\alpha_1 : -0.9609$

$\alpha_2 : 5.0453$

[120]:
```
np.mean((((best_params - best_params_total) / best_params * 100)**2)**(1/2))
```

[120]:
```
8.941383426874651
```

2.6 Task 6

[121]:
```
latex_annotations_mu1_total = f"$\\mu_1: {best_params_total[0]:.4f}$"
latex_annotations_mu2_total  = f"$\\mu_2: {best_params_total[1]:.4f}$"
latex_annotations_alpha1_total  = f"$\\alpha_1: {best_params_total[2]:.4f}$"
latex_annotations_alpha2_total  = f"$\\alpha_2: {best_params_total[3]:.4f}$"

# Function to generate plots for each test
def generate_and_save_plot(lambda_exp, P_exp, P_function, stress_type):
    # Calculate the quality for the test
    error_value = quality_function(best_params_total, lambda_exp, P_exp,␣
 ↪P_function) * 100
    print(f"\nQuality for {stress_type}: {error_value:.4f} %")

    # Plot the stress
    fig = go.Figure()

    # Experimental data
    fig.add_trace(go.Scatter(x=lambda_exp, y=P_exp, mode='lines+markers',␣
 ↪name=f'Experimental ({stress_type})', line=dict(color='red', dash='dash')))

    # Simulated data using the total fitted parameters
    simulated_stress = P_function(lambda_exp, *best_params_total)
    fig.add_trace(go.Scatter(x=lambda_exp, y=simulated_stress,␣
 ↪mode='lines+markers', name=f'Simulated {stress_type}',␣
 ↪line=dict(color='blue')))

    # Layout and annotations
    fig.update_layout(
```

```
        #title=f'Experimental vs. Simulated {stress_type} Stress',
        showlegend=True,
        plot_bgcolor='white',
        xaxis=dict(title='Stretch [1]', gridcolor='lightgrey', zeroline=True,␣
↪zerolinewidth=1, zerolinecolor='black'),
        yaxis=dict(title='Engineering stress [MPa]', gridcolor='lightgrey',␣
↪zeroline=True, zerolinewidth=1, zerolinecolor='black'),
        annotations=[
            dict(text=latex_annotations_mu1_total, x=0.07, y=0.97,␣
↪xref='paper', yref='paper', showarrow=False, align='left',␣
↪font=dict(family='Arial, sans-serif')),
            dict(text=latex_annotations_mu2_total, x=0.20, y=0.97,␣
↪xref='paper', yref='paper', showarrow=False, align='left',␣
↪font=dict(family='Arial, sans-serif')),
            dict(text=latex_annotations_alpha1_total, x=0.07, y=0.89,␣
↪xref='paper', yref='paper', showarrow=False, align='left',␣
↪font=dict(family='Arial, sans-serif')),
            dict(text=latex_annotations_alpha2_total, x=0.20, y=0.89,␣
↪xref='paper', yref='paper', showarrow=False, align='left',␣
↪font=dict(family='Arial, sans-serif')),
        ],
        shapes=[
            dict(
                type='rect',
                xref='paper',
                yref='paper',
                x0=0.055,
                y0=0.82,
                x1=0.315,
                y1=1,
                fillcolor='white',
                opacity=1,
                layer='below',
                line=dict(color='black', width=2)
            ),
            dict(
                type='rect',
                xref='paper',
                yref='paper',
                x0=0.055,
                y0=0.75,
                x1=0.315,
                y1=0.82,
                fillcolor='white',
                opacity=1,
                layer='below',
```

```
                line=dict(color='black', width=2)
            ),
        ]
    )

    fig.add_annotation(
        text=f"Quality: {error_value:.4f} %",
        x=0.11, y=0.81,
        xref='paper', yref='paper',
        showarrow=False,
        align='left',
        font=dict(family='Arial, sans-serif')
    )
    # Save the plot
    save_folder = "Cont_mecha_2HW"
    if not os.path.exists(save_folder):
        os.makedirs(save_folder)

    html_path = os.path.join(save_folder, f"{stress_type.lower()}_plot_total.
↪html")
    pdf_path = os.path.join(save_folder, f"{stress_type.lower()}_plot_total.
↪pdf")
    jpg_path = os.path.join(save_folder, f"{stress_type.lower()}_plot_total.
↪jpg")

    fig.write_html(html_path)
    pio.write_image(fig, pdf_path, width=1000, height=600)
    fig.write_image(jpg_path, width=900, height=600)

    # Display the plot
    fig.show()

# Generate and save plots for each test
generate_and_save_plot(lambda_exp_equibiaxial, P_exp_equibiaxial,␣
↪calculate_stress_equibiaxial, 'Equibiaxial')
generate_and_save_plot(lambda_exp_planar, P_exp_planar,␣
↪calculate_stress_planar, 'Planar')
generate_and_save_plot(lambda_exp_uniaxial, P_exp_uniaxial,␣
↪calculate_stress_uniaxial, 'Uniaxial')
```

```
Quality for Equibiaxial: 3.0041 %

Quality for Planar: 3.1389 %

Quality for Uniaxial: 4.2447 %
```

2.7 Task 7

[152]:
```
# Define symbolic variables
X, Y, Z, k = symbols('X Y Z k')

# Displacement equations
ux_equation = 2 - k * Y
uy_equation = -0.6 * X + 0.1 * Y
uz_equation = -2

# Create the displacement vector U as a column vector
U = Matrix([ux_equation, uy_equation, uz_equation])

# Calculate the gradient of U (F = Grad U)
F = U.jacobian([X, Y, Z]) + eye(3)
F
```

[152]:

$$\begin{bmatrix} 1 & -k & 0 \\ -0.6 & 1.1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

[153]:
```
# Calculate the determinant of F (J = det(F))
J = det(F)

# Solve for the parameter k such that J = 1
equation = Eq(J, 1)
solutions = solve(equation, k)
sol = solutions[0]
```

[154]:
```
# Print the values in LaTeX format using IPython.display
display(Markdown(f"Displacement vector :\n${latex(U)}$"))
display(Markdown(f"\nF matrix: ${latex(F)}$"))
display(Markdown(f"\nDeterminant $J$: \n${latex(J)}$"))
display(Markdown(f"\nParameter $k$ (solution for $J = 1$): \n$k = 
↪{latex(sol)}$"))
```

Displacement vector : $\begin{bmatrix} -Yk+2 \\ -0.6X+0.1Y \\ -2 \end{bmatrix}$

F matrix: $\begin{bmatrix} 1 & -k & 0 \\ -0.6 & 1.1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Determinant J: $1.1 - 0.6k$

Parameter k (solution for $J = 1$): $k = 0.166666666666667$

2.8 Task 8

[155]:
```
F.subs(k,sol)
```

[155]:

$$\begin{bmatrix} 1 & -0.166666666666667 & 0 \\ -0.6 & 1.1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

[156]:
```
C = F.transpose() * F
C_val = C.subs(k, sol)

b = F*F.transpose()
b_val = b.subs(k, sol)
b_val.applyfunc(lambda x: sp.N(x, 4))
```

[156]:

$$\begin{bmatrix} 1.028 & -0.7833 & 0 \\ -0.7833 & 1.57 & 0 \\ 0 & 0 & 1.0 \end{bmatrix}$$

[158]:
```
(eigenVALS, eigenVECS) = np.linalg.eig(np.array(b_val, dtype=float))
eigenVALS_root = eigenVALS**(1/2)
```

2.9 Task 9

[160]:
```
# Define symbols
mu1, mu2, alpha1, alpha2, lamb = sp.symbols('mu1 mu2 alpha1 alpha2 lamb')

# Compute the partial derivatives
partial_W_partial_lambda1 = (2*(mu1/alpha1 * lamb**(alpha1-1) + mu2/alpha2 *␣
↪lamb**(alpha2-1))).subs({mu1: best_params_total[0], mu2:␣
↪best_params_total[1], alpha1: best_params_total[2], alpha2:␣
↪best_params_total[3], lamb: eigenVALS_root[0]})
partial_W_partial_lambda2 = (2*(mu1/alpha1 * lamb**(alpha1-1) + mu2/alpha2 *␣
↪lamb**(alpha2-1))).subs({mu1: best_params_total[0], mu2:␣
↪best_params_total[1], alpha1: best_params_total[2], alpha2:␣
↪best_params_total[3], lamb: eigenVALS_root[1]})
partial_W_partial_lambda3 = (2*(mu1/alpha1 * lamb**(alpha1-1) + mu2/alpha2 *␣
↪lamb**(alpha2-1))).subs({mu1: best_params_total[0], mu2:␣
↪best_params_total[1], alpha1: best_params_total[2], alpha2:␣
↪best_params_total[3], lamb: eigenVALS_root[2]})
```

[162]:
```
sigma_new = eigenVALS_root[0] * partial_W_partial_lambda1 * sp.
↪Matrix(eigenVECS[:,0]) * sp.Matrix(eigenVECS[:,0]).T + eigenVALS_root[1] *␣
↪partial_W_partial_lambda2 * sp.Matrix(eigenVECS[:,1]) * sp.Matrix(eigenVECS[:
↪,1]).T + eigenVALS_root[2] * partial_W_partial_lambda3 * sp.
↪Matrix(eigenVECS[:,2]) * sp.Matrix(eigenVECS[:,2]).T
sigma_new
```

[162]:

$$\begin{bmatrix} -25.3938375804359 & -10.4615353513333 & 0 \\ -10.4615353513333 & -18.1523776493002 & 0 \\ 0 & 0 & -22.149528954286 \end{bmatrix}$$

[172]:
```
sp.Matrix(eigenVECS[:,2])
```

[172]:
$$\begin{bmatrix} 0 \\ 0 \\ 1.0 \end{bmatrix}$$

[164]:
```
s_new = sigma_new - 1/3*(sigma_new[0,0] + sigma_new[1,1] + sigma_new[2,2]) *␣
↪eye(3)
p = -s_new[2,2]
s_new
```

[164]:
$$\begin{bmatrix} -3.49525618576188 & -10.4615353513333 & 0 \\ -10.4615353513333 & 3.74620374537383 & 0 \\ 0 & 0 & -0.250947559611951 \end{bmatrix}$$

[165]:
```
s_new_full = s_new + eye(3) *p
s_new_full
```

[165]:
$$\begin{bmatrix} -3.24430862614993 & -10.4615353513333 & 0 \\ -10.4615353513333 & 3.99715130498578 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

[166]:
```
ss = s_new*s_new.transpose()
mises_eq = sp.sqrt(3/2 * (ss[0,0] + ss[1,1] + ss[2,2]))
```

[167]:
```
ss
```

[167]:
$$\begin{bmatrix} 121.660537711304 & -2.62529676621123 & 0 \\ -2.62529676621123 & 123.47776440905 & 0 \\ 0 & 0 & 0.0629746776751939 \end{bmatrix}$$

[168]:
```
print(f"Mises equivalent stress: {mises_eq:.4f} [MPa]")
```

```
Mises equivalent stress: 19.1782 [MPa]
```