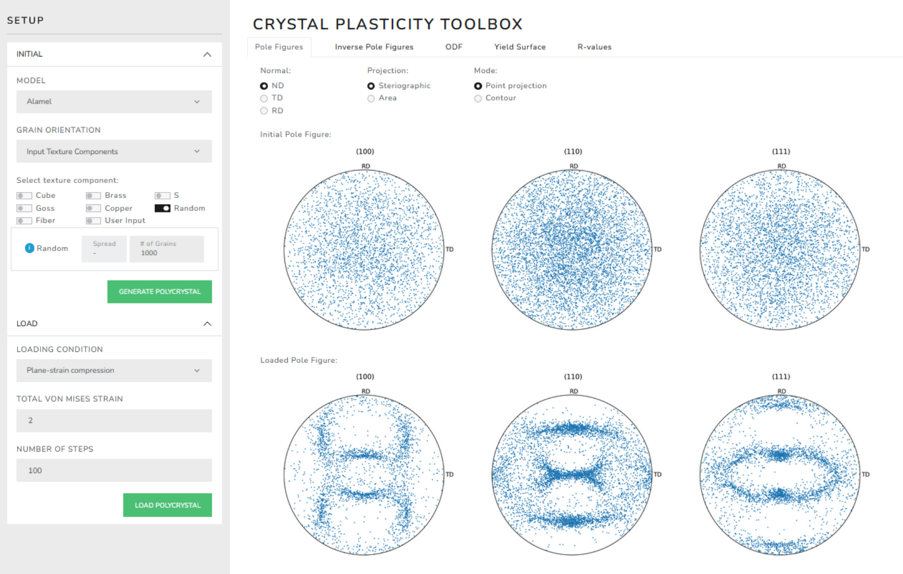
Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Materials Science and Engineering

Dianne Kim Bienes

Cloud-based crystal plasticity web application

Master's thesis in Materials Science and Engineering (Materials for Energy Technology)
Supervisor: Tomáš Mánik
Co-supervisor: Bjørn Holmedal, Hassan Moradi Asadkandi
June 2023


SETUP
INITIAL
MODEL: Alamel
GRAIN ORIENTATION: Input Texture Components
Select texture component:
☐ Cube ☐ Brass ☐ S
☐ Goss ☐ Copper ☑ Random
☐ Fiber ☐ User Input
Random — Spread: - — # of Grains: 1000
LOAD
LOADING CONDITION: Plane-strain compression
TOTAL VON MISES STRAIN: 2
NUMBER OF STEPS: 100

CRYSTAL PLASTICITY TOOLBOX
Normal: ● ND ○ TD ○ RD
Projection: ● Steriographic ○ Area
Mode: ● Point projection ○ Contour

NTNU
Norwegian University of
Science and Technology

Dianne Kim Bienes

Cloud-based crystal plasticity web application

Master's thesis in Materials Science and Engineering (Materials for Energy Technology)
Supervisor: Tomáš Mánik
Co-supervisor: Bjørn Holmedal, Hassan Moradi Asadkandi
June 2023

Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Materials Science and Engineering

NTNU
Norwegian University of
Science and Technology

Abstract

An open-source cloud-based application has been developed to facilitate crystal plasticity simulations using the fully-constrained Taylor model and Alamel model. The application is primarily constructed using Python, leveraging the Dash framework for the creation of the user interface and controller layer. It was deployed to Azure through Docker. While incorporating a few novel Python functions and a newly built user interface, the application also incorporates existing Fortran and Python programs developed by the Department of Materials Science and Engineering at NTNU, enhancing its capabilities and reliability. The crystal plasticity toolbox offers a diverse range of visualizations, including pole figures, inverse pole figures, orientation distribution function (ODF) figures, two-dimensional yield surfaces and R-values figures. These visualizations are essential for comprehending and analyzing the crystallographic texture and plastic deformation behavior of polycrystalline materials. Users can conveniently upload the initial texture of a polycrystalline aggregate from an external file, such as data acquired through Electron back-scattered diffraction (EBSD). Users can also synthesize textures by defining a spread around given orientations as well as fibre textures. Additionally, the toolbox facilitates the examination of a single crystal's orientation, specified by three Euler angles, for educational purposes. An advantageous feature of the toolbox is its scalability, allowing for future expansion to support other mean-field crystal plasticity models.

Preface

This master's thesis is written as part of the requirements for the completion of my Master's Degree in Materials Science and Engineering at the Norwegian University of Science and Technology (NTNU), under the Faculty of Natural Sciences and Department of Material Science and Engineering. This is written during the spring of 2023 and is a continuation of my specialization project which was done in the autumn of 2022.

I would like to extend my gratitude to my main supervisor, Tomáš Mánik, who introduced me to the world of crystal plasticity and provided valuable feedback during the development of the toolbox. I would also like to thank my co-supervisors, Hassan Moradi Asadkandi and Bjørn Holmedal, for the support during the toolbox development.

I had no previous knowledge in crystal plasticity models and I have not developed an application before, but now there exists a crystal plasticity toolbox that I have designed, coded, tested and deployed — from scratch.

I dedicate this to my parents, Dimna and Michael, my brother, Orville, and to my Nanay Payling. They made me who I am today.

And, of course, this is for Jac. Thank you for always making me coffee and ensuring that I see this through until the end.

Trondheim, 12.06.2023
Dianne Kim Bienes

Contents

Abstract . . . iii
Preface . . . v
Contents . . . vii
Figures . . . ix
Tables . . . xi
Code Listings . . . xiii
Acronyms . . . xv
1 Introduction . . . 1
  1.1 Background and Motivation . . . 1
  1.2 Aim and Scope . . . 2
2 Theory . . . 3
  2.1 Single crystal plasticity . . . 3
    2.1.1 Schmid's Law . . . 3
    2.1.2 Kinematics of single crystal plasticity . . . 5
  2.2 Crystal plasticity models for polycrystals . . . 7
    2.2.1 FC-Taylor model . . . 8
    2.2.2 Alamel Model . . . 9
  2.3 Material texture . . . 10
    2.3.1 Pole figure . . . 11
    2.3.2 Inverse pole figure . . . 12
    2.3.3 ODF figure . . . 12
  2.4 Yield surface . . . 13
  2.5 R-values . . . 14
3 Software development . . . 15
  3.1 Requirements and specifications . . . 15
  3.2 Technology and tools . . . 17
    3.2.1 Python . . . 17
    3.2.2 F2PY . . . 17
    3.2.3 Dash . . . 18
    3.2.4 VS Code . . . 21
    3.2.5 Docker . . . 21
    3.2.6 Azure . . . 21
  3.3 Software architecture . . . 22
  3.4 Generation of python extension modules . . . 25

- 3.5 Build and deployment . . . 27
- **4 Results and discussion . . . 35**
  - 4.1 Implementation . . . 35
    - 4.1.1 Generation of initial and loaded polycrystal . . . 35
    - 4.1.2 Generation of pole figures . . . 36
    - 4.1.3 Generation of inverse pole figures . . . 38
    - 4.1.4 Generation of ODF figures . . . 39
    - 4.1.5 Generation of yield surface . . . 40
    - 4.1.6 Generation of R-values . . . 40
  - 4.2 Crystal plasticity toolbox overview . . . 43
  - 4.3 Generated figures . . . 47
  - 4.4 Challenges during implementation . . . 58
- **5 Conclusion . . . 61**
  - 5.1 Conclusion . . . 61
  - 5.2 Future work . . . 61
- **References . . . 63**
- **A Conversations with an AI (ChatGPT) . . . 69**
- **B Code listings . . . 71**
  - B.1 Pre-requisite Installation . . . 71
  - B.2 Layout Codes . . . 72
  - B.3 Callback Codes . . . 93
  - B.4 Model Component Codes . . . 112
  - B.5 Main Dash Application . . . 123
  - B.6 Added functions in `crystal_plasticity_module.py` . . . 123
  - B.7 Added function in `odflib.py` . . . 130
  - B.8 Updated function in `TexTool_Driver.py` . . . 130
  - B.9 Updated function in `TexTool_Funcs.py` . . . 131

Figures

1.1 A screenshot from the implementation in Jupyter Notebook. 2

2.1 Illustration of the geometry of slip. 4
2.2 FCC slip planes and slip directions (slip systems). 5
2.3 Decomposition of deformation gradient. 5
2.4 Sample illustrations of crystal plasticity models. 7
2.5 Schematic of a sample global fixed coordinate system, C_G, and crystal coordinate system, C_L. 10
2.6 Euler angle rotation. 11
2.7 Pole figure superimposed to a standard stereographic projection and corresponding inverse pole Ffigure example. 12
2.8 ODF of 95% cold-rolled copper with constant φ_2). 13

3.1 An example of a basic user interface for the `State` implementation in a callback function. 20
3.2 Generic data flow diagram for a MVC architecture. 22
3.3 High-level layout of application. 23
3.4 Application file tree structure. 24
3.5 Deployment flow for local development and production environments. 28
3.6 Docker image in Docker desktop application. 29
3.7 Docker container in Docker desktop application. 29
3.8 Crystal plasticity toolbox application running in the local environment through Docker. 30
3.9 Creation of new container registry in Azure. 31
3.10 Docker image pushed to Azure container registry. 32
3.11 Creation of new web application in Azure. 33
3.12 Crystal plasticity toolbox application running in cloud. 33

4.1 Initial load view of the cloud-based crystal plasticity toolbox. 43
4.2 Pole figure tab of the cloud-based crystal plasticity toolbox. 44
4.3 Inverse pole figure tab of the cloud-based crystal plasticity toolbox. 45
4.4 ODF figure tab of the cloud-based crystal plasticity toolbox. 46
4.6 R-values figure tab of the cloud-based crystal plasticity toolbox. . . 46

4.5 Yield surface figure tab of the cloud-based crystal plasticity toolbox. 47
4.7 Pole figures generated using stereographic point projection for FCC structure with 1000 random grains subjected to axisymmetric tension along z-axis using Taylor Model. 48
4.8 Pole figures generated using stereographic projection in contour plots for FCC structure with 1000 random grains subjected to axisymmetric tension along z-axis using Taylor Model. 49
4.9 Pole figures generated using area point projection for FCC structure with 1000 random grains subjected to plane-strain compression using Alamel Model. 50
4.10 Pole figures generated using area projection in contour plots for FCC structure with 1000 random grains subjected to plane-strain compression using Alamel Model. 51
4.11 Inverse pole figures generated for FCC structure with 1000 random grains subjected to axisymmetric tension along z-axis using Taylor Model. 52
4.12 Inverse pole figures generated for FCC structure with 1000 random grains subjected to plane-strain compression using Alamel Model. . 53
4.13 ODF figures generated for FCC structure with 1000 random grains subjected to axisymmetric tension along z-axis using Taylor Model. 54
4.14 ODF figures generated for FCC structure with 1000 random grains subjected to plane-strain compression using Alamel Model. 55
4.15 Two-dimension yield surfaces generated for FCC structure with 1000 random grains subjected to plane-strain compression using Alamel Model. 56
4.16 R-value and yield stress figures generated for FCC structure with 1000 random grains subjected to plane-strain compression using Alamel Model. 57

A.1 ChatGPT: Steps how to deploy a Dash application with .so python modules to Azure with Docker. 69
A.2 ChatGPT: Steps how to install libgfortran5 using Dockerfile. 70
A.3 ChatGPT: Steps to install Intel® oneAPI Math Kernel Library. 70

Tables

3.1 Crystal plasticity toolbox functional requirements 15
3.2 Crystal plasticity toolbox non-functional requirements 17

Code Listings

3.1 Example code for layout and basic callback function 20
3.2 Example code for the `State` implementation in a callback function for Figure 3.1 20

B.1 requirements.txt 71
B.2 Dockerfile.prod 71
B.3 Code for sidebar layout 72
B.4 Code for content layout 85
B.5 Code for common layout 92
B.6 Code for sidebar callbacks 93
B.7 Code for content callbacks 100
B.8 datasource.py 112
B.9 uploadfile.py 122
B.10 app.py 123
B.11 plot_orientations_plotly function 123
B.12 plot_orientation_plotly function 124
B.13 save_plotly function 130
B.14 Plot_PF function 130
B.15 Projection function 131

Acronyms

AI Artificial Intelligence. 59

BCC Body-centered cubic. 5

CPFEM Crystal plasticity finite element method. 1, 8

CPFFT Crystal plasticity fast Fourier transform-based or Crystal plasticity spectral methods. 8

CRSS Critical resolved shear stress. 3, 4, 14

CSS Cascading style sheets. 19, 24

EBSD Electron back-scattered diffraction. iii, 61

F2PY Fortran to Python interface generator. vii, 17, 18, 25, 26, 35, 58, 61

FC Full-constrained. vii, 2, 8, 9, 43

FCC Face-centered cubic. 4, 8, 16, 36, 44, 47

GUI Graphical User Interface. 2, 17–19, 23, 24, 35, 39, 40, 43, 45, 61

HTML Hypertext markup language. 19, 37, 39

LAPACK Linear Algebra PACKage. 58

MIT Massachusetts Institute of Technology. 19

MVC Model-Controller-View. ix, 22, 23

ND Normal Direction. 37, 44

NTNU Norwegian University of Science and Technology. 1, 25

NumPy Numerical Python. 18, 26, 36

ODF Orientation distribution function. iii, viii–x, 2, 12, 13, 16, 35, 39, 44–47, 54, 55, 61

oneMKL Intel® oneAPI Math Kernel Library. 26, 58, 59

RD Rolling Direction. 37, 44

RGC Relaxed grain cluster. 1, 7

TD Transverse Direction. 37, 44

URL Uniform Resource Locators. 32, 33

VPSC Visco-plastic self-consistent. 1

VS Code Visual Studio Code. 21, 27–29, 31

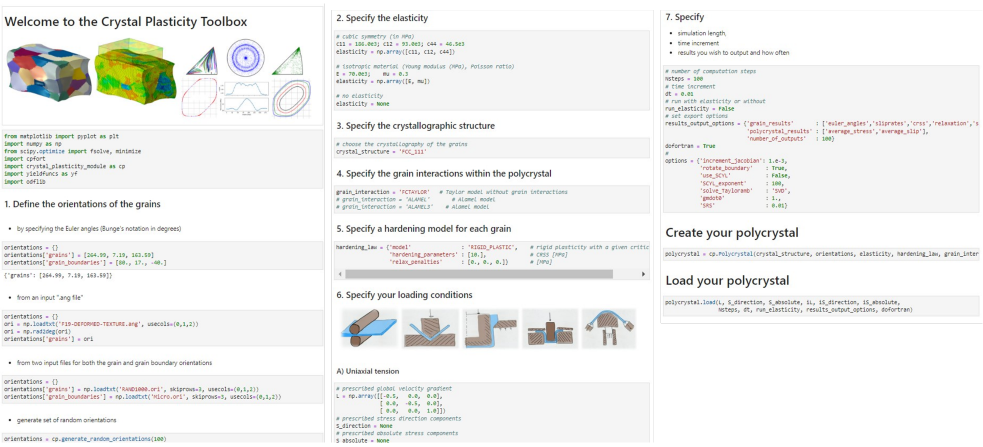
Chapter 1

Introduction

1.1 Background and Motivation

Understanding material deformation, when subjected to external forces, is important for predicting mechanical properties, which enables more informed decisions on how to improve material performance, and ensures the reliability and safety of engineered systems. Plastic deformation, which is irreversible or permanent, is a key factor in determining the causes of material failure. On a macroscopic level, the plastic deformation is often modelled as isotropic, using simple and computationally fast Von Mises plasticity model. However, the origin of plasticity on the microscopic scale is by nature strongly anisotropic. To understand the mechanisms of plastic deformation for crystalline materials, such as metals, crystal plasticity theory has been developed. This theory assumes that, at ambient temperatures, twinning or dislocation movement along specific planes and directions are the main causes for plastic deformation in crystalline materials [1, 2]. Previous researchers have already studied and created various crystal plasticity models (e.g. Sachs model [3], Taylor model [4], Alamel model [5] , Visco-plastic self-consistent (VPSC) model [6], Relaxed grain cluster (RGC) model [7], Crystal plasticity finite element method (CPFEM) [8] and crystal plasticity fast Fourier transform-based or crystal plasticity spectral methods (CPFFT) [9]) to help predict material texture and mechanical properties.

At the Department of Materials Science and Engineering in the Norwegian University of Science and Technology (NTNU), various microstructure and physics-based models have been developed during the past two decades. These have been implemented in both Fortran and Python, which are related to material texture and mean-field crystal plasticity models, such as the Taylor and Alamel models, along with more advanced CPFEM codes. Presently, the Taylor and Alamel models are visualized using Jupyter Notebook, as shown in Figure 1.1. The objective of this project is to create a contemporary web application that encompasses these existing programs, making them readily accessible to users.



Welcome to the Crystal Plasticity Toolbox

```
from matplotlib import pyplot as plt
import numpy as np
from scipy.optimize import fsolve, minimize
import cpfort
import crystal_plasticity_module as cp
import yieldfuncs as yf
import odflib
```

1. Define the orientations of the grains

- by specifying the Euler angles (Bunge's notation in degrees)

```
orientations = {}
orientations['grains'] = [264.99, 7.19, 163.59]
orientations['grain_boundaries'] = [80., 17., -40.]
```

```
{'grains': [264.99, 7.19, 163.59]}
```

- from an input ".ang file"

```
orientations = {}
ori = np.loadtxt('F19-DEFORMED-TEXTURE.ang', usecols=(0,1,2))
ori = np.rad2deg(ori)
orientations['grains'] = ori
```

- from two input files for both the grain and grain boundary orientations

```
orientations = {}
orientations['grains'] = np.loadtxt('RAND1000.ori', skiprows=3, usecols=(0,1,2))
orientations['grain_boundaries'] = np.loadtxt('Micro.ori', skiprows=3, usecols=(0,1,2))
```

- generate set of random orientations

```
orientations = cp.generate_random_orientations(100)
```

2. Specify the elasticity

```
# cubic symmetry (in MPa)
c11 = 186.0e3; c12 = 93.0e3; c44 = 46.5e3
elasticity = np.array([c11, c12, c44])

# isotropic material (Young modulus (MPa), Poisson ratio)
E = 70.0e3;    mu = 0.3
elasticity = np.array([E, mu])

# no elasticity
elasticity = None
```

3. Specify the crystallographic structure

```
# choose the crystallography of the grains
crystal_structure = 'FCC_111'
```

4. Specify the grain interactions within the polycrystal

```
grain_interaction = 'FCTAYLOR'   # Taylor model without grain interactions
# grain_interaction = 'ALAMEL'       # Alamel model
# grain_interaction = 'ALAMEL3'    # Alamel model
```

5. Specify a hardening model for each grain

```
hardening_law = {'model'                 : 'RIGID_PLASTIC',    # rigid plasticity with a given critic
                 'hardening_parameters' : [10.],               # CRSS [MPa]
                 'relax_penalties'      : [0., 0., 0.]}        # [MPa]
```

6. Specify your loading conditions

A) Uniaxial tension

```
# prescribed global velocity gradient
L = np.array([[-0.5,   0.0,  0.0],
              [ 0.0,  -0.5,  0.0],
              [ 0.0,   0.0,  1.0]])
# prescribed stress direction components
S_direction = None
# prescribed absolute stress components
S_absolute = None
```

7. Specify

- simulation length,
- time increment
- results you wish to output and how often

```
# number of computation steps
Nsteps = 100
# time increment
dt = 0.01
# run with elasticity or without
run_elasticity = False
# set export options
results_output_options = {'grain_results'       : ['euler_angles','sliprates','crss','relaxation','s
                          'polycrystal_results' : ['average_stress','average_slip'],
                          'number_of_outputs'   : 100}
dofortran = True
#
options = {'increment_jacobian': 1.e-3,
           'rotate_boundary'    : True,
           'use_SCYL'           : False,
           'SCYL_exponent'      : 100,
           'solve_Tayloramb'    : 'SVD',
           'gmdot0'             : 1.,
           'SRS'                : 0.01}
```

Create your polycrystal

```
polycrystal = cp.Polycrystal(crystal_structure, orientations, elasticity, hardening_law, grain_inter
```

Load your polycrystal

```
polycrystal.load(L, S_direction, S_absolute, iL, iS_direction, iS_absolute,
                 Nsteps, dt, run_elasticity, results_output_options, dofortran)
```

Figure 1.1: A screenshot from the implementation in Jupyter Notebook.

1.2 Aim and Scope

The aim of the master's thesis is to develop a web-based application with a user-friendly graphicsl user interface (GUI), that would serve as a platform to visualize material texture and to perform crystal plasticity calculations by means of full-constrained (FC) Taylor and two-grain Alamel models. The application will allow for quick visualization and evaluation of pole figures, inverse pole figures, orientation distribution function (ODF) figures, two-dimension yield surfaces, and R-values.

The scope and requirements of the project will be thoroughly discussed in Chapter 3, however, at a high-level, the following points outline the key aspects covered:

- Generate initial and loaded figures for pole figures, inverse pole figures, ODF figures, two-dimension yield Surfaces, and R-values
- Input the following parameters in the GUI for the simulation: (1) grain orientation, (2) loading parameters, (3) total Von Mises strain, and (4) number of steps

Chapter 2

Theory

This chapter gives a brief introduction to the theory behind the toolbox. Since the main focus of the master's thesis is on the development of the crystal plasticity toolbox, this chapter only gives a high-level discussion on single crystal plasticity, polycrystal plasticity models, material texture, yield surface and R-values.

2.1 Single crystal plasticity

2.1.1 Schmid's Law

Plastic deformation for single crystals is facilitated by the movement of dislocations along specific slip systems. These slip systems consist of close-packed slip directions and typically close-packed slip planes. This arrangement enables dislocations to move more easily since it produces the least Burger's vector magnitude [10], which is a measure of the lattice distortion caused by the dislocation. As described by Schmid [1], crystallographic slip takes place on a slip plane along a specific slip direction when the resolved shear stress reaches the critical resolved shear stress (CRSS) value.

Schmid's Law can be visually represented using Figure 2.1, which depicts a single crystal under uniaxial tension $\boldsymbol{\sigma}$. In the figure, the slip direction $\mathbf{s}$ is on the slip plane which has the normal direction $\mathbf{n}$. Additionally, the angles ϕ and λ are shown, representing the angles between the loading axis and the slip normal plane, and between the loading axis and the slip plane, respectively. The resolved shear stress component can be calculated using the Equation 2.1, where the term $\cos\phi\ \cos\lambda$ is known as the Schmid factor, which represents the projection of the applied stress onto the slip system and determines the tendency for plastic deformation along that particular slip system.

$$\tau = \mathbf{n} \cdot \boldsymbol{\sigma} \cdot \mathbf{s} = \sigma\ \cos\phi\ \cos\lambda \tag{2.1}$$

The resolved shear stress τ on a slip system α can be defined as the component of the Cauchy stress tensor that acts along the slip direction of the slip system. The

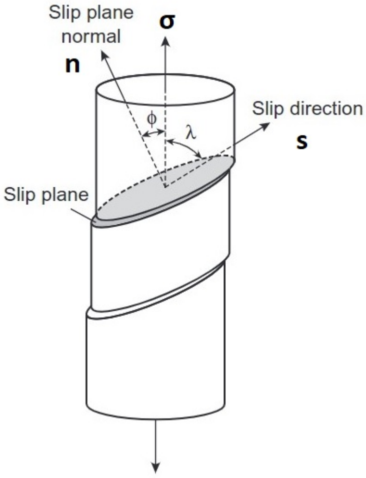


Figure 2.1: Illustration of the geometry of slip [11]. (*Figure from Introduction to Dislocations, Fifth Edition, D. Hull,D.J. Bacon, "Chapter 3 - movement of dislocations", Pp. 43–62, 2011, with permission from Elsevier*)

resolved shear stress is responsible for driving the motion of dislocations and initiating plastic deformation and can be defined by Equation 2.2, where the Schmid tensor is $\boldsymbol{M}^{\alpha} = \boldsymbol{s}^{\alpha} \otimes \boldsymbol{n}^{\alpha}$. Here, α represents an identifier for a slip system.

$$|\tau^{\alpha}| = |\boldsymbol{\sigma} : \tfrac{1}{2}(\boldsymbol{M}^{\alpha} + \boldsymbol{M}^{\alpha T})| = |\boldsymbol{\sigma} : \tfrac{1}{2}(\boldsymbol{s}^{\alpha} \otimes \boldsymbol{n}^{\alpha} + \boldsymbol{n}^{\alpha} \otimes \boldsymbol{s}^{\alpha})| \leq \tau_c^{\alpha} \tag{2.2}$$

Each slip system within a crystal can have a different orientation relative to the loading axis. Consequently, each slip system will exhibit different values for the resolved shear stress τ^{α}. Among these various slip systems, the one with the highest τ^{α} value which also reached the CRSS, τ_c, is where dislocation motion occurs. Usually τ_c increases with the plastic deformation of the crystal, i.e. the crystal hardens. Equation 2.3 demonstrates that the uniaxial stress will have different values depending on the crystal orientation. This leads to anisotropic plastic deformation at the microscale level, where the material's response to deformation varies depending on crystal orientation [1]. Hence, the tensile stress σ_c can be expressed as Equation 2.3.

$$\sigma_c = \frac{\tau_c}{\cos\phi\ \cos\lambda} \tag{2.3}$$

Crystal structures consist of specific slip systems and in face-centered cubic (FCC) crystal structures, 12 slip systems are typically employed to accommodate plastic deformation. These slip systems result from the combination of four {111} close-packed slip planes, each of which has three <110> slip directions. Figure 2.2 provides a visualization of these slip systems. In contrast, for body-centered cubic

(BCC) crystal structures, the slip systems are found on planes {110}, {112}, and {123}, wherein the slip direction for all these planes is consistently along <111> [12].

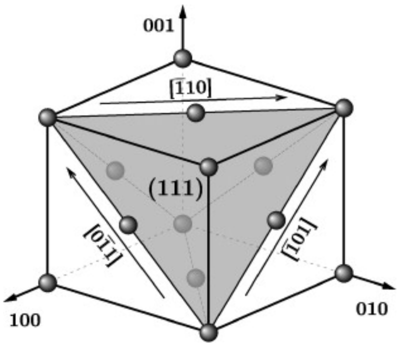


Figure 2.2: FCC slip planes and slip directions (slip systems) [13]. (*Figure from Computational Materials Science, Vol. 70, Y. Guilhem, S. Basseville, F. Curtit, J.-M. Stéphan and G. Cailletaud, "Numerical investigations of the free surface effect in three-dimensional poly-crystalline aggregates", Pp. 150–162, 2013, with permission from Elsevier*)

2.1.2 Kinematics of single crystal plasticity

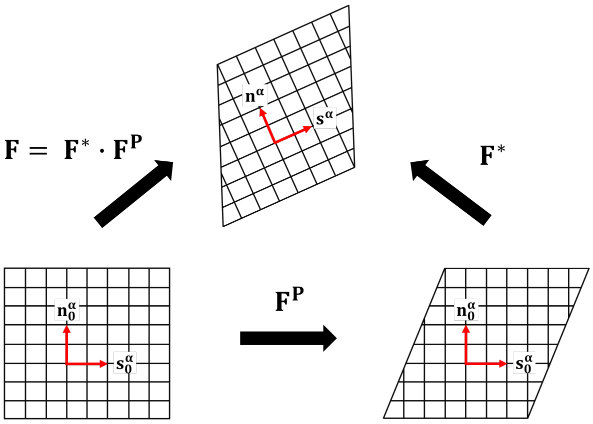


Figure 2.3: Decomposition of deformation gradient [14].

The deformation of a single crystal can be described by decomposing its continuous deformation gradient $\mathbf{F}$ into its plastic part $\mathbf{F}^\mathbf{P}$ and elastic part $\mathbf{F}^*$, which also accounts for rigid body rotation. Figure 2.3 schematically represents this decomposition. An initial reference system is depicted for the single crystal, as can be seen in the lower left part of Figure 2.3, with $\mathbf{n}_0^\alpha$ representing the initial unit slip plane normal vector and $\mathbf{s}_0^\alpha$ representing the initial unit slip direction for the α slip system. When the crystal lattice undergoes a continuous deformation $\mathbf{F}$, it experiences plastic deformation through the plastic deformation gradient $\mathbf{F}^\mathbf{P}$, resulting in shearing caused by dislocation slip. This leads to an intermediate configuration, as shown in the lower right of Figure 2.3. Since this intermediate configuration is assumed to be stress-free, the crystal lattice and orientation from the initial configuration is preserved. The elastic deformation includes also rigid body rotations and the elastic deformation gradient $\mathbf{F}^*$ transforms vector $\mathbf{s}_0^\alpha$ into $\mathbf{s}^\alpha$ and vector $\mathbf{n}_0^\alpha$ into $\mathbf{n}^\alpha$. As a result, it produces the deformed and stressed configuration of the crystal lattice, which is represented at the top part of Figure 2.3. The mathematical representation of the deformation gradient decomposition can be expressed as shown in Equation 2.4 [1, 10].

$$\mathbf{F} = \mathbf{F}^* \cdot \mathbf{F}^p \tag{2.4}$$

The total deformation can also be described using the velocity gradient $\mathbf{L}$, which can be further divided into its plastic component $\mathbf{L}^p$ and elastic component $\mathbf{L}^*$. This can be mathematically represented by Equation 2.5.

$$\mathbf{L} = \dot{\mathbf{F}} \cdot \mathbf{F}^{-1} = \mathbf{L}^* + \mathbf{L}^p = \dot{\mathbf{F}}^* \cdot (\mathbf{F}^*)^{-1} + \mathbf{F}^* \cdot \dot{\mathbf{F}}^p \cdot (\mathbf{F}^p)^{-1} \cdot (\mathbf{F}^*)^{-1} \tag{2.5}$$

Furthermore, the velocity gradient $\mathbf{L}$ can also be decomposed into two parts, the deformation velocity part $\mathbf{D}$ and the spin velocity part $\mathbf{W}$, as can be seen in Equation 2.6. Additionally, these parts can be further decomposed into their respective plastic and elastic components, as can be seen in Equation 2.7. Plastic deformation rate $\mathbf{D}^p$ and plastic spin $\mathbf{W}^p$ occur due to dislocation slip within the crystal lattice, while elastic deformation rate $\mathbf{D}^*$ and elastic spin $\mathbf{W}^*$ result from the stretching and rigid rotation of the crystal lattice.

$$\mathbf{L} = \mathbf{D} + \mathbf{W} \tag{2.6}$$

$$\mathbf{D} = \mathbf{D}^* + \mathbf{D}^p, \ \mathbf{W} = \mathbf{W}^* + \mathbf{W}^p \tag{2.7}$$

The Schmid tensor $\mathbf{M}^\alpha$ for a specific slip system α is represented as the tensor product of the slip direction $\mathbf{s}^\alpha$ and the slip plane normal $\mathbf{n}^\alpha$, as shown in Equation 2.8. This Schmid factor can be further separated into a symmetric component $\mathbf{P}^\alpha$ and $\mathbf{\Omega}^\alpha$, as shown in Equations 2.9 and 2.10.

$$\mathbf{M}^\alpha = \mathbf{s}^\alpha \otimes \mathbf{n}^\alpha = \mathbf{P}^\alpha + \mathbf{\Omega}^\alpha \tag{2.8}$$

$$\mathbf{P}^\alpha = \frac{\mathbf{s}^\alpha \otimes \mathbf{n}^\alpha + \mathbf{n}^\alpha \otimes \mathbf{s}^\alpha}{2} \tag{2.9}$$

$$\mathbf{\Omega}^\alpha = \frac{\mathbf{s}^\alpha \otimes \mathbf{n}^\alpha - \mathbf{n}^\alpha \otimes \mathbf{s}^\alpha}{2} \tag{2.10}$$

The plastic velocity gradient $\mathbf{L}^p$ can be expressed as the summation of the shearing occurring in the crystal lattice, since plastic deformation is a result of shearing within the slip system. In Equation 2.11, $\dot{\gamma}^\alpha$ is the shear rate of α slip system.

$$\mathbf{L}^p = \sum_{\alpha=1}^{n} \dot{\gamma}^\alpha (\mathbf{s}^\alpha \otimes \mathbf{n}^\alpha) = \sum_{\alpha=1}^{n} \dot{\gamma}^\alpha (\mathbf{P}^\alpha + \mathbf{\Omega}^\alpha) \qquad (2.11)$$

2.2 Crystal plasticity models for polycrystals

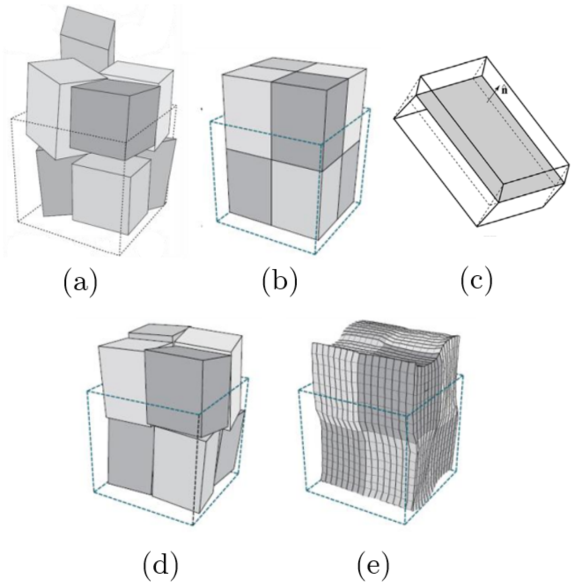


Figure 2.4: Sample illustrations of crystal plasticity models: (a) Sachs model [15], (b) Taylor model [15], (c) Alamel model [16], (d) Relaxed grain cluster (RGC) model [15], and (e) Full-field models such as Crystal plasticity finite element method (CPFEM) [15].

Homogenization methods or models are utilized to predict the crystal plasticity behavior in polycrystalline materials. These models can be classified into two categories: mean-field models and full-field models [14].

Mean-field models are statistical models that represent a polycrystal by volume fractions of crystal orientations. These models do not explicitly consider the interaction between grains, but they incorporate some level of local interaction between neighboring grains or small grain clusters. Mean-field models provide qualitative accuracy and fast simulations, however, they lack spatial resolution compared to full-field models. Some examples of these models are the Sachs model [3], Taylor model [4], Alamel model [5], and RGC model [7], which can be visualized in Figure 2.4a-d. These models assume homogeneous stress or strain to simplify the rather complex mechanisms in a polycrystal.

Full-field models consider both short-range and long-range grain interaction, which can be visualized in Figure 2.4e. Grain morphology modelling in full-field models mimic real microstructures during simulation, which makes models under this category more accurate but considerably higher in computational costs. Examples of models under this category are the CPFEM [8] and CPFFT [9].

The crystal plasticity toolbox supports the Taylor and Alamel models, thus a high-level discussion on the theories behind these models will be included in the succeeding sub-sections.

2.2.1 FC-Taylor model

The FC-Taylor model [4] was one of the first models, developed in 1938, which tried to understand and predict how the plastic behavior of individual crystals can influence the overall mechanical response and texture development in a polycrystal. This model provides a framework to calculate the plastic anisotropy of a textured polycrystal or predict the crystallographic textures that result from significant plastic deformations in single-phased polycrystals.

Taylor assumes that each grain in a polycrystal undergoes uniform strain. The strain experienced by each grain is equal to the global strain, thus each grain experiences the same velocity gradient, and that the grains also do not interact with each other. This assumption simplifies the analysis by allowing the macroscopic deformation of the polycrystal to be equated to the deformation of a single crystal with a known crystallographic orientation [17].

In the context of metals, the incompressibility condition, wherein the volume remains constant, implies that the plastic deformation rate tensor $\mathbf{D}^p$ only has five independent components. Thus, plastic deformation rate $\mathbf{D}^p$ and plastic spin $\mathbf{W}^p$ can be represented as shown in Equations 2.12 and 2.13, respectively, with five independent slip systems [18].

$$\mathbf{D}^p = \sum_{\alpha=1}^{n} \dot{\gamma}^\alpha \mathbf{P}^\alpha = \sum_{\alpha} \frac{1}{2}(\mathbf{M}^\alpha + \mathbf{M}^{\alpha T})\dot{\gamma}^\alpha \tag{2.12}$$

$$\mathbf{W}^p = \sum_{\alpha=1}^{n} \dot{\gamma}^\alpha \mathbf{\Omega}^\alpha = \sum_{\alpha} \frac{1}{2}(\mathbf{M}^\alpha + \mathbf{M}^{\alpha T})\dot{\gamma}^\alpha \tag{2.13}$$

However, there is no unique solution to these five equations from Equation 2.12, since in reality most metals have more than five possible slip systems. For example, FCC metals typically have twelve potential slip systems, and in the case of five active slip systems out of the twelve, there would be 792 possible combinations. To address this issue, Taylor introduced an assumption that the solutions with the least plastic work dissipation in the slip systems, shown in Equation 2.14, were considered acceptable solutions. It should be noted that even with this criterion, typically no unique solution exists, i.e. multiple solutions minimize the plastic work rate. This is called the Taylor ambiguity. To determine a unique solution, an additional condition need to be specified or the rate-dependency introduced [17].

$$\sum_{\alpha} |\dot{\gamma}^\alpha| \tau_c^\alpha = min \tag{2.14}$$

In response to the challenges of identifying the active slip systems, Bishop and Hill [19, 20] proposed an alternative theory that builds upon Taylor's work. Instead of focusing on the solution that minimizes the internal plastic work, their approach utilizes a stress-based procedure for the iso-strain scenario. Their approach aimed to directly determine stress states that could simultaneously activate at least five independent slip systems. By considering yield criterion and maximizing the external work, shown in Equation 2.15, the permissible stress states can be narrowed down to one that maximizes the external work [17].

$$\dot{W} = \sigma_{ij} D^{p}_{ij} = max \tag{2.15}$$

Bishop and Hill [19, 20] showed that the principle of maximum work is equivalent to Taylor's approach of minimizing internal energy dissipation [17]. Thus, the model that combines Taylor's and Bishop-Hill's methods is often referred to as the Taylor-Bishop-Hill model or the FC-Taylor model.

2.2.2 Alamel Model

The Taylor model is extremely restricting since it assumes iso-strain inside each grain and interaction between grains is not considered. To relax the iso-strain assumption, models which consider grain clusters were eventually studied and developed. The Alamel model [5], introduced by Van Houtte et al., is one example for this, wherein the polycrystal is represented by grain pairs, with orientations that as a whole represent a given texture. This model considers the local interactions between the grain pair by allowing relaxation of constraints on the shear components of the prescribed deformation. The yield criteria of the slip systems and the equilibrium conditions of the shear stress components are considered when calculating the stress tensors in each crystal of one grain pair [5, 21].

In the Alamel model, a grain pair consists of a "stack" of two grains, a and b, with a common grain boundary which can be arbitrarily inclined. The grain pair is exposed to a prescribed velocity gradient $\boldsymbol{L}$ with spin velocity $\mathbf{W}_L$. The shear rate is represented as $\dot{\gamma}^{\alpha}$ and the unit vector along the slip direction of system is represented as $\hat{\boldsymbol{s}}^{\alpha}$ of α slip system. Grains a and b has an interface with a normal vector $\boldsymbol{n}$, and they have local velocity gradients, $\boldsymbol{l}^a$ and $\boldsymbol{l}^b$, which could be different from the global velocity gradient $\boldsymbol{L}$. The difference of the local and global velocity gradients is due to allowed relaxation, $\boldsymbol{K}^r$, on the shear components [5, 21].

$$L = \boldsymbol{W}_L = + \sum_{\alpha} \dot{\gamma}^{\alpha} \hat{\boldsymbol{s}}^{\alpha} \otimes \boldsymbol{n}^{\alpha} \tag{2.16}$$

$$\boldsymbol{l}^a = \boldsymbol{L} + \sum_{r} \boldsymbol{K}^r \dot{\gamma}^r \tag{2.17}$$

$$\boldsymbol{l}^b = \boldsymbol{L} - \sum_{r} \boldsymbol{K}^r \dot{\gamma}^r \tag{2.18}$$

The Alamel model incorporates symmetric relaxation of the two local velocity gradient components such that $\sum_r \boldsymbol{K}^r = \boldsymbol{e} \otimes \boldsymbol{n}$ where $\boldsymbol{e} \perp \boldsymbol{n}$, which ensures constant stress equilibrium at the boundary except for the normal component [5, 21].

2.3 Material texture

Material texture refers to the statistical distribution of crystallographic orientations within a polycrystal. When a polycrystal undergoes plastic deformation, the individual crystals can rotate and align themselves into preferred orientations, resulting in the development of crystallographic texture. This texture can significantly influence various material properties. The introduction of texture introduces anisotropy, meaning that the material properties become direction-dependent. Therefore, it is crucial to have control over the texture formation process in order to achieve desired material properties tailored for specific applications [22].

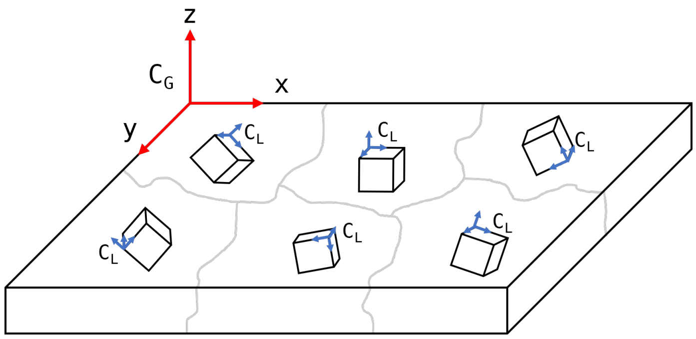


Figure 2.5: Schematic of a sample global fixed coordinate system, C_G, and crystal coordinate system, C_L [23].

To define the orientation of a crystal, we establish a fixed global coordinate system C_G for the polycrystal, as shown in Figure 2.5. Additionally, for each crystal within the polycrystal, a fixed local coordinate system C_L is defined. In this local coordinate system, each direction represents a crystallographic direction, and this definition should remain consistent for all crystals. The orientation of a crystal relative to the polycrystal can be defined by the rotation g (see Equation 2.19) that transforms the global coordinate system C_G to the local coordinate system C_L. This rotation can be described by three angles: φ_1, Φ, and φ_2. These angles are known as Euler angles and they specify the crystal's orientation within the polycrystal [23].

$$g = \{\varphi_1, \Phi, \varphi_2\} \tag{2.19}$$

Figure 2.6 illustrates the process of rotations using Bunge's notation and Euler angles. In this notation, the coordinate axes for the global coordinate system C_G are denoted as X, Y and Z, while the coordinate axes for the local coordinate system C_L are denoted as X', Y' and Z'. Initially, the crystal's axes align with the axes of C_G. To determine the crystal's orientation relative to the polycrystal, the following steps are performed, which results in the final orientation of the crystal's axes represented by X', Y' and Z' in the local coordinate system C_L [23]:

1. Rotate the crystal about the Z-axis by an angle φ_1.
2. Rotate the crystal about the X-axis (using the new axes obtained from the first rotation) by an angle Φ.
3. Finally, rotate the crystal again about the Z-axis (using the new axes obtained from the second rotation) by an angle φ_2.

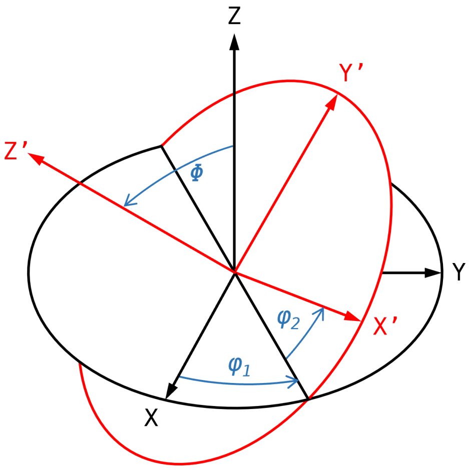


Figure 2.6: Euler angle rotation [23].

2.3.1 Pole figure

A common method to represent and visualize material texture is through the use of pole figures. Pole figures provide a two-dimensional stereographic projection of the orientation distribution of specific crystallographic planes within a material, with respect to a chosen frame of reference. In a pole figure, each point corresponds to a specific crystal orientation. By examining the distribution of points on a pole figure, one can determine if there is clustering or preferred orientation of crystals within the polycrystal, indicating the presence of texture. Clusters of points on the pole figure suggest that certain crystallographic planes are more likely to be aligned in a particular direction. To analyze and compare pole figures, standard geographic projection techniques can be employed. These techniques help identify patterns and trends in the texture of the material. Figure 2.7a illustrates an example of a pole figure for a standard stereographic projection [22].

2.3.2 Inverse pole figure

In addition to pole figures, inverse pole figures can also be utilized to represent material texture. Inverse pole figures, as shown in Figure 2.7b, are named "inverse" because they are projections relative to the crystal axes rather than the sample axes. For materials with a cubic crystal structure, there are 24 inverse pole figure triangles in the stereographic projection. However, due to symmetry, it is sufficient to represent the inverse pole figure with just one triangle. This representation captures the distribution of crystallographic orientations and reveals the preferred orientations or texture in the material. By examining the inverse pole figure, one can gain insights into the orientation distribution and anisotropy of the material. It provides a useful tool for analyzing and understanding the texture characteristics of materials with cubic crystal structures [22].

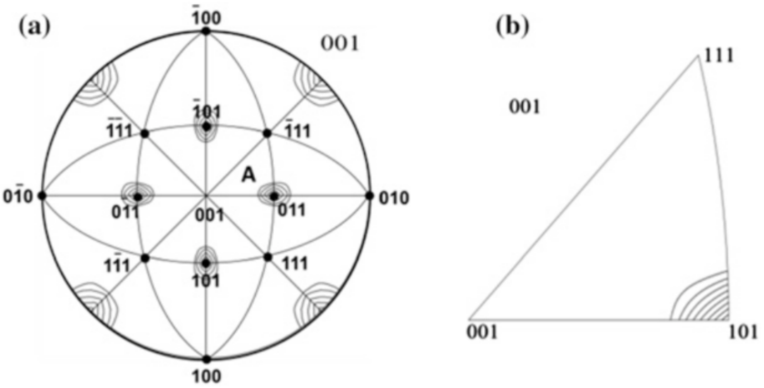


Figure 2.7: (a) Pole figure superimposed to a standard stereographic projection, (b) Inverse Pole Figure example [22]. (*Figure from Crystallographic Texture of Materials, S. Suwas and R. K. Ray, "Representation of texture", Pp. 11–38, 2014, with permission from Springer Nature*)

2.3.3 ODF figure

The ODF is a representation of the continuous distribution of crystal orientations within a polycrystal. It provides information about the frequency of occurrence for different crystal orientations in a three-dimensional Euler space, as depicted in Figure 2.8. The Euler space is defined by the three Euler angles discussed earlier in this section. By analyzing the ODF, one can gain insights into the preferred orientations and anisotropic behavior of the material at a macroscopic level.

The ODF, which can be denoted as $f(\varphi_1, \Phi, \varphi_2)$, expresses the probability density of finding a crystal with a specific orientation defined by the Euler angles φ_1, Φ and φ_2. In other words, it quantifies the likelihood of a crystal having a particular orientation within the polycrystal. Mathematically, the ODF can be represented as Equation 2.20. Here, g represents a specific crystal orientation, $\mathrm{d}g$ is the differential measure of the spread of orientations contained in a volume ΔV, and V is the total volume of the polycrystal. The equation describes the relative distribution of crystal orientations within the polycrystal.

$$f(g)\mathrm{d}g = \frac{\Delta V}{V} \tag{2.20}$$

According to Bunge [23], the ODF can be expressed as Equation 2.21. Here, $\boldsymbol{C}_l^{mn}$ represents the series coefficients, and $\boldsymbol{P}_l^{mn}(\boldsymbol{\Phi})\boldsymbol{e}^{-im\varphi_2}\boldsymbol{e}^{-in\varphi_1}$ is a generalization associated with Legendre functions.

$$f(\varphi_1, \Phi, \varphi_2) = \sum_{l=0}^{\infty} \sum_{m=-l}^{+l} \sum_{n=-l}^{+l} \boldsymbol{C}_l^{mn} \boldsymbol{P}_l^{mn}(\boldsymbol{\Phi}) \boldsymbol{e}^{-im\varphi_2} \boldsymbol{e}^{-in\varphi_1} \tag{2.21}$$

Pole figures, on the other hand, are obtained by integrating the ODF over the angle φ_2. This means that it is possible to derive the ODF from pole figure data.

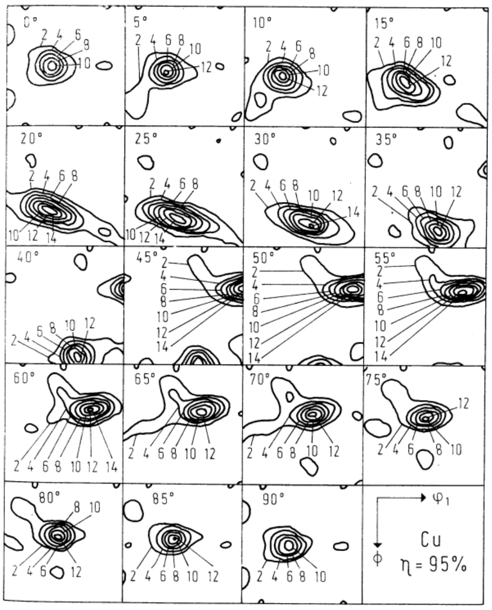


Figure 2.8: ODF of 95% cold-rolled copper with constant φ_2 [23]. (*Figure from Texture Analysis in Materials Science, H.-J.Bunge, "11 - Some Results of Texture Analysis", Pp. 226-278, 1969, with permission from Elsevier*)

2.4 Yield surface

A material exposed to a load deforms either in an elastic or plastic manner, depending on the critical stress level known as the yield stress. Below the yield stress, the material would behave elastically, which means that it will not permanently deform and it will return to its original state once unloading occurs. However, when the yield stress is reached or exceeded, the material undergoes permanent plastic deformation. When the material undergoes another cycle of reloading and unloading, it can be observed that the new yield stress upon reloading corresponds to the stress level just prior to previous unloading. When multiple force systems are applied, the yield criterion is not just a single number, but is expressed as as function $f(\sigma_{ij})$ that would describe the yield surface. In general, when $f(\sigma_{ij}) < c$, where $c =$ constant, the deformation is elastic. If $f(\sigma_{ij}) = c$, the deformation is plastic [24].

In crystal plasticity, yield surfaces are used to determine the onset of plastic deformation for individual crystals or grains within a polycrystalline material. The relationship between continuum plasticity theories and crystal plasticity models lies with the Taylor factor m and the yield surface exponent A. The flow stress σ can be related to the CRSS τ_c through the Taylor factor m as: $\sigma = m\tau_c$ [25]. Experimental measurements for the Taylor factor m is challenging, but it can be estimated or calculated using crystal plasticity models.

A yield function can be used to approximate the yield surface in classical plasticity theory, and it can be written in the form: $f(\boldsymbol{\sigma}) = \varphi(\boldsymbol{\sigma}) - \sigma_Y$. Here, the equivalent stress corresponds to $\varphi(\boldsymbol{\sigma})$ and σ_Y corresponds to the yield stress of the material. For isotropic materials, the yield function is expressed as a function of the principal stresses $(\sigma_1, \sigma_2, \sigma_3)$ and the most widely used function is the Von-Mises yield criterion. Hershey [26] and Hosford [27] proposed a non-quadratic form of this criterion shown in Equation 2.22. In this function, $\boldsymbol{\sigma}$ is the Cauchy stress tensor and the value for the exponent A depends on crystal structure [24].

$$f(\boldsymbol{\sigma}) = 2^{\frac{-1}{A}}(|\sigma_1 - \sigma_2|^A + |\sigma_2 - \sigma_3|^A + |\sigma_3 - \sigma_1|^A)^{\frac{1}{A}} - \sigma_Y \tag{2.22}$$

2.5 R-values

The plastic anisotropy of a material can be quantified using a metric known as the R-value or Lankford coefficient. The R-value is the ratio of the plastic strain rate across the width of a sample and the magnitude of the plastic strain rate in the thickness of the sample produced by tensile loading, and it is a function of the angle θ between the rolling direction and uniaxial tension axis, as shown in Equation 2.23. Here, D^p_{22} is the strain rate along the width and D^p_{33} is the strain rate along thickness of the materials [28].

$$R(\theta) = \frac{D^p_{22}}{D^p_{33}} \tag{2.23}$$

The R-values can change depending on the sample's orientation, thus it can vary based on a material's crystallographic texture, which makes it a measure of the directional dependence of plastic deformation. Large R-values indicate good material formability as there is little deformation along the thickness which is desired as it prevents localization and eventually failure. On the contrary, low R-values indicate poor material formability [28].

Chapter 3

Software development

3.1 Requirements and specifications

The main goal of the master's thesis is to deploy a crystal plasticity toolbox to cloud which would be able to generate visualization for crystal plasticity models. The functional requirements for the application are listed in Table 3.1, while the the non-functional requirements are listed in Table 3.2.

Table 3.1: Crystal plasticity toolbox functional requirements

ID	Description
F-REQ-01	Crystal Plasticity Model: User should be able to generate crystal plasticity visualizations for Taylor model
F-REQ-02	Crystal Plasticity Model: User should be able to generate crystal plasticity visualizations for Alamel model
F-REQ-03	Grain Orientation Setup: User should be able to select pre-defined texture components (Cube, Brass, S, Goss, Copper), specify spread and number of grains
F-REQ-04	Grain Orientation Setup: User should be able to select random orientation and spread, and specify number of grains
F-REQ-05	Grain Orientation Setup: User should be able to select fiber texture component, specify spread, number of grains, Euler angles and axis
F-REQ-06	Grain Orientation Setup: User should be able to select to input custom spread, number of grains and Euler angles
F-REQ-07	Grain Orientation Setup: User should be able to mix multiple texture components in the polycrystal
F-REQ-08	Grain Orientation Setup: User should be able to upload *.ori* files as source for the polycrystal

Continued on next page

Table 3.1 – continued from previous page

ID	Description
F-REQ-09	Grain Orientation Setup: User should be able to upload *.ang* files as source for the polycrystal, and enter threshold
F-REQ-10	Loading Conditions Setup: User should be able to select from predefined loading conditions (Uniaxial Tension, Plane-Strain Compression, Mixed Boundary Condition)
F-REQ-11	Loading Conditions Setup: User should be able to select to input custom loading values for Global Velocity Gradient (L)
F-REQ-12	Loading Conditions Setup: User should be able to specify total Von Mises Strain
F-REQ-13	Loading Conditions Setup: User should be able to specify total number of steps for loading
F-REQ-14	Pole Figures: User should be able to generate initial and loaded pole figures
F-REQ-16	Pole Figures: User should be able to select projection type, which would be either stereographic or area
F-REQ-17	Pole Figures: User should be able to select to view either point projection or contour pole figures
F-REQ-18	Pole Figures: User should be able view pole figure for (100), (110) and (111) FCC planes
F-REQ-19	Inverse Pole Figures: User should be able to generate initial and loaded inverse pole figures
F-REQ-20	Inverse Pole Figures: User should be able to view the trajectory of the Loaded Inverse Pole Figures
F-REQ-21	ODF: User should be able to see initial and loaded ODF figures
F-REQ-22	ODF: User should be able to change the ODF levels and regenerate the figure
F-REQ-23	Yield Surface: User should be able to view initial and loaded yield surfaces
F-REQ-24	Yield Surface: User should be able select the x and y axis for the yield surface
F-REQ-25	R-values: User should be able to view initial and loaded R-values
F-REQ-26	R-values: User should be able select the normal axis for the R-values

Table 3.2: Crystal plasticity toolbox non-functional requirements

ID	Description
NF-REQ-01	Generate Python executable module for existing crystal plasticity Fortran codes (`cko2odf.f`, `gauss_odf.f`, `gauss_odf_wrap.f`, `simplx.for`, `minpack.f90`, `globals.f90`, `linesearch.f90`, `utils.f90`, `crystal_plasticity.f90`)
NF-REQ-02	Web appplication should be deployed to cloud
NF-REQ-03	Figure generation should be relatively "fast"
NF-REQ-04	Web application should be user-friendly and intuitive
NF-REQ-05	Code should be easily understandable (proper comments)
NF-REQ-06	Application should be available for use by the public (open-source)

3.2 Technology and tools

Identifying technologies and tools that would best suit the project is one of the most important parts in software development. In this section, the tools used to develop the cloud-based crystal plasticity toolbox is presented.

3.2.1 Python

Python is an open-source programming language, which supports object-oriented, procedural and functional programming. It is used in various applications such as scientific computing, data analysis, and web development. It is a widely used programming language known by its simplicity and readability, making it easy to learn. It has a large standard library that provides modules and functions for various needs. Aside from the standard libraries, it also has a wide-range of packages that can be easily installed through package managers [29].

It was known from the beginning of the project that existing Fortran codes for crystal plasticity calculations would be used for the application. This is because Fortran is known to be highly optimized for numerical calculations, which makes it useful for scientific and engineering applications [30]. However, Fortran, by itself, is not capable of creating modern and user-friendly GUI, thus the need for another programming language which has both the possibility of using the existing Fortran code and creating a sleek and modern web-based application. Thus, Python was selected as the main programming language for the development of the cloud-based crystal plasticity toolbox.

3.2.2 F2PY

Fortran to Python interface generator (F2PY) is a tool that simplifies the process of integrating Fortran code to Python applications. It is used to generate Python extension modules that enables calling external subroutines for Fortran 77/90/95, module subroutines for Fortran 90/9, and C functions, within the Python code just like any other Python module. Access to COMMON blocks for Fortran 77 and module data for Fortran 90/95 can also be enabled through this [31]. F2PY is included as sub-package within Numerical Python (NumPy) Python library [32].

To create the executable Python extension from Fortran code, the following are needed: Fortran source code, signature file, and a Fortran compiler to generate the extension modules. The following are the steps to run F2PY [31]:

1. The Fortran code to be used in Python should be ready and compatible with F2PY and the Fortran compiler chosen.
2. Generate the signature file (.pyf) for the Fortran code by running a similar code below:

   ```
       f2py -h <signature filename.pyf> <options> <fortran source files>
   ```

   This would generate the `.pyf` signature file, which contains the information for integration to the Fortran code `fortrancode.f90`.
3. Construct the extension module by running a similar code below, wherein `<fortran source files>` can include signature files:

   ```
       f2py -m <module name> <options> <fortran source files>
   ```

4. Generate the extension module by running a similar code below, wherein `<fortran source files>` can include signature files:

   ```
       f2py -c <options> <fortran source files>
   ```

   This command compiles the Fortran code and generates a Python extension module.

When the aforementioned steps are run in a Windows environment, the generated extension module would be in *.pyd* file. When run in a Linux environment, the generated extension module would be a *.so* file. The extension modules can be used in Python through the usual `import` of modules:

```
from <module name> import <function>
import <module name> as <module alias>
```

3.2.3 Dash

Dash was chosen as the main development platform to create the cloud-based crystal plasticity web application, since it utilizes the open-source Python framework and it can be used to build web applications for interactive data visualization. Dash is a low-code platform, thus it enables fast creation of a modern and interactive user interface. Technologies such as `Flask` (web server), `React.js` (GUI), and `Plotly.js` (plotting or chart generation tool) are inherently integrated in this platform, which provides a variety of components and tools for creating charts, graphs, tables, and other visualizations. Dash is licensed under the permissive

Massachusetts Institute of Technology (MIT) software license and was introduced in 2017 [33].

Dash installation can be done by running the following command in the terminal [34]:

```
pip install dash
```

Dash consists of two fundamental parts:

1. layout
2. callback functions

The layout part controls the appearance and structure of the Dash application. It determines how the application is visually presented to the the user. It is in the layout where the GUI elements, such as graphs, buttons, texts, tabs, and input fields, are declared. Also, as the component name suggests, it defines the layout of the application or the placement of the GUI elements, which is represented in the code as a hierarchical component tree. When Dash is installed, it includes the Dash HTML (Hypertext Markup Language) Components module (`dash.html`) and Dash Core Components module (`dash.dcc`) which are the most commonly used modules to build the layout. Styling and creation of HTML elements is done by `Dash.html` as it provides a set of classes and functions for generating HTML tags and attributes within the Python code. Other pre-built components can be created through `dash.dcc`, which does not only use HTML, but also uses CSS (Cascading Style Sheets), JavaScript and `React.js`, and this makes it easier to create a more complex and interactive GUI. The `dash.dcc` graphs are created through `Plotly.js` plotting library, which is also included in the Dash installation. To further customize the GUI, a custom CSS can be used to change the font or even modify the default element styles [35]. Dash Bootstrap Components (`dash_bootstrap_components.themes`) is a library that provides a pre-built set of Bootstrap components and themes, which extends the default functionalities provided by `dash.dcc`, and this can be installed by running the following command [36]:

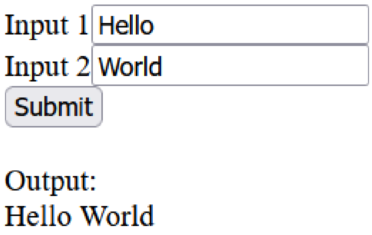
```
pip install dash-bootstrap-components
```

Callback functions make interactivity possible in Dash applications. These functions define how an output component's value or property will be changed based on changes done to the value or property of an input component. These functions act as a decorator, wherein an input and output is specified and the function takes in the input values as arguments and returns the updated values as output [37]. An example of a basic callback function is shown in Code listing 3.1, wherein the `@app.callback` decorator will call the function `update_output` when the "`value`" `component_property` of "`app-input`" `component_id` is changed. This would then update the "`children`" `component_property` of "`app-output`" `component_id`. In the `app.layout`, this would mean that when the user enters value in the `dcc.Input` field, the entered text will be shown in the `html.Div` section of the GUI.

Code listing 3.1: Example code for layout and basic callback function

```
app.layout = html.Div([
    dcc.Input(id='app-input',type='text'),
    html.Div(id='app-input')
])

@app.callback(
    Output(component_id='app-output', component_property='children'),
    Input(component_id='app-input', component_property='value')
)
def update_output(input_value):
    return value_output
```

Callbacks are automatically triggered by Dash when an input component is changed, but, in most use cases, multiple inputs are needed before the callback function should be run. In these cases, `State` can be utilized to make Dash wait for a trigger, which is usually a button click, before running the function [37]. An example of an implementation utilizing `State` can be visualized in Figure 3.1 and the sample code can be seen in Code listing 3.2.



Input 1 Hello
Input 2 World

Output:
Hello World

Figure 3.1: An example of a basic user interface for the `State` implementation in a callback function.

Code listing 3.2: Example code for the `State` implementation in a callback function for Figure 3.1

```
app.layout = html.Div([
    html.Label('Input␣1'),
    dcc.Input(id='input-1', type='text'),
    html.Br(),
    html.Label('Input␣2'),
    dcc.Input(id='input-2', type='text'),
    html.Br(),
    html.Button(id='submit-button-state', n_clicks=0, children='Submit'),
    html.Br(),
    html.Br(),
    html.Label('Output:'),
    html.Div(id='output-1')
    ])

@app.callback(
    Output('output-1', 'children'),
    Input('submit-button-state','n_clicks'),
    State('input-1', 'value'),
    State('input-2', 'value'),
    prevent_initial_call=True)
```

```
def update_ouput(n_clicks, input_1, input_2):
    output_value = input_1 + "␣" + input_2
    return output_value
```

3.2.4 VS Code

Visual Studio Code (VS Code) was the source code editor utilized during the implementation of the crystal plasticity toolbox. VS Code is an open-source and powerful code editor that can be used in Windows, MacOS and Linux operating systems. It is lightweight, but it provides support to a wide-range of programming languages, including Python and Fortran. It has built-in integration to other software development tools, such as Gitlab for source control management, Docker for environment setup and deployment, and Azure for cloud hosting. VS Code also has IntelliSence, which makes coding easier as it suggests intuitive code completion, syntax highlighting, and auto-formatting [38].

3.2.5 Docker

Docker is an open-source platform for building, running, and deploying application in containers, which are standardized units of software packages. Docker enables the creation of an image, which is a read-only package that includes all components needed to run the application, such as the code, run-time settings and other dependencies. The instructions on how to create and run the image is defined in a `Dockerfile`. Each step in the `Dockerfile` creates a layer of the image, thus every time an instruction is changed only the changed part is rebuilt, making the builds faster and lightweight. An image becomes a container during run-time, which is an isolated environment for running an application. Since the container already contains everything that is needed to run the application, consistent run-time of the application is assured regardless of the choice of hosting or infrastructure, as long as it supports Docker [39]. It was eventually decided to utilize Docker in this project to build and deploy the crystal plasticity toolbox to cloud, since there were a lot of package dependencies and with Docker containers it made it less complicated to deploy the application to cloud.

3.2.6 Azure

One of the main goals of this project to ensure that the crystal plasticity toolbox is hosted in cloud, making it readily available for anyone with an internet connection. To be able to do this, a hosting solution in the cloud is needed and Azure was utilized for this project. Aside from Azure’s compatibility and seamless integration to the other tools and technologies used, it also offers free credit to students up to a certain amount, thus it was chosen as the hosting platform for this project.

Azure is a cloud computing platform created by Microsoft. Even though it is tailor-fitted to work best with other Microsoft tools and Windows-based services, it is compatible with various programming languages, frameworks and infrastructures. It has a wide range of tools and services for building, managing and deploying cloud-based applications, such as virtual machines, databases, container registries and applications services [40]. The two Azure products used for this project are the Azure App Service and the Azure Container Registry. Azure App Service was used for the of hosting of the web application. This product supports various programming languages, such as Python, and it also supports both Windows and Linux operating systems. [41]. Azure Container Registry was used for the storage and management of the Docker container images [42]

3.3 Software architecture

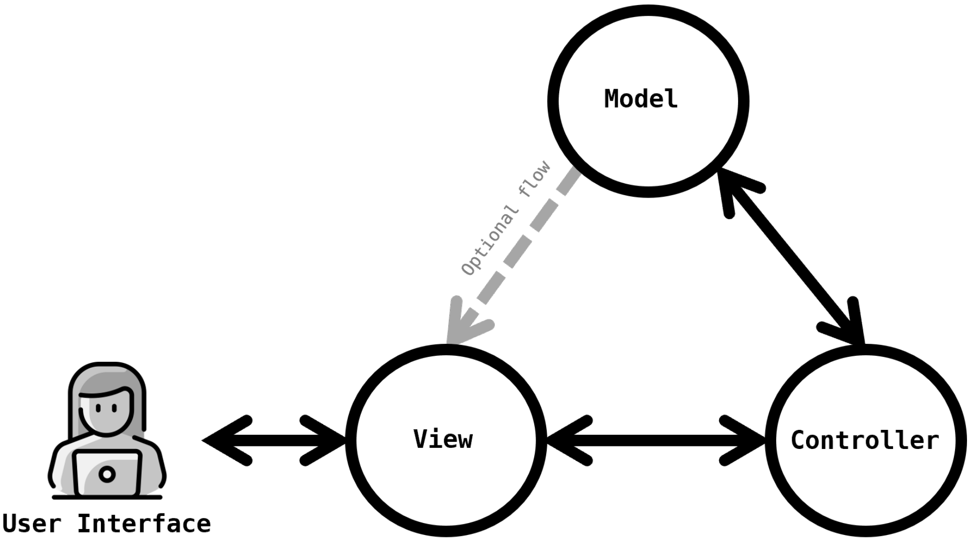


Figure 3.2: Generic data flow diagram for a MVC architecture [43].

Model-Controller-View (MVC) is a software architecture pattern which is a common framework used for building web applications. MVC separates the application into three interconnected components [43]:

- The **model** is the part that controls and defines how data is stored, retrieved and manipulated. This represents data storage for the application.
- The **view** is the part that renders the user interface and handles user interaction or inputs. This is the tangible part of the application for an end-user.
- The **controller** is the part that acts as the intermediary between the model and view. It processes business logic to manage data in the model, and it also instructs the view part what to render as output.

Having these three separated and loosely coupled allows for separation of concerns in an application, which means that one component can be changed without necessarily affecting the other components. This enables easier software implementation, testing and maintenance.

Figure 3.2 shows a simplified data flow representation between the three components. A user can interact through a GUI, like in a web browser, and the view component receives the user input. The view sends the input to the controller for processing based on business logic. The controller could interact with the model to update or manage data, and it could also send back instructions to view [43].

LOADING CONDITION Plane-strain compression

TOTAL VON MISES STRAIN 1

NUMBER OF STEPS 100

Generate yield surface for the following:

☑ Initial ☑ Loaded

X-axis 11

Y-axis 22

Figure 3.3: High-level layout of application.

Figure 3.3 shows that the GUI of the application is divided into two: the sidebar and content. The sidebar is the part of the GUI where the user can control the setup of the polycrystal, while the content is the part where the graphs and other visualization will be rendered. The application follows the MVC pattern by separating the code defining the GUI and the business logic. As can be seen under the "layout" folder there is a "content" folder that is separated into `content.py`, which contains the application callbacks, and `content_callbacks.py`, which contains the code defining the look of the application. The same pattern is also done for the "sidebar" part of the application, which is separated into `sidebar.py` and `sidebar_callbacks.py`. The main layout is defined in `layout.py`, which combines both `content.py` and `sidebar.py`.

Figure 3.4 shows `datasource.py`, which is located under the "main" folder, and it contains the code that acts as a controller. This Python file contains functions for the generation of the data and figures, wherein it connects with the modules generated from pre-existing Fortran codes (`auswert.cpython-37m-x86_64-linux-gnu.so`, `cpfort.cpython-37m-x86_64-linux-gnu.so`), and Python codes (`crystal_plasticity_module.py`, `odflib.py`, `TexTool_Driver.py`, `TexTool_Funcs.py`). The code that processes the uploaded files is in `upload_file.py`. The entry point of the application would be `app.py`, which defines the main Dash application.

The GUI style used for the application is the open-source Bootstrap theme Lux [44]. Using a predefined theme for the styling enabled faster development. Other customization in the styling were done through CSS in `input.css` which is stored under the "assets" folder.

```
main
├── assets
│   └── input.css
├── layout
│   ├── content
│   │   ├── content.py
│   │   └── content_callbacks.py
│   └── sidebar
│       ├── sidebar.py
│       └── sidebar_callbacks.py
├── uploads
├── .gitignore
├── app.py
├── auswert.cpython-37m-x86_64-linux-gnu.so
├── cpfort.cpython-37m-x86_64-linux-gnu.so
├── crystal_plasticity_module.py
├── datasource.py
├── Dockerfile.prod
├── GenTex.py
├── liba1.dat
├── liba23.dat
├── libbu.dat
├── libmln.dat
├── libxk.dat
├── odflib.py
├── requirements.txt
├── sslookup.dat
├── TexTool_Driver.py
├── TexTool_Funcs.py
└── upload_file.py
```

Figure 3.4: Application file tree structure.

3.4 Generation of python extension modules

Fortran codes for crystal plasticity calculations, which were developed at the Department of Materials Science and Engineering at NTNU, were converted into Python executable modules using F2PY. Initially, since development was done locally in a Windows 11 64-bit operating system machine without deployment to cloud, Windows-compatible modules were generated by following the steps below:

1. Download and install the following:
    - Anaconda for Windows [45]
    - Intel® Fortran Compiler [46]
2. Update the content of `site.cfg` file to the following:

```
[mkl]
mkl_libs = mkl_lapack95_lp64,mkl_blas95_lp64,mkl_intel_lp64,mkl_intel_thread,
mkl_core,libiomp5md
lapack_libs = mkl_lapack95_lp64,mkl_blas95_lp64,mkl_intel_lp64,
mkl_intel_thread,mkl_core,libiomp5md
library_dirs = C:/Program Files (x86)/Intel/oneAPI/mkl/latest/lib/intel64;
C:/Program Files (x86)/Intel/oneAPI/compiler/latest/windows/compiler/lib/
intel64_win
include_dirs = C:/Program Files (x86)/Intel/oneAPI/mkl/latest/include
```

   This file can be found in the following location in the local machine: `C:\User\<username>\anaconda3\Lib\site-packages\numpy\distutils`
3. Open a command line tool and by using `conda`, which is included in the installation in step 1, create an environment for Python 3.7.5:

```
conda create --name <env name> python=3.7.5
```

4. Activate the environment that was recently create by running the following:

```
conda activate <env name>
```

5. Ensure that the environment created will have the pre-requisite packages by installing `requirements.txt` using `conda`. Refer to Code listing B.1 in Appendix B for content of the requirements file. Run the following in the command line tool and ensure that this is done in the same folder that contains the `requirements.txt`:

```
conda install -n <env_name> requirements.txt
```

6. Activate the Intel® Fortran Compiler, installed in step 2, by running the following in the command line tool:

```
"C:\Program Files (x86)\Intel\oneAPI\compiler\latest\env\vars.bat" intel64
```

7. In command line tool, navigate to the folder which contains the Fortran files:
    - `cko2odf.f`
    - `gauss_odf.f`
    - `gauss_odf_wrap.f`

  - `simplx.for`
  - `minpack.f90`
  - `globals.f90`
  - `linesearch.f90`
  - `utils.f90`
  - `crystal_plasticity.f90`

8. In command line tool, run the following to generate the Python executable modules:

```
f2py -m auswert -c auswert.pyf cko2odf.f gauss_odf.f gauss_odf_wrap.f

f2py --link-lapack_opt -m cpfort -c simplx.for minpack.f90 globals.f90
linesearch.f90 utils.f90 crystal_plasticity.f90
```

9. The following Windows-compatible Python executable modules should be generated in the same folder that contains the Fortran files:

```
auswert.cp37-win_amd64.pyd
cpfort.cp37-win_amd64
```

The need to generate Linux-compatible Python executable modules arose when attempts to deploy the web application to cloud were done. The following steps were followed to generate Linux-compatible modules through a machine with Ubuntu as the Linux distribution:

1. Download and install the following:
  - Intel® oneAPI Math Kernel Library (oneMKL) for Linux [47]
  - Anaconda for Linux [45]
2. Open the terminal and run the following command in the same folder the Anaconda installer was downloaded:

```
bash <installer file name>.sh
```

3. In the terminal and by using `conda`, which is included in the installation in step 2, create an environment for Python 3.7.5:

```
conda create -n <env name> python=3.7.5
```

4. Activate the environment that was recently create by running the following:

```
conda activate <env name>
```

5. In the terminal and by using `conda`, install NumPy, which includes F2PY, by running the following:

```
conda install numpy
```

6. In the terminal, install `gfortran`, by running the following:

```
sudo apt-get install gfortran
```

7. In the terminal, navigate to the folder which contains the Fortran files:

   - `cko2odf.f`
   - `gauss_odf.f`
   - `gauss_odf_wrap.f`
   - `simplx.for`
   - `minpack.f90`
   - `globals.f90`
   - `linesearch.f90`
   - `utils.f90`
   - `crystal_plasticity.f90`

8. In terminal, run the following to generate the Python executable modules:

```
f2py --fcompiler=gnu95 -m auswert -c auswert.pyf cko2odf.f gauss_odf.f
gauss_odf_wrap.f

f2py --fcompiler=gnu95 --f90flags="-ffree-line-length-none" -m cpfort -c
simplx.for minpack.f90 globals.f90 linesearch.f90 utils.f90
crystal_plasticity.f90 -L/opt/intel/oneapi/mkl/latest/ib/intel64
-lmkl_intel_lp64 -lmkl_sequential -lmkl_core -lpthread -lm -ldl
```

9. The following Linux-compatible Python executable modules should be generated in the same folder that contains the Fortran files:

```
    auswert.cpython-37m-x86_64-linux-gnu.so
    cpfort.cpython-37m-x86_64-linux-gnu.so
```

During implementation, the Windows-compatible modules (*.pyd* file extension) were used for fast debugging in the the Windows machine used for development. However, for deploying the application to cloud, the Linux-compatible modules (*.so* file extension) were used.

3.5 Build and deployment

In this section, the process of deploying the Dash application to a local development environment and to the cloud is discussed. Figure 3.5 shows the high-level deployment process in three steps. Step one involves building the Docker image and running the container locally, which is the step required to deploy the application locally for testing and debugging. Steps two and three involves the steps needed for deployment to production or to cloud, wherein the Docker image would be uploaded to Azure container registry and using the uploaded image in Azure app service to make the application available in cloud [48].

Prior to deployment of the application, the following pre-requisites should be done:

- VS Code [49] is installed.
- Docker Desktop [50] is installed .
- There is an active Azure subscription and user credentials.
- Files needed for the application are stored in a folder in the local machine following the file tree structure in Figure 3.4. The files should already in-

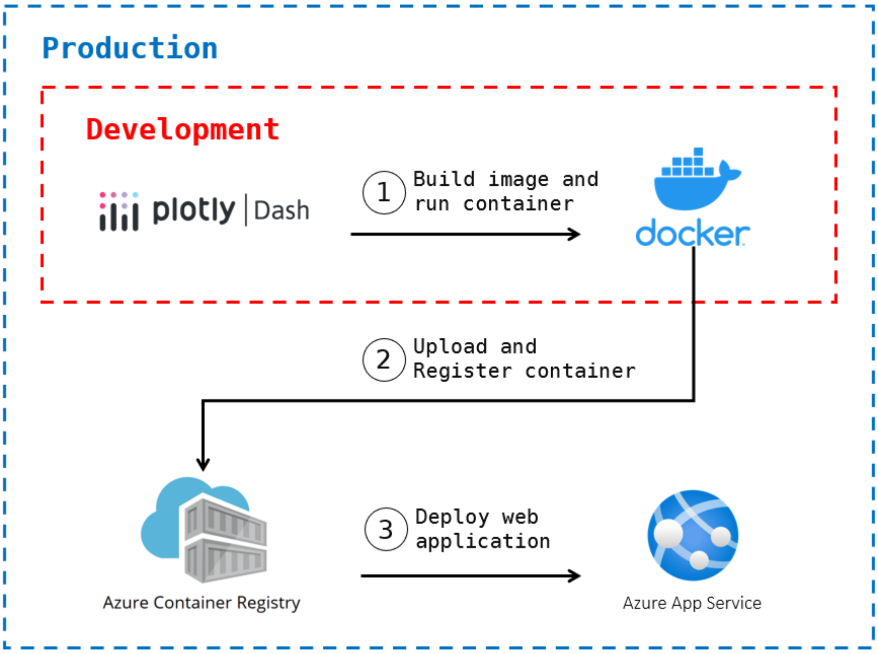


Figure 3.5: Deployment flow for local development and production environments [48].

clude the Linux-compatible Python executable modules generated through the steps described in Section 3.4. These can be downloaded in the Gitlab link shown in Appendix B.

- Development setup in VS Code should be done. In VS Code, open the folder containing the application files. Then, navigate to the command palette and enter "*Python: Select Intepreter*". Select the environment created in Section 3.4 with Python version 3.7.5.

The following are the detailed steps how to create a Dash application running on Docker in a local development environment or machine:

1. In the VS Code terminal, ensure you are in the path of the main folder that contains the application files and run the following to build the Docker image:

   ```
   docker build -t <Docker image name>:<version> -f Dockerfile.prod .
   ```

   For example, the Docker image name could be `crystal-plasticity-toolbox`. The version should be incremented if image version control is desired. Thus, for version 1.0 for `crystal-plasticity-toolbox` Docker image, the following should be run:

   ```
   docker build -t crystal-plasticity-toolbox:1.0 -f Dockerfile.prod .
   ```

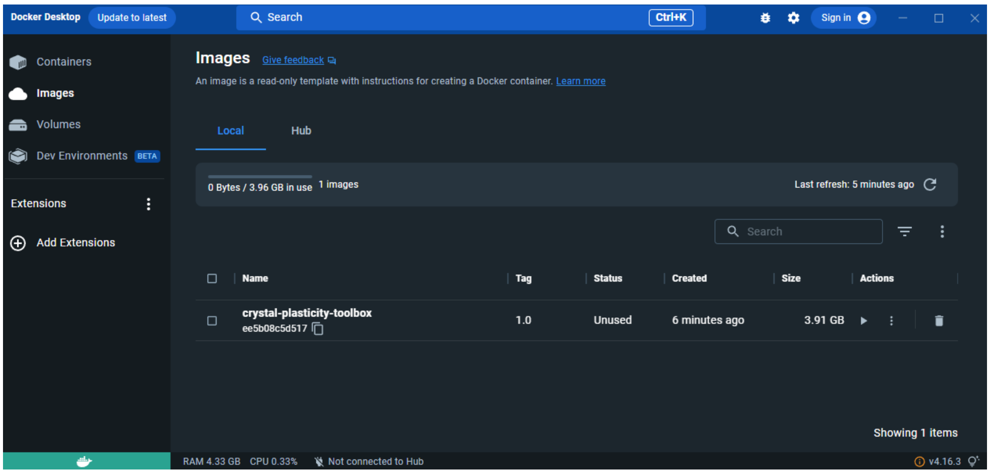
2. After the build of the Docker image is done, Docker desktop can be opened to see the recently built image (see Figure 3.6). This image can be run in a

container by either clicking the play button under *Actions* column in the *Images* section in the Docker Desktop application, or by running the following in the VS Code terminal:

```
docker run -d -p 80:80 <Docker image name>:<version>
```

For the example `crystal-plasticity-toolbox` Docker image version 1.0, the following should be run:

```
docker run -d -p 80:80 crystal-plasticity-toolbox:1.0
```



Images Give feedback

An image is a read-only template with instructions for creating a Docker container. Learn more

0 Bytes / 3.96 GB in use 1 images

Last refresh: 5 minutes ago

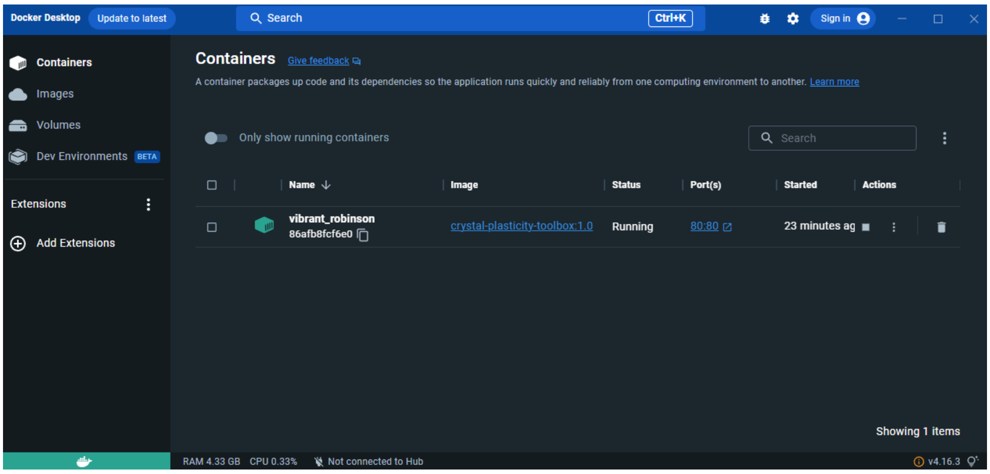
☐	Name	Tag	Status	Created	Size	Actions
☐	crystal-plasticity-toolbox ee5b08c5d517	1.0	Unused	6 minutes ago	3.91 GB	

Showing 1 items

RAM 4.33 GB CPU 0.33% Not connected to Hub v4.16.3

Figure 3.6: Docker image in Docker desktop application.

3. The Dash application can now be accessed through the web browser, by either clicking the link under *Port(s)* column in the *Containers* section in the Docker Desktop application (see Figure 3.7), or by opening the browser and accessing `http://localhost/`.



Containers Give feedback

A container packages up code and its dependencies so the application runs quickly and reliably from one computing environment to another. Learn more

Only show running containers

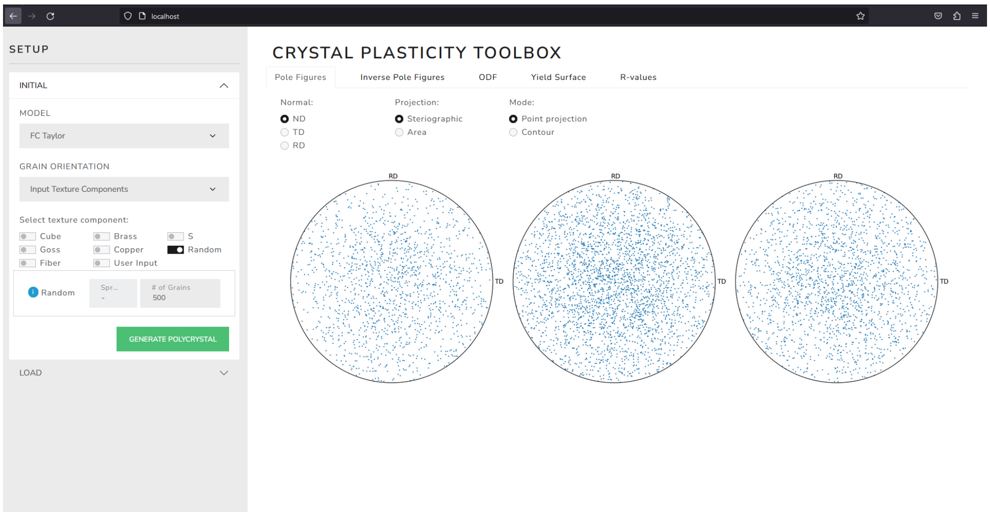
☐	Name	Image	Status	Port(s)	Started	Actions
☐	vibrant_robinson 86afb8fcf6e0	crystal-plasticity-toolbox:1.0	Running	80:80	23 minutes ag	

Showing 1 items

RAM 4.33 GB CPU 0.33% Not connected to Hub v4.16.3

Figure 3.7: Docker container in Docker desktop application.

This local instance of the crystal plasticity toolbox (see Figure 3.8) running in Docker can be used for local testing and quick debugging. If there are issues found and if any of the code is changed, steps 1 and 2 need to be done again to re-build the Docker image.



MODEL: FC Taylor

GRAIN ORIENTATION: Input Texture Components

Select texture component:
☐ Cube ☐ Brass ☐ S
☐ Goss ☐ Copper ☑ Random
☐ Fiber ☐ User Input

Random | Spr... - | # of Grains 500

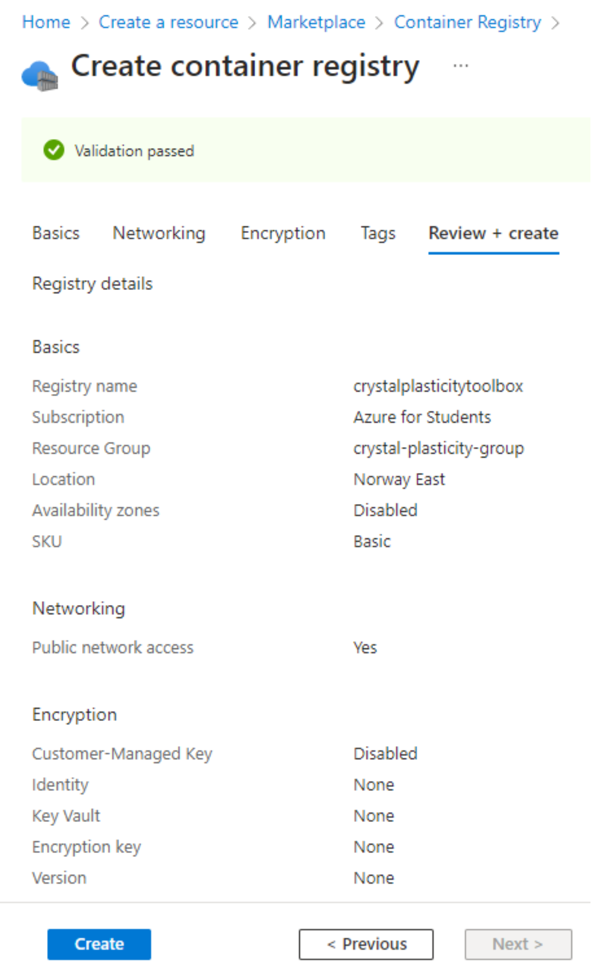
Normal: ● ND ○ TD ○ RD

Projection: ● Steriographic ○ Area

Mode: ● Point projection ○ Contour

Figure 3.8: Crystal plasticity toolbox application running in the local environment through Docker.

4. To proceed with deployment to cloud, the Docker image created in step 2 would need to be registered in Azure Container Registry. In the Azure marketplace, search for "*Container registries*" and create a new container registry by filling out all the required fields (see Figure 3.9). Take note of the details of the container registry, like the name and the resource group, as these will be needed in the succeeding steps.



Create container registry

Validation passed

Registry details

Basics

Registry name	crystalplasticitytoolbox
Subscription	Azure for Students
Resource Group	crystal-plasticity-group
Location	Norway East
Availability zones	Disabled
SKU	Basic

Networking

Public network access	Yes

Encryption

Customer-Managed Key	Disabled
Identity	None
Key Vault	None
Encryption key	None
Version	None

Figure 3.9: Creation of new container registry in Azure.

5. In the VS Code terminal, run the following and the web browser will be opened for login to Azure:

```
az login
```

6. Once login to Azure is successful, run a similar command in the VS Code terminal to access the Azure container registry:

```
az acr login --name <registry name>
```

   Enter the registry name in step 4, thus the following will be run for the example given:

```
az acr login --name crystalplasticitytoolbox
```

7. Once "Login Succeeded" is shown in the terminal, the Docker image can now be pushed to Azure container registry by running similar commands:

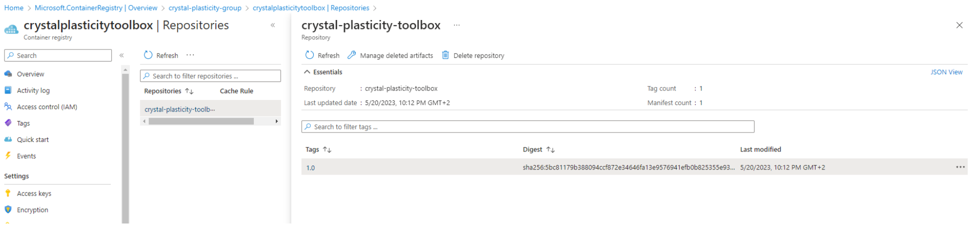
```
docker tag <Docker image name>:<version> <registry name>.azurecr.io/<Docker
```

```
image name>:<version>
docker push <registry name>.azurecr.io/<Docker image name>:<version>
```

For the example given in the previous steps, the commands should look like the following:

```
docker tag crystal-plasticity-toolbox:1.0 crystalplasticitytoolbox.azurecr.io
/crystal-plasticity-toolbox:1.0
docker push crystalplasticitytoolbox.azurecr.io/crystal-plasticity-toolbox:1.0
```

8. After the push to repository is done, the Docker image should be listed as a repository in Azure container registry as can be seen in Figure 3.10.



Repository : crystal-plasticity-toolbox
Last updated date : 5/20/2023, 10:12 PM GMT+2
Tag count : 1
Manifest count : 1

Tags	Digest	Last modified
1.0	sha256:5bc81179b388094ccf872e34646fa13e9576941efb0b825355e93...	5/20/2023, 10:12 PM GMT+2

Figure 3.10: Docker image pushed to Azure container registry.

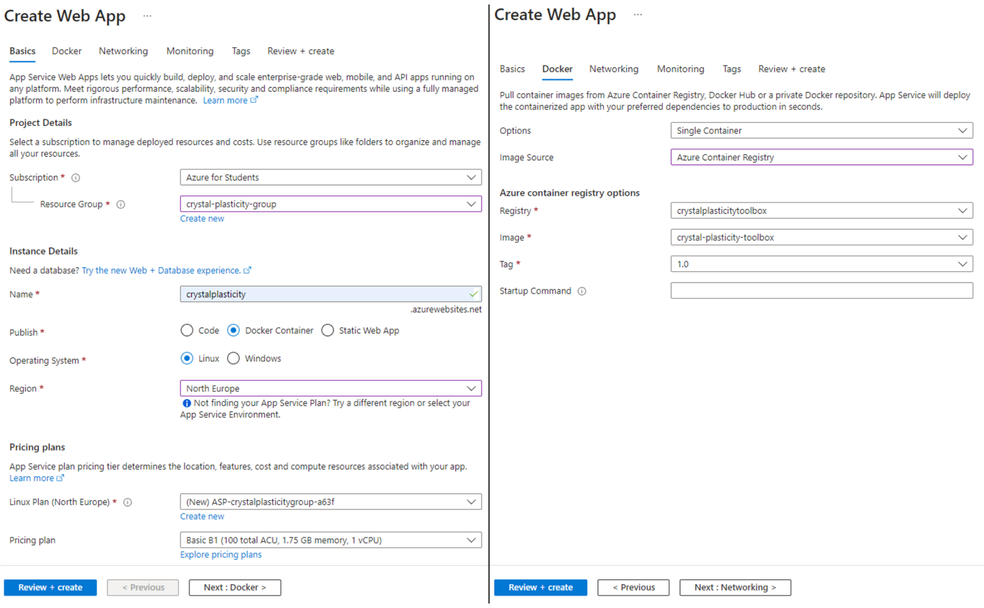
9. Enable admin user rights for the container registry by running a similar command in the terminal:

```
az acr update -n <registry name> --admin-enabled true
```

For the example given in the previous steps, the command should look like the following:

```
az acr update -n crystalplasticitytoolbox --admin-enabled true
```

10. The Docker image can now be deployed as a web application through Azure App Service. In the Azure marketplace, search for "*Web app*" and create a web application by filling out all the required fields (see Figure 3.11). The value entered in the *Name* field will determine the URL of the web application. In the *Basics* tab, ensure to select "*Docker Container*" value in the *Publish* field and "*Linux*" value in the *Operating System* field. In the *Docker* tab, select "*Azure Container Registry*" value in *Image Source* field and select the registry created in step 4 in *Registry* field.



Create Web App ···

App Service Web Apps lets you quickly build, deploy, and scale enterprise-grade web, mobile, and API apps running on any platform. Meet rigorous performance, scalability, security and compliance requirements while using a fully managed platform to perform infrastructure maintenance. Learn more

Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription * Azure for Students
Resource Group * crystal-plasticity-group
Create new

Instance Details

Need a database? Try the new Web + Database experience.

Name * crystalplasticity .azurewebsites.net
Publish * ○ Code ● Docker Container ○ Static Web App
Operating System * ● Linux ○ Windows
Region * North Europe
Not finding your App Service Plan? Try a different region or select your App Service Environment.

Pricing plans

App Service plan pricing tier determines the location, features, cost and compute resources associated with your app. Learn more

Linux Plan (North Europe) * (New) ASP-crystalplasticitygroup-a63f
Create new
Pricing plan Basic B1 (100 total ACU, 1.75 GB memory, 1 vCPU)
Explore pricing plans

Create Web App ···

Pull container images from Azure Container Registry, Docker Hub or a private Docker repository. App Service will deploy the containerized app with your preferred dependencies to production in seconds.

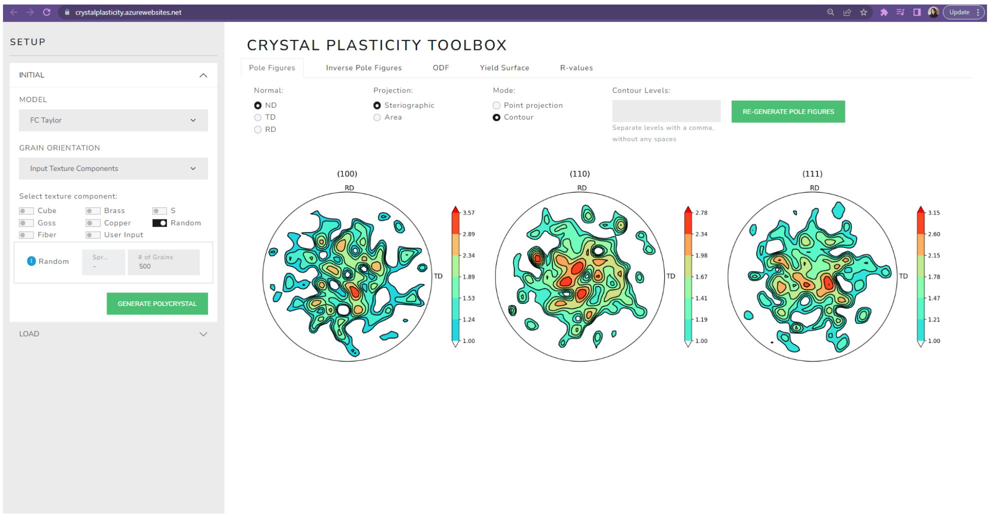
Options Single Container
Image Source Azure Container Registry

Azure container registry options

Registry * crystalplasticitytoolbox
Image * crystal-plasticity-toolbox
Tag * 1.0
Startup Command

Figure 3.11: Creation of new web application in Azure.

11. Azure will now start running a container with the Docker image pushed in the container registry. Access the web application by either clicking *Browse* button in the Azure web page or by accessing the web URL in a browser. In the example provided in the previous steps, the URL is `https://crystalplasticity.azurewebsites.net/`.



crystalplasticity.azurewebsites.net

SETUP

INITIAL

MODEL: FC Taylor

GRAIN ORIENTATION: Input Texture Components

Select texture component:
☐ Cube ☐ Brass ☐ S
☐ Goss ☐ Copper ☑ Random
☐ Fiber ☐ User Input

Random Spr... - # of Grains 500

CRYSTAL PLASTICITY TOOLBOX

Normal: ● ND ○ TD ○ RD

Projection: ● Steriographic ○ Area

Mode: ○ Point projection ● Contour

Contour Levels:

Separate levels with a comma, without any spaces

Figure 3.12: Crystal plasticity toolbox application running in cloud.

Chapter 4

Results and discussion

4.1 Implementation

4.1.1 Generation of initial and loaded polycrystal

The Python code that acts as the controller, `datasource.py`, is called when the user clicks "Generate Polycrystal" button in the GUI, and it triggers the generation of the initial polycrystal data. Based on the selected grain orientation value, it either calls `TexTool_Driver.py` or `crystal_plasticity_module.py` to generate the orientations. If the selected grain orientation setup for the initial polycrystal is to input texture components, and the selected texture components are either cube, brass, S, goss, copper, random, or user defined, then the orientation generation is through `TexTool_Driver.py` with `GenTex.py` or `TexTool_Funcs.py`, which are from the TexTool application [51]. If the selected texture component is fibre, then the orientation generation is through `crystal_plasticity_module.py`. If the user uploads a supported file (*.ori* or *.ang*), the initial grain orientation would come from the uploaded file. The generation of the initial polycrytal follows the algorithm shown in Algorithm 1.

For the generation of the loaded polycrystal, it follows the algorithm shown in Algorithm 2. The trigger for the generation is when the user clicks the "Load Polycrystal" button. It is generated through `crystal_plasticity_module.py`, wherein the initial polycrystal data is processed through the `cpfort` module, one of the Python executable modules from Fortran generated through F2PY, which contains crystal plasticity calculations for both Taylor and Alamel models. In the generation of the initial and loaded polycrystals, hardening mechanisms and elasticity effects are currently not implemented in the crystal plasticity calculations.

The generated initial polycrystal and loaded polycrystal Python objects are saved as separate *.pkl* files. This is done through Dill [52], an external Python library, which is an extension of the built-in Pickle [53] module, which serializes and converts Python objects into a byte stream. This serialized "pickle" can then later be de-serialized and reconstructed back into Python objects, which can be used by the functions for the generation of pole figures, inverse pole figures, ODFs, yield surfaces and R-values. During the initial stages of the implementation, attempts to utilize `dcc.Store` [54], a component of the Dash framework, to store the initial and loaded polycrystal Python objects were done, however this was unsuccessful since `dcc.Store` does not support serialization of Python objects. Since `dcc.Store` stores data in the client side and utilizes the browser's local storage, its performance is greatly affected when processing huge data sizes. Thus, it was also decided to utilize "pickling" to save the initial and loaded orientations, which are NumPy arrays. The application still utilizes `dcc.Store` in other areas which do not process large data sets. The full implementation can be seen in Appendix B.3 and Appendix B.4.

Algorithm 1: Initial polycrystal generation

```
Input: Generate Polycrystal Button Click, Grain Orientation Data Source, Crystal
       Plasticity Model, Texture Component, Spread, Number of Grains, Euler Angles,
       Axis, Uploaded File
Output: Initial Polycrystal Data, Initial Orientations Data
Generate Polycrystal Button Click
datasource.create_initial_polycrystal(Grain Orientation Data Source, Crystal Plasticity
  Model, Texture Component, Spread, Number of Grains, Euler Angles, Axis, Uploaded
  File)
if Grain Orientation Data Source = Texture Components then
    if Texture Components != Fibre then
        TexTool_Driver.Generate_BTN(Texture Component, Number of Grains, Spread,
          Euler Angles)
        if Texture Components IN (Cube, Goss, Brass, Copper, S, User Input) then
            Texture Components Orientations ← GenTex.USERTEX(Number of Grains,
              Euler Angles, Spread)
        else if Texture Components = Random then
            Random Orientations ← TexTool_Funcs.Get_Random_Eulers(Number of
              Grains)
    else if Texture Components = Fibre then
        Fibre Orientations ← crystal_plasticity_module.fibre_spread(Euler Angles, Axis,
          Spread, Number of Grains)
    Orientations ← Append(Texture Components Orientations, Random Orientations,
      Fibre Orientations)
else if Grain Orientation Data Source = Upload file then
    Orientations ← Euler angles from Uploaded File
Grain Interaction ← Crystal Plasticity Model
Elasticity ← None
Crystal Structure ← FCC (111)
Hardening ← Rigid plastic
Initial Orientations Data ← Orientations
Initial Polycrystal Data ← crystal_plasticity_module.Polycrystal(Orientations, Grain
  Interaction, Elasticity, Crystal Structure, Hardening)
```

4.1.2 Generation of pole figures

Generation of pole figures is automatically triggered when the user clicks either "Generate Polycrystal" or "Load Polycrystal" button. The user can change the pole

Algorithm 2: Loaded polycrystal generation

```
Input: Load Polycrystal Button Click, Initial Polycrystal Data, Loading Condition, Global
       Velocity Gradient, Total Von Mises Strain, Number of Steps
Output: Loaded Polycrystal Data, Loaded Orientations Data
Load Polycrystal Button Click
datasource.create_loaded_polycrystal(Initial Polycrystal Data, Loading Condition, Global
  Velocity Gradient, Total Von Mises Strain, Number of Steps)
if Initial Polycrystal Data = None then
    Loaded Orientations Data ← None

Loaded Polycrystal Data ← None else
    if Loading Condition = Axisymmetric tension along z-axis then
        Loading Condition Values ← Axisymmetric tension along z-axis predefined values
    else if Loading Condition = Plane-strain tension then
        Loading Condition Values ← Plane-strain tension predefined values
    else if Loading Condition = Mixed condition then
        Loading Condition Values ← Mixed condition predefined values
    else if Loading Condition = Input Custom Values then
        Loading Condition Values ← Global Velocity Gradient
    Deformation Rate ← datasource.deformation_rate(Global velocity gradient values
      from Loading Condition Values)
    Max Time ← Total Von Mises Strain, Deformation Rate
    dt ← Max Time, Steps
    Steps ← Number of Steps
    Elasticity ← None
    Open Initial Polycrystal Data
    Loaded Orientations Data ← Initial Polycrystal
      Data.crystal_plasticity_module.ori_loaded_val
    Loaded Polycrystal Data ← Initial Polycrystal
      Data.crystal_plasticity_module.load(Loading Condition Values, dt, Steps, Elasticity)
```

figures generated depending on the selected normal direction (ND, RD or TD), projection type (stereographic or equal area), pole figure type (point projection or contour). If the user selects contour as the pole figure type, the contour levels can be changed depending on the user's preference.

The algorithm for pole figure generation is shown in Algorithm 3 and the `Plot_PF` function in `TexTool_Driver.py` is utilized to generate the pole figures, which is from the TexTool application [51]. The `Plot_PF` function calls the `Projection` function in `TexTool_Funcs.py` to generate the projections, and this initially only supported projection for ND. This code was modified to support TD and RD projections also. For contour plots, the `Plot_PF` function calls the `Points2Dens` function in `TexTool_Funcs.py`. The pole figures are created through the Matplotlib [55] plotting library and the figures are saved as *.png* images, encoded into a HTML element, and then shown in Dash. The full implementation can be seen in Appendix B.4, Appendix B.6, Appendix B.3 Code listing B.7, Appendix B.8 and Appendix B.9.

Algorithm 3: Pole Figure Generation

```
Input: Initial Polycrystal Data, Loaded Polycrystal Data, Pole figure type (Point Projection
       or Contour), Normal Direction (ND, TR or RD), Projection Type (Stereographic or
       Equal Area), Contour levels
Output: Initial Pole Figure, Loaded Pole Figure
if Initial Polycrystal Data exists then
    Open Initial Polycrystal Data
    Initial Orientations ← Orientations from Initial Polycrystal Data
    Initial Pole Figure ← TexTool_Driver.Plot_PF(Pole figure type, Projection Type,
      Normal Direction, Contour levels, Initial Orientations)
else if Initial Polycrystal Data does not exist then
    Initial Pole Figure ← None
if Loaded Polycrystal Data exists then
    Open Loaded Polycrystal Data
    Loaded Orientations ← Orientations from Loaded Polycrystal Data
    Loaded Pole Figure ← TexTool_Driver.Plot_PF(Pole figure type, Projection Type,
      Normal Direction, Contour levels, Loaded Orientations)
else if Loaded Polycrystal Data does not exist then
    Loaded Pole Figure ← None
```

4.1.3 Generation of inverse pole figures

Generation of inverse pole figures is automatically triggered when the user clicks either "Generate Polycrystal" or "Load Polycrystal" button. The user has the option to view either a static or an animated version of the inverse pole figure, and the user can also decide to view either an initial, loaded or trajectory version.

Functions from `crystal_plasticity_module.py` are used for the generation of the inverse pole figures. The existing functions in `crystal_plasticity_module.py` utilized Matplotlib, which does not support interactive graphs in Dash. Thus, new functions `plot_orientations_plotly` and `plot_orientation_plotly` were introduced to generate plotting data for plotting in Plotly [56] to be able to support interactive graphs, such as the animated inverse pole figure. The algorithm for inverse pole figure generation is shown in Algorithm 4, and the full implementation can be seen in Appendix B.4, Appendix B.6, and in Appendix B.3 Code listing B.7.

Algorithm 4: Inverse Pole Figure Generation

```
Input: Initial Polycrystal Data, Loaded Polycrystal Data, Inverse Pole Figure Type
       (Initial/Loaded/Trajectory or Animated), Plot Type (Initial, Loaded or Trajectory)
Output: Inverse Pole Figure
if Initial Polycrystal Data exists then
    datasource.generate_inversepolefigure(Initial Polycrystal Data, Loaded Polycrystal
      Data)
    if Loaded Polycrystal Data does not exist then
        Open Initial Polycrystal Data
        Initial Inverse Pole Figure Data ← Initial Polycrystal
          Data.plot_orientations_plotly(Plot Type Inverse Pole Figure)
        Inverse Pole Figure ← Generate intial figure from Initial Inverse Pole Figure Data
    else if Loaded Polycrystal exists then
        Open Loaded Polycrystal Data
        Loaded Inverse Pole Figure Data ← Loaded Polycrystal
          Data.plot_orientations_plotly(Plot Type Inverse Pole Figure)
    if Animated Inverse Pole Figure is True then
        Inverse Pole Figure ← Generate animated figure from Loaded Inverse Pole Figure
          Data
    else if Animated Inverse Pole Figure is False then
        Generate empty Inverse Pole Figure
        if Initial/Loaded/Trajectory Inverse Pole Figure has Initial then
            Inverse Pole Figure ← Generate initial figure from Loaded Inverse Pole
              Figure Data
        else if Initial/Loaded/Trajectory Inverse Pole Figure has Loaded then
            Inverse Pole Figure ← Generate loaded figure from Loaded Inverse Pole
              Figure Data
        else if Initial/Loaded/Trajectory Inverse Pole Figure has Trajectory then
            Inverse Pole Figure ← Generate trajectory figure from Loaded Inverse Pole
              Figure Data
else if Initial Polycrystal Data does not exist then
    Inverse Pole Figure ← None
```

4.1.4 Generation of ODF figures

The generation of the ODF figures can be triggered by clicking "Generate ODF" in the GUI. The contour levels of the figure can be controlled by the user through the "ODF Levels" field.

Generation of the ODF figures are done through the existing functions `Orientations` and `ODF` in `odflib.py`. Similar to the pole figures, Matplotlib is utilized for the generation of the ODF figures, which are saved as *.png* files, then encoded to a HTML element, and then shown in the GUI. The saving of the Matplotlib figures as *.png* images are done through the newly added function `save_plotly` in `odflib.py`. The algorithm for inverse pole figure generation is shown in Algorithm 5, and the full implementation can be seen in Appendix B.4 and in Appendix B.3 Code listing B.7.

Algorithm 5: ODF Figure Generation

```
Input: Initial Polycrystal Data, Loaded Polycrystal Data, ODF Levels
Output: Initial ODF Figure, Loaded ODF Figure
if Initial Polycrystal Data exists then
    datasource.generate_odf(ODF Levels, Initial Polycrystal Data, Loaded Polycrystal
      Data)
    Open Initial Polycrystal Data
    ODF Initial Data ← Initial Polycrystal Data.plot_orientations_plotly(Plot Type ODF)
    Initial Orientations ← odflib.Orientations(ODF Initial Data)
    Matplotlib Inital ODF Figure = odflib.ODF(Initial Orientations)
    if Loaded Polycrystal Data does not exist then
        Initial ODF Figure ← Matplotlib Inital ODF Figure.save_plotly(Plot Type ODF)
        Loaded ODF Figure ← None
    else if Loaded Polycrystal exists then
        Open Loaded Polycrystal Data
        ODF Loaded Data ← Loaded Polycrystal Data.plot_orientations_plotly(Plot Type
          ODF)
        Loaded Orientations ← odflib.Orientations(ODF Loaded Data)
        Matplotlib ODF Figure = odflib.ODF(Loaded Orientations)
        Initial ODF Figure ← Matplotlib Inital ODF Figure.save_plotly(Plot Type ODF)
        Loaded ODF Figure ← Matplotlib Loaded ODF Figure.save_plotly(Plot Type
          ODF)
else if Initial Polycrystal Data does not exist then
    Initial ODF Figure ← None
    Loaded ODF Figure ← None
```

4.1.5 Generation of yield surface

The generation of the two-dimensional yield surfaces is triggered by clicking "Generate Yield Surface" in the GUI. The user can select to view the initial and/or loaded figures, and the user can also select the x-axis and y-axis for the yield surface. The yield surface figure is created through the `yield_locus` function under `Polycrystal` class in `crystal_plasticity_module.py`, and changes were done to convert the figures from Matplotlib to Plotly. The algorithm followed for this is shown in Algorithm 6, and the full implementation can be seen seen in Appendix B.4 and in Appendix B.3 Code listing B.7.

4.1.6 Generation of R-values

The algorithm to generate the R-values and yield stress graphs can be seen in Algorithm 7. The trigger to generate these graphs is through the button "Generate R-values" in the GUI. The user can either select to view initial or loaded figures, and the user can also select the normal axis for the R-values. The function `getR-values` under `Polycrystal` class in `crystal_plasticity_module.py` is utilized for the generation. Similar to the other figures in the application, changes were done to convert the figures from Matplotlib to Plotly. The full implementation can be seen seen in Appendix B.4 and in Appendix B.3 Code listing B.7.

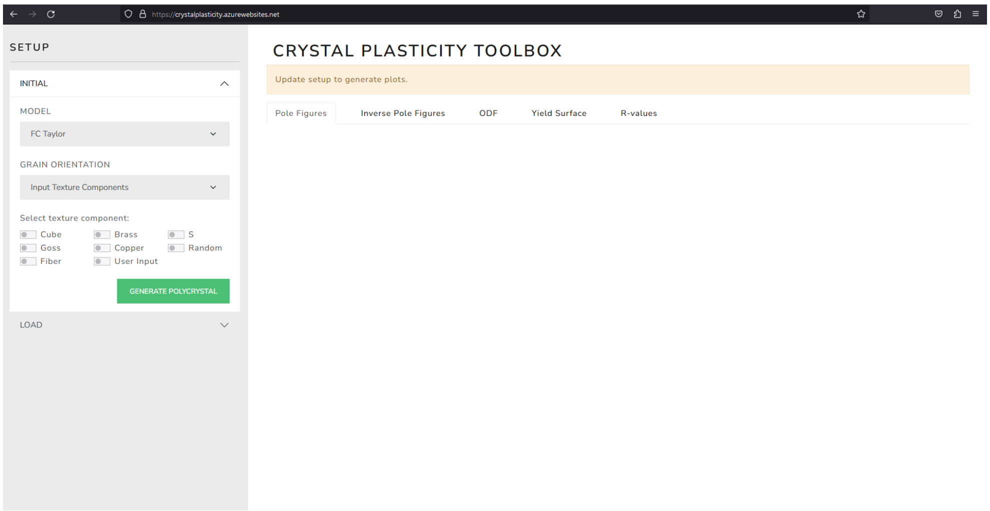
Algorithm 6: Yield Surface Figure Generation

```
Input: Initial Polycrystal Data, Loaded Polycrystal Data, Plot Axes, Initial or Loaded Yield
       Surface
Output: Yield Surface Figure
if Initial Polycrystal Data exists then
    datasource.generate_ys(Plot Axes, Initial Polycrystal Data, Loaded Polycrystal Data)
    if Loaded Polycrystal Data does not exist then
        Open Initial Polycrystal Data
        Initial Yield Surface Data ← Initial Polycrystal Data.yield_locus(Plot Axes)
        Yield Surface Figure ← Generate initial figure from Initial Yield Surface Data
    else if Loaded Polycrystal exists then
        Open Initial Polycrystal Data
        Open Loaded Polycrystal Data
        if Initial or Loaded Yield Surface has Initial then
            Initial Yield Surface Data ← Initial Polycrystal Data.yield_locus(Plot Axes)
            Initial Yield Surface Figure ← Generate initial figure from Initial Yield
              Surface Data
            Yield Surface Figure ← Add Initial Yield Surface Figure to figure
        if Initial or Loaded Yield Surface has Loaded then
            Loaded Yield Surface Data ← Loaded Polycrystal Data.yield_locus(Plot Axes)

            Loaded Yield Surface Figure ← Generate loaded figure from Loaded Yield
              Surface Data
            Yield Surface Figure ← Add Loaded Yield Surface Figure to figure
else if Initial Polycrystal Data does not exist then
    Yield Surface Figure ← None
```

Algorithm 7: R-values Figure Generation

```
Input: Initial Polycrystal Data, Loaded Polycrystal Data, Normal Axis, Initial or Loaded R-values
Output: R-values Figure
if Initial Polycrystal Data exists then
    datasource.generate_rv(Normal Axis, Initial Polycrystal Data, Loaded Polycrystal Data)
    if Loaded Polycrystal Data does not exist then
        Angles ← 0 to 90 degrees with increments of 5 degrees, converted to radians
        Open Initial Polycrystal Data
        for Angle in Angles do
            Tensile Axis ← From Normal Axis and Angle
            Initial R-values Data for Angle ← Initial Polycrystal Data.crystal_plasticity_module.getRvalues(Normal Axis, Tensile Axis)
            Initial R-values Data ← Append(Initial R-values Data for Angle)
        R-values Figure ← Generate initial figure from Initial R-values Data and Angles
    else if Loaded Polycrystal exists then
        Angles ← 0 to 90 degrees with increments of 5 degrees, converted to radians
        if Initial or Loaded R-values = Initial then
            Open Initial Polycrystal Data
            for Angle in Angles do
                Tensile Axis ← From Normal Axis and Angle
                Initial R-values Data for Angle ← Initial Polycrystal Data.crystal_plasticity_module.getRvalues(Normal Axis, Tensile Axis)
                Initial R-values Data ← Append(Initial R-values Data for Angle)
            R-values Figure ← Generate initial figure from Initial R-values Data and Angles
        else if Initial or Loaded R-values = Loaded then
            Open Loaded Polycrystal Data
            for Angle in Angles do
                Tensile Axis ← From Normal Axis and Angle
                Loaded R-values Data for Angle ← Loaded Polycrystal Data.crystal_plasticity_module.getRvalues(Normal Axis, Tensile Axis)
                Loaded R-values Data ← Append(Loaded R-values Data for Angle)
            R-values Figure ← Generate loaded figure from Loaded R-values Data and Angles
else if Initial Polycrystal Data does not exist then
    R-values Figure ← None
```

4.2 Crystal plasticity toolbox overview



MODEL: FC Taylor

GRAIN ORIENTATION: Input Texture Components

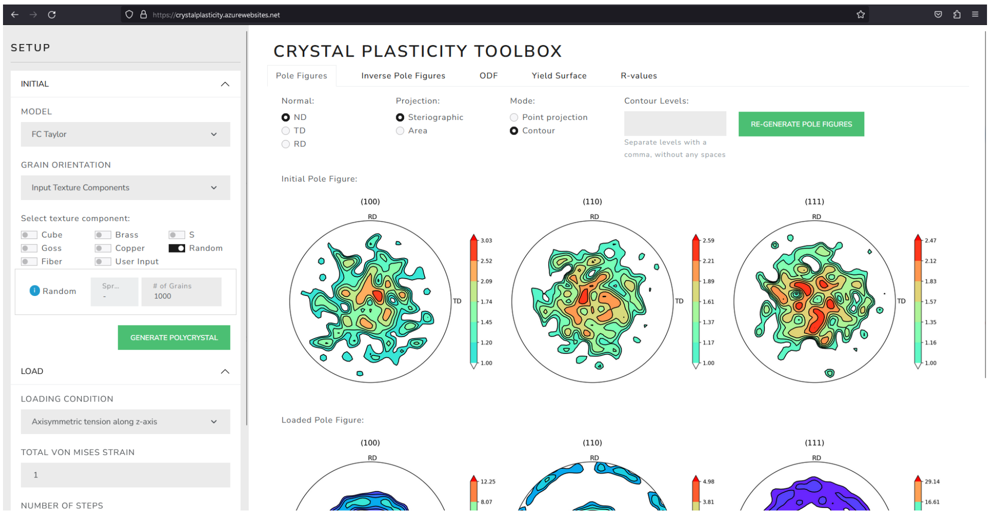
Update setup to generate plots.

Figure 4.1: Initial load view of the cloud-based crystal plasticity toolbox.

The GUI of the application is divided into two areas: the sidebar and the main view for the generated figures. The sidebar is located on the left side of the GUI, and this is where the user can configure the setup for the initial and loaded polycrystal. The main view for the generated figures is located on the right side of the GUI, wherein the different figures that can be generated by the toolbox are separated into different tabs: Pole figures, Inverse Pole Figures, ODF, Yield Surface, and R-values (see Figure 4.1).

The initial setup of the polycrystal is done in the left sidebar, wherein the user will be able to select either "FC Taylor" or "Alamel" in the "Model" dropdown field. The user can also select the source of the grain orientations in the "Grain Orientation" dropdown field. When "Input Texture Components" is selected as the grain orientation source, the user will be able to select the texture components for the polycrystal. Depending on the selected texture component radio buttons, the user will be able to define other details for the initial polycrystal. If the radio buttons "Cube", "Brass", "S", "Goss" or "Copper" is selected, then the user will be able to enter the spread and the number of grains. If the radio button "Random" is selected, the user will be able to enter the number of grains, but not the spread. If the radio button "Fibre" is selected, the user will be able to enter the spread, number of grains, Euler Angles (Bunge's notation in degrees for the three angles of rotation: φ_1, Φ, and φ_2), and axis (x,y, and z). And if the radio button "User Input" is selected, the user will be able to enter the spread, number of grains, and Euler Angles. The user can upload a file as the source of the grain orientations by selecting "Upload file" as the value in "Grain Orientation field", which can be useful for analysis of material texture from external sources. Currently, either a *.ang* or *.ori* file is supported for this functionality. If a *.ang* file is uploaded, the threshold for the values to be taken from the file can be defined. By clicking the "Generate Polycrystal" button, the user will be able to trigger the generation of the initial polycrystal and also automatically generate the initial pole figure and inverse pole figure. The user will be able to generate the initial ODF, yield surface and R-values by navigating to their respective tabs, optionally changing some settings, and then clicking their respective generate buttons.

The initial polycrystal can be subjected to a deformation to create the loaded polycrystal. There is an option for the user to select a pre-defined loading condition in the "Loading Condition" dropdown field, which can either be "Axisymmetric tension along z-axis", "Plane-strain compression", or "Mixed (strain+stress) boundary condition". The user can also select "Input Custom Values" to enter custom values for the global velocity gradient (L). To further define the deformation behaviour, the user can control the values for the "Total Von Mises Strain" and "Number of Steps". To apply the loading setup to the polycrystal, the user can click the "Load Polycrystal" button, which will also trigger the generation of the loaded versions of the pole figure and inverse pole figure. Similar to the initial polycrystal, the loaded ODF, yield surface and R-values can be generated by navigating to their respective tabs, optionally changing some settings, and then clicking their respective generate buttons.



https://crystalplasticity.azurewebsites.net

SETUP

INITIAL

MODEL: FC Taylor

GRAIN ORIENTATION: Input Texture Components

Select texture component:
☐ Cube ☐ Brass ☐ S
☐ Goss ☐ Copper ☑ Random
☐ Fiber ☐ User Input

Random Spr... - # of Grains 1000

LOAD

LOADING CONDITION: Axisymmetric tension along z-axis

TOTAL VON MISES STRAIN: 1

NUMBER OF STEPS

CRYSTAL PLASTICITY TOOLBOX

Normal: ● ND ○ TD ○ RD

Projection: ● Steriographic ○ Area

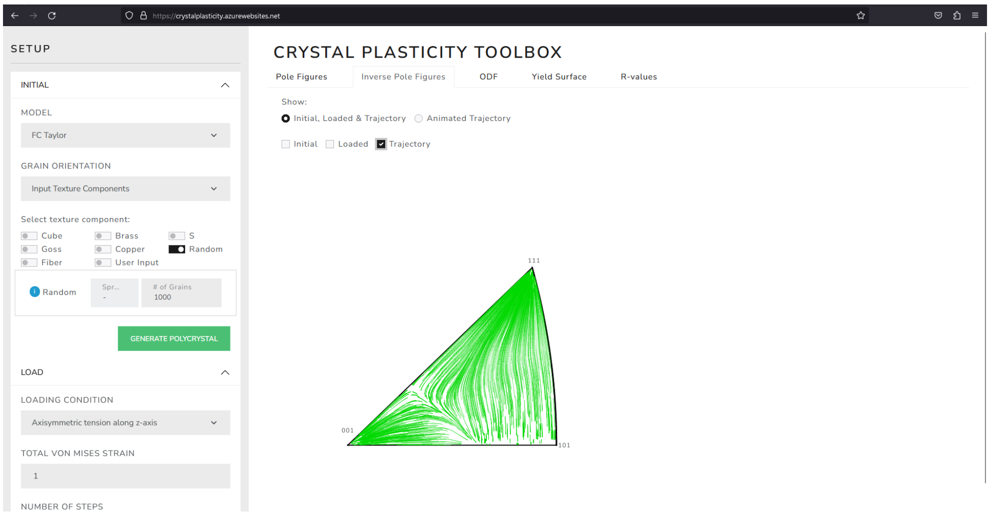
Mode: ○ Point projection ● Contour

Contour Levels:

Separate levels with a comma, without any spaces

Figure 4.2: Pole figure tab of the cloud-based crystal plasticity toolbox.

The first tab in the main view is the "Pole Figures" tab, where the initial and loaded pole figures will be shown (see Figure 4.2). The user has the option to change a few settings for the pole figures generated, such as the normal direction (ND, TD or RD), the projection mode (stereographic or equal area), and the pole figure type (point or contour). If the user selects "Contour" as the pole figure type, the "Contour Levels" field will be shown and the user will be able to define specific levels for the contour plot. The pole figures shown will be for the FCC structure and for the planes (100), (110) and (111).



MODEL
FC Taylor

GRAIN ORIENTATION
Input Texture Components

Select texture component:
○ Cube ○ Brass ○ S
○ Goss ○ Copper ● Random
○ Fiber ○ User Input

Random Spr... - # of Grains 1000

LOADING CONDITION
Axisymmetric tension along z-axis

TOTAL VON MISES STRAIN
1

NUMBER OF STEPS

Show:
● Initial, Loaded & Trajectory ○ Animated Trajectory

☐ Initial ☐ Loaded ☑ Trajectory

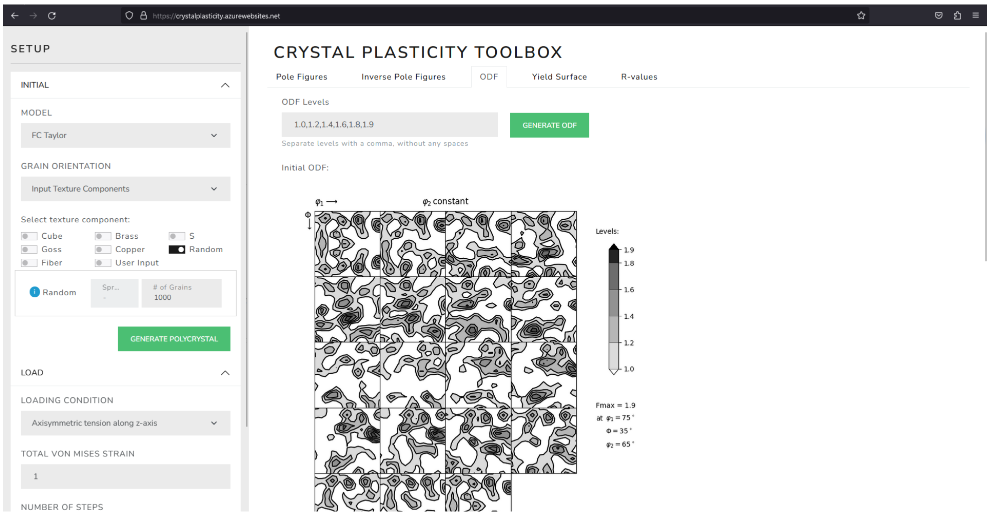
Figure 4.3: Inverse pole figure tab of the cloud-based crystal plasticity toolbox showing trajectories of single grain orientations during axisymmetric tension are shown, starting from the random texture.

The second tab is the "Inverse Pole Figures" tab, where the initial, loaded or trajectory version of the inverse pole figure is shown (see Figure 4.3). The user can select to show a static or animated version of the figure. In the animated version, the movement or the rotation of the grains can be visualized by showing the points moving to a preferred direction.

The third tab is the "ODF" tab, which shows the ODF figures (see Figure 4.4). In this tab, the figures are not automatically generated by clicking the "Generate Polycrystal" or "Load Polycrystal" buttons. The user has the option to change the default contour levels (2, 4, 8, 12, 16, 20, 25, 30, 35), and then click "Generate ODF" button to view the figures. If the polycrystal is not loaded, this tab will only show the initial ODF figures. And if the polycrystal is loaded, this tab will show both the initial and loaded ODF figures.

The fourth tab is the "Yield Surface" tab, which shows the two-dimension yield surface figures (see Figure 4.5). In this tab, the figures are also not automatically generated by clicking the "Generate Polycrystal" or "Load Polycrystal" buttons. The user can select the stress components to be plotted on the x and y-axis for the yield surface figure, then click the "Generate Yield Surface" button to generate the figure. When a high number of grains and deformation is entered in the setup, the yield surface generation could take up a lot of machine resources and could take a long time to run, thus, controls are added in the GUI so that the user has the option to generate either only the initial version, or only the loaded version, or both at the same time.

The fifth and last tab is the "R-values" tab, which shows the R-values and Yield Stress graphs (see Figure 4.6). Similar to the previous two tabs, the figures here are also not automatically generated by clicking the "Generate Polycrystal"



https://crystalplasticity.azurewebsites.net

CRYSTAL PLASTICITY TOOLBOX

SETUP

INITIAL

MODEL: FC Taylor

GRAIN ORIENTATION: Input Texture Components

Select texture component:

☐ Cube ☐ Brass ☐ S
☐ Goss ☐ Copper ☑ Random
☐ Fiber ☐ User Input

Random Spr... - # of Grains 1000

LOAD

LOADING CONDITION: Axisymmetric tension along z-axis

TOTAL VON MISES STRAIN: 1

NUMBER OF STEPS

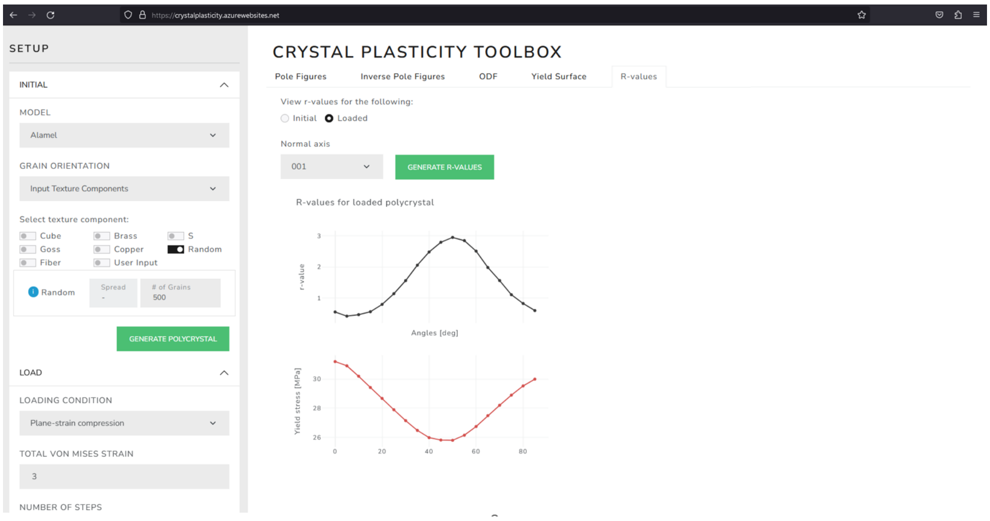
ODF Levels: 1.0,1.2,1.4,1.6,1.8,1.9

Separate levels with a comma, without any spaces

Initial ODF:

Figure 4.4: ODF figure tab of the cloud-based crystal plasticity toolbox showing ODF for a random texture.

or "Load Polycrystal" buttons. The user has the option to change the normal axis for the generation of the R-values, then click the "Generate R-values" button to generate the figure. If the polycrystal is loaded, the user will also have the option to view either the initial or loaded version.



https://crystalplasticity.azurewebsites.net

CRYSTAL PLASTICITY TOOLBOX

SETUP

INITIAL

MODEL: Alamel

GRAIN ORIENTATION: Input Texture Components

Select texture component:

☐ Cube ☐ Brass ☐ S
☐ Goss ☐ Copper ☑ Random
☐ Fiber ☐ User Input

Random Spread - # of Grains 500

LOAD

LOADING CONDITION: Plane-strain compression

TOTAL VON MISES STRAIN: 3

NUMBER OF STEPS

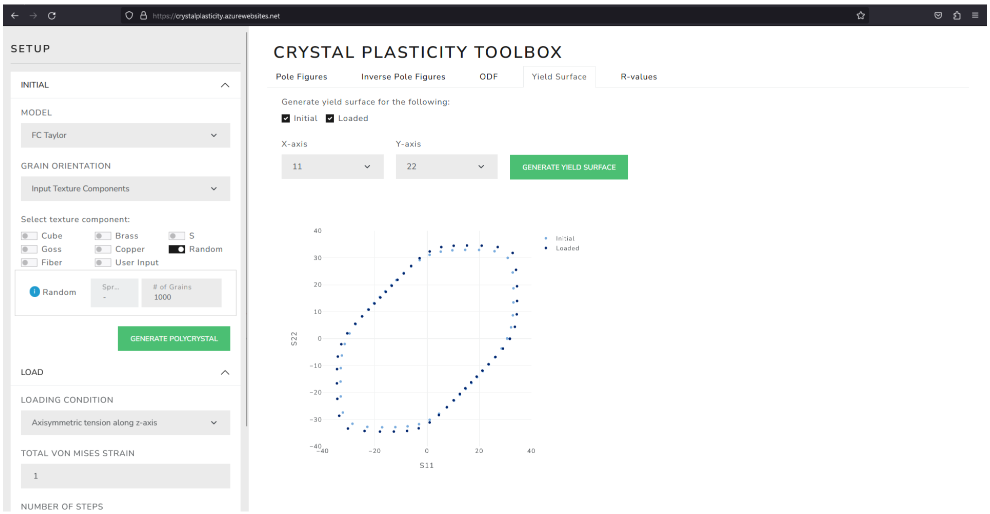
View r-values for the following:

○ Initial ● Loaded

Normal axis: 001

R-values for loaded polycrystal

Figure 4.6: R-values figure tab of the cloud-based crystal plasticity toolbox showing R-values for normal axis [001] for 500 random grains subjected to Plane-strain compression with Total Von Mises Strain of 3.


https://crystalplasticity.azurewebsites.net

SETUP

INITIAL

MODEL
FC Taylor

GRAIN ORIENTATION
Input Texture Components

Select texture component:
☐ Cube ☐ Brass ☐ S
☐ Goss ☐ Copper ☑ Random
☐ Fiber ☐ User Input

Random Spr... - # of Grains 1000

LOAD

LOADING CONDITION
Axisymmetric tension along z-axis

TOTAL VON MISES STRAIN
1

NUMBER OF STEPS

CRYSTAL PLASTICITY TOOLBOX

Generate yield surface for the following:
☑ Initial ☑ Loaded

X-axis
11

Y-axis
22

Figure 4.5: Yield surface figure tab of the cloud-based crystal plasticity toolbox showing yield surface for a random texture (initial) and the texture after axisymmetric tension (loaded).

4.3 Generated figures

The figures generated for the following simulation are shown in this section:

- Grain Orientation: Random
- Number of grains: 1000
- Model: One for Taylor and one for Alamel
- Loading condition: Run for both axisymmetric tension along z-axis and plane-strain compression, with Total Von Mised Strain 0.5, 1, 2, and 5

Figure 4.7 and Figure 4.8 show the evolution of the pole figures generated through stereographic projection for FCC planes (100), (110) and (111), subjected to axisymmetric tension along z-axis using Taylor model. As the Total Von Mises strain is increased, grains rotate to a fibre texture, as shown in both point and contour pole figures. Figure 4.9 and Figure 4.10 are similar pole figures, but for plane-strain compression and using Alamel model.

The evolution of the inverse pole figures as deformation is increased can be seen in Figure 4.11 and Figure 4.12, using Taylor and Alamel model, respectively. For loading with axisymmetric tension along z-axis, the grains rotate to an orientation of either [111] or [011] direction. For loading with plane-strain compression, the grains rotate towards stable orientations along the so-called beta-fibre.

ODF figures for axisymmetric tension along z-axis using Taylor Model can be seen in Figure 4.13, while ODF figures for plane-strain compression using Alamel model can be seen in Figure 4.13. Figure 4.15 shows the evolution of two-dimensional yield surfaces, and Figure 4.16 shows the evolution of R-values and yield stress, wherein both are for plane-strain compression using Alamel model.

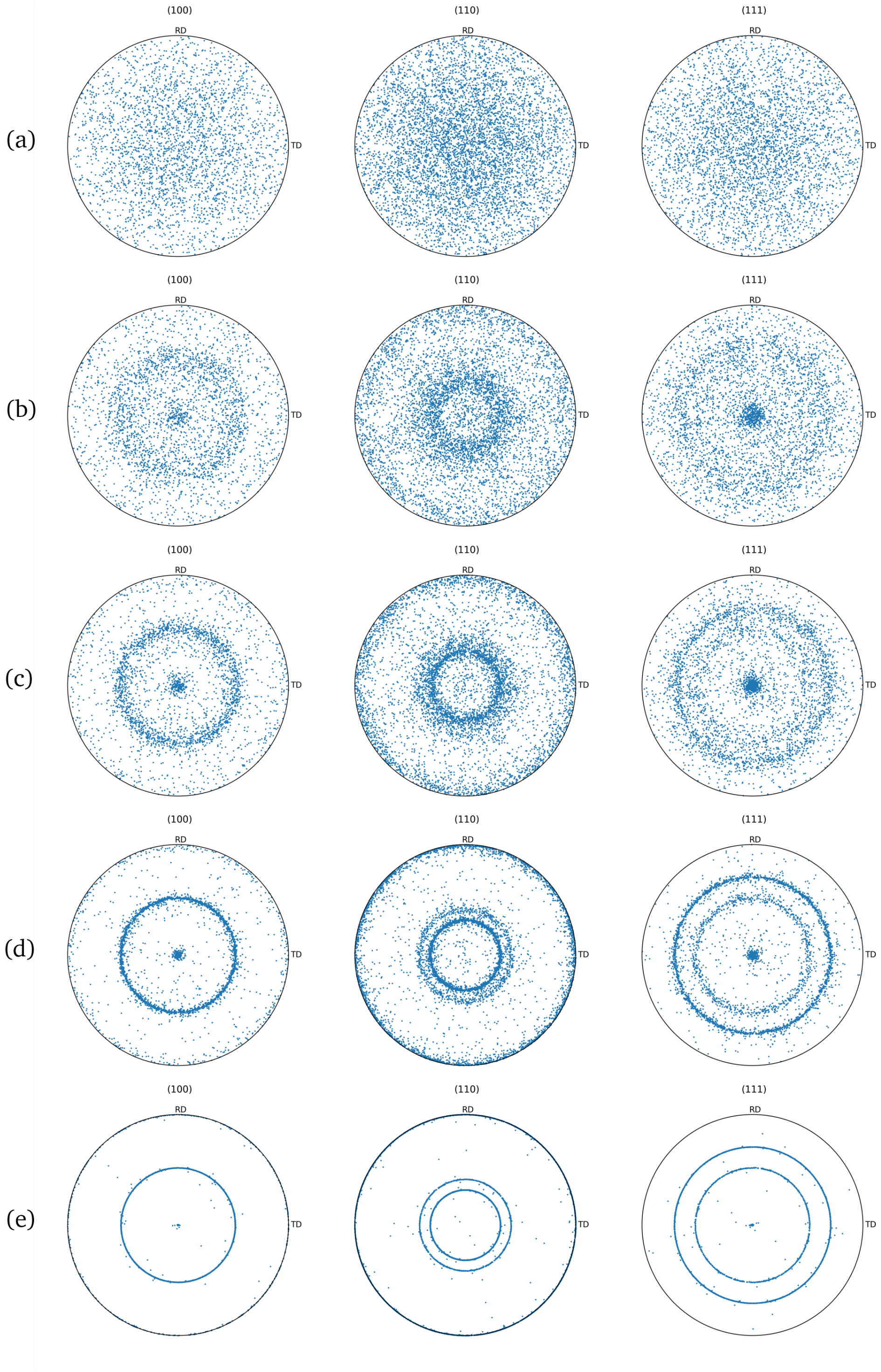


Figure 4.7: Pole figures generated using stereographic point projection for FCC structure with 1000 random grains subjected to axisymmetric tension along z-axis using Taylor Model: (a) Initial polycrystal; not loaded, (b) Total Von Mises Strain of 0.5, (c) Total Von Mises Strain of 1, (d) Total Von Mises Strain of 2, (e) Total Von Mises Strain of 5.

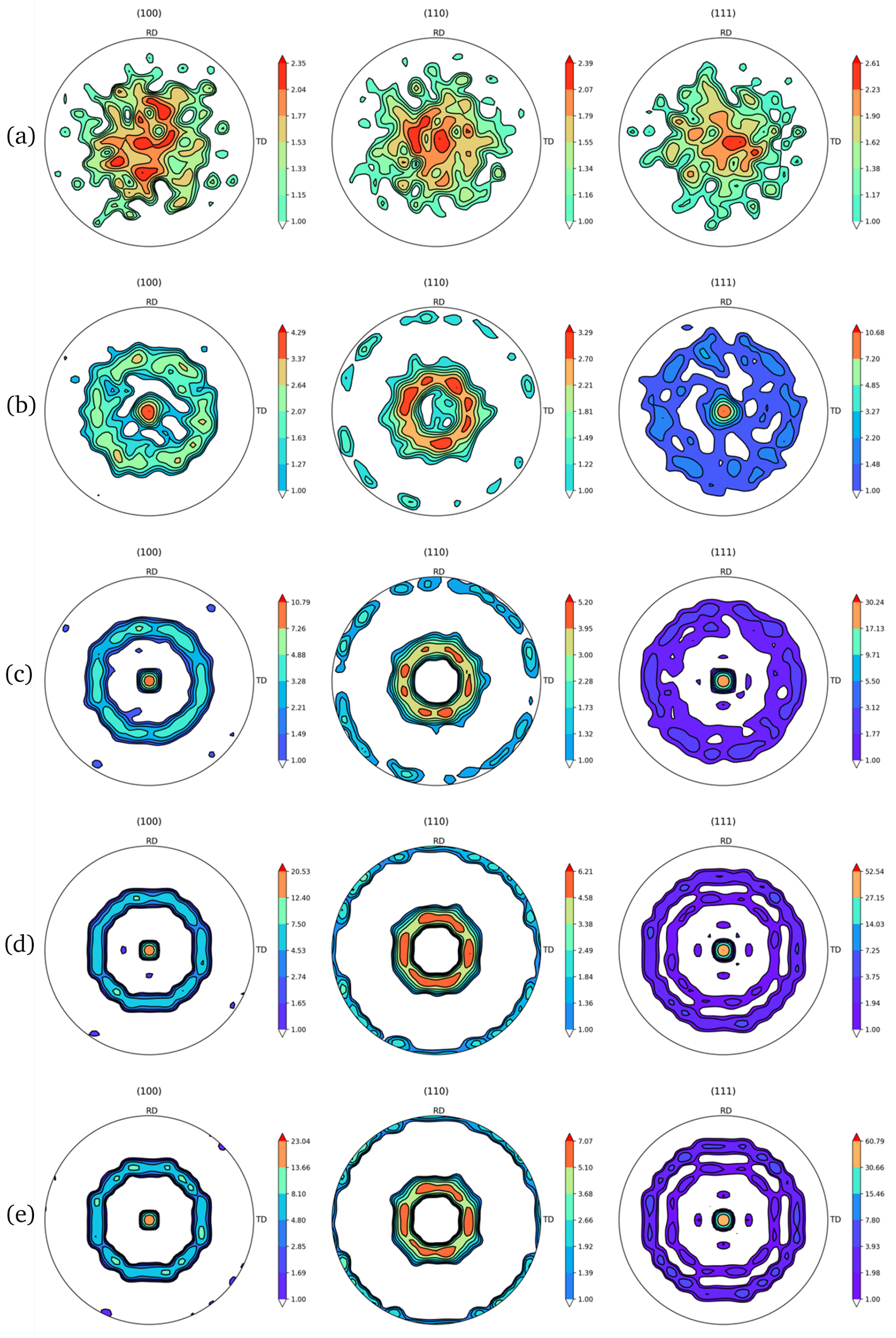


Figure 4.8: Pole figures generated using stereographic projection in contour plots for FCC structure with 1000 random grains subjected to axisymmetric tension along z-axis using Taylor Model: (a) Initial polycrystal; not loaded, (b) Total Von Mises Strain of 0.5, (c) Total Von Mises Strain of 1, (d) Total Von Mises Strain of 2, (e) Total Von Mises Strain of 5.

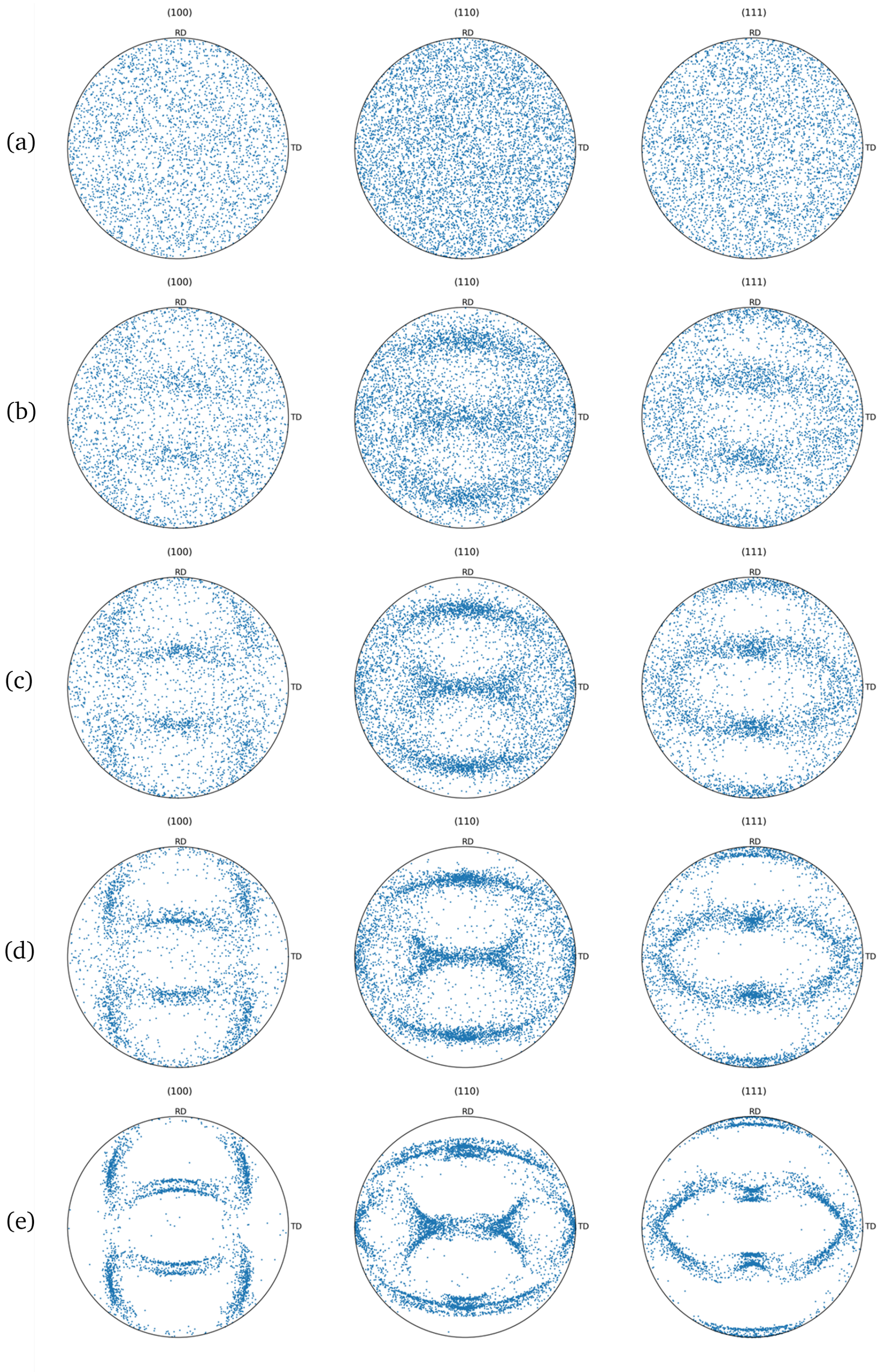


Figure 4.9: Pole figures generated using area point projection for FCC structure with 1000 random grains subjected to plane-strain compression using Alamel Model: (a) Initial polycrystal; not loaded, (b) Total Von Mises Strain of 0.5, (c) Total Von Mises Strain of 1, (d) Total Von Mises Strain of 2, (e) Total Von Mises Strain of 5.

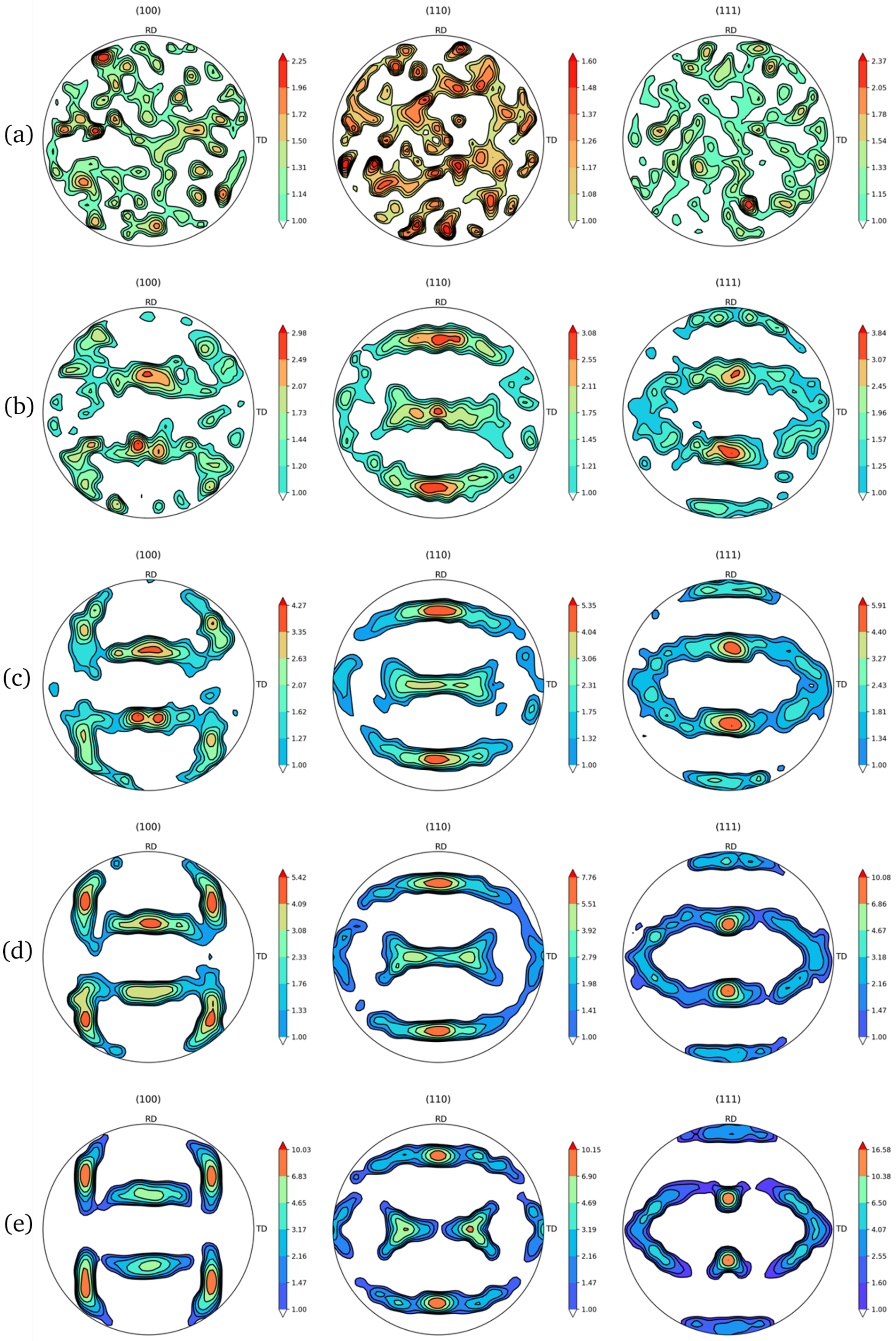


Figure 4.10: Pole figures generated using area projection in contour plots for FCC structure with 1000 random grains subjected to plane-strain compression using Alamel Model: (a) Initial polycrystal; not loaded, (b) Total Von Mises Strain of 0.5, (c) Total Von Mises Strain of 1, (d) Total Von Mises Strain of 2, (e) Total Von Mises Strain of 5.

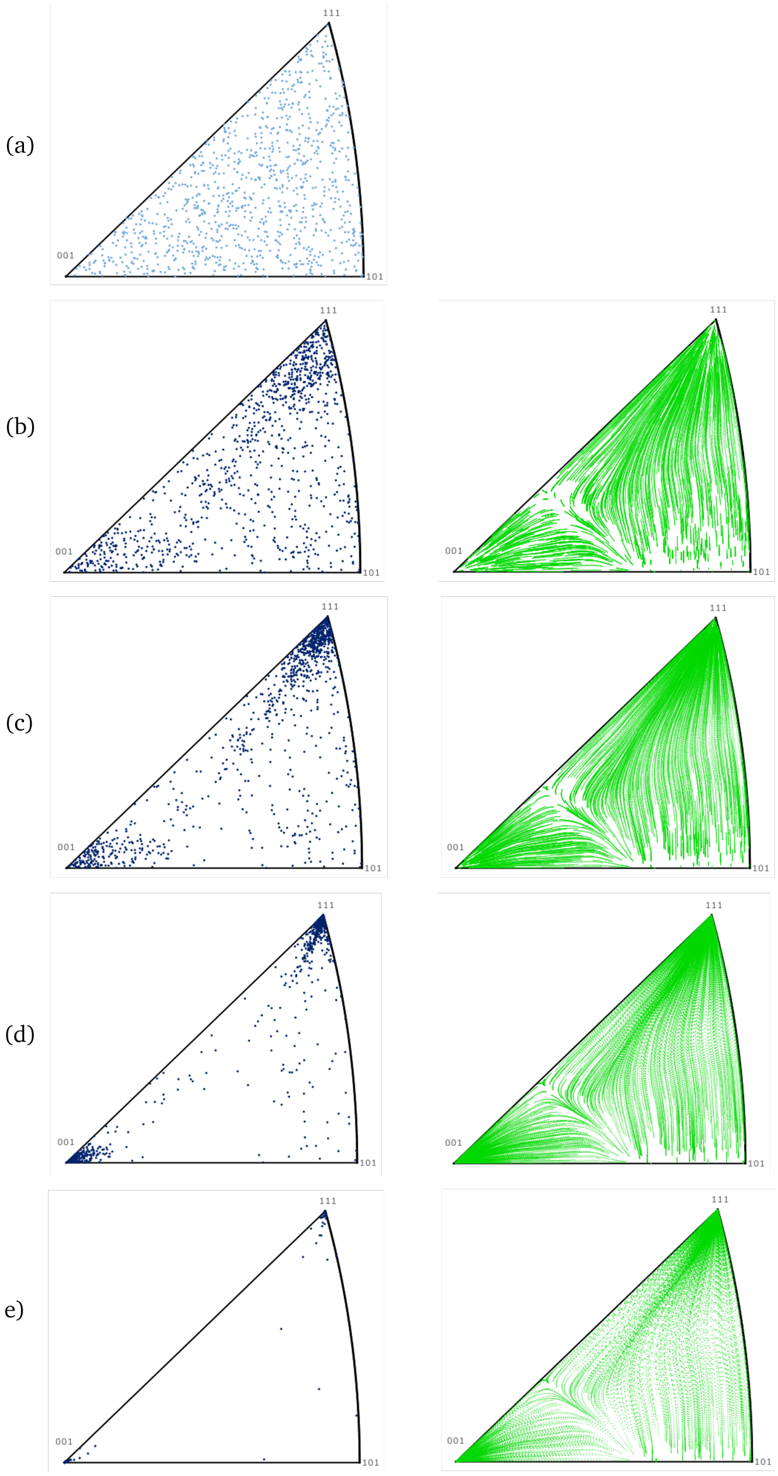


Figure 4.11: Inverse pole figures generated for FCC structure with 1000 random grains subjected to axisymmetric tension along z-axis using Taylor Model, with the left figures showing the loaded version and the right figures showing the trajectories: (a) Initial polycrystal; not loaded, (b) Total Von Mises Strain of 0.5, (c) Total Von Mises Strain of 1, (d) Total Von Mises Strain of 2, (e) Total Von Mises Strain of 5.

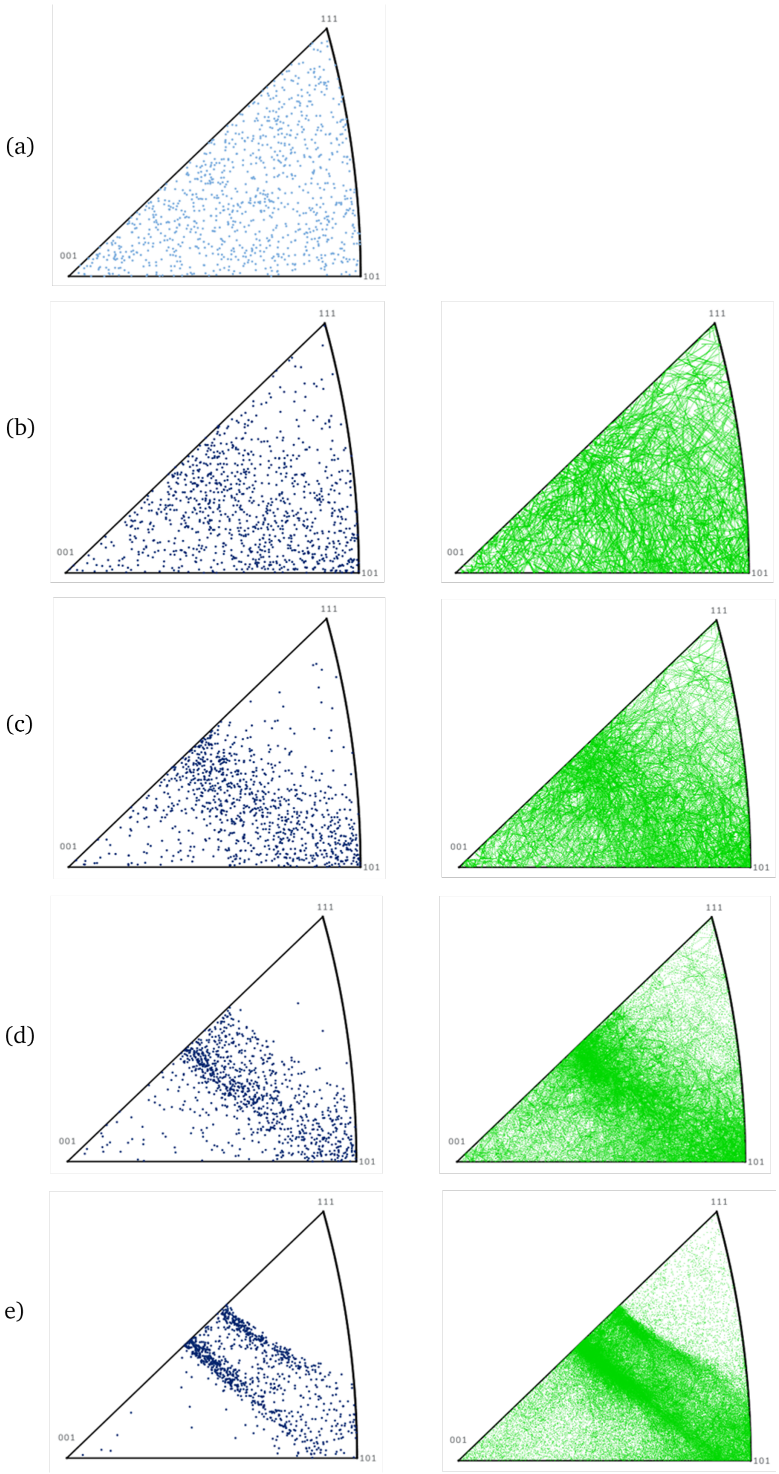


Figure 4.12: Inverse pole figures generated for FCC structure with 1000 random grains subjected to plane-strain compression using Alamel Model, with the left figures showing the loaded version and the right figures showing the trajectories: (a) Initial polycrystal; not loaded, (b) Total Von Mises Strain of 0.5, (c) Total Von Mises Strain of 1, (d) Total Von Mises Strain of 2, (e) Total Von Mises Strain of 5.

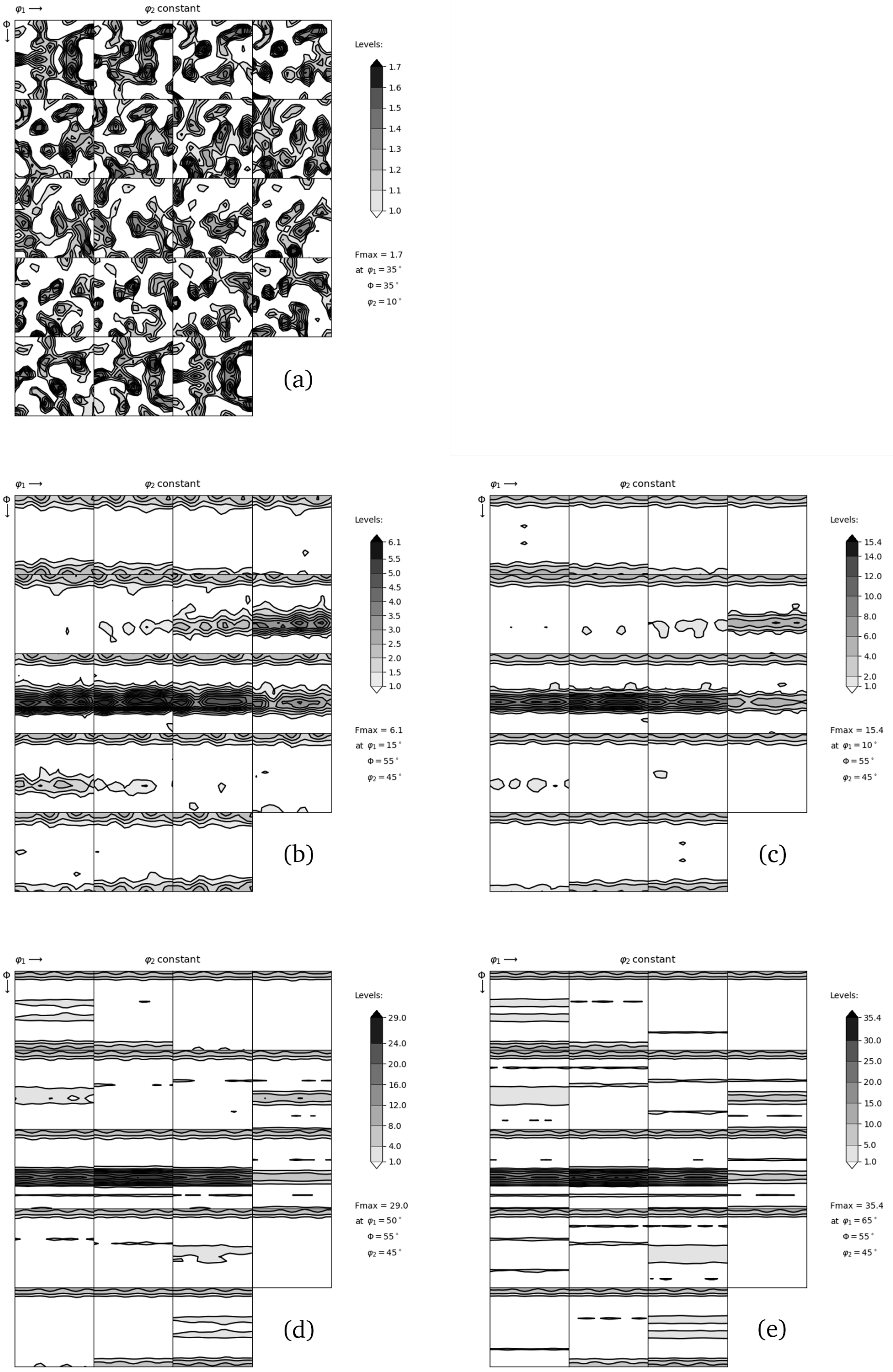


Figure 4.13: ODF figures generated for FCC structure with 1000 random grains subjected to axisymmetric tension along z-axis using Taylor Model: (a) Initial polycrystal; not loaded, (b) Total Von Mises Strain of 0.5, (c) Total Von Mises Strain of 1, (d) Total Von Mises Strain of 2, (e) Total Von Mises Strain of 5.

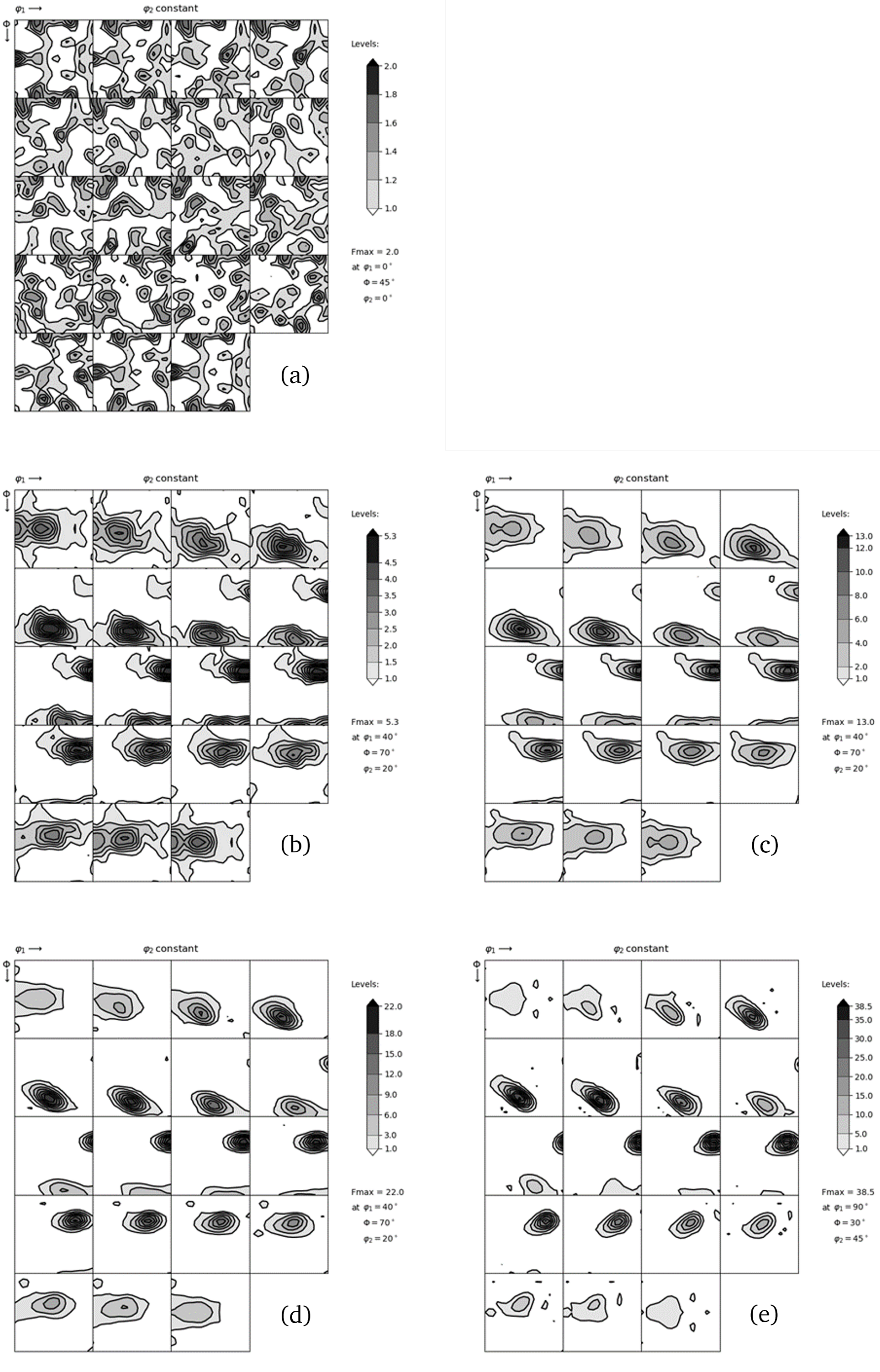


Figure 4.14: ODF figures generated for FCC structure with 1000 random grains subjected to plane-strain compression using Alamel Model: (a) Initial polycrystal; not loaded, (b) Total Von Mises Strain of 0.5, (c) Total Von Mises Strain of 1, (d) Total Von Mises Strain of 2, (e) Total Von Mises Strain of 5.

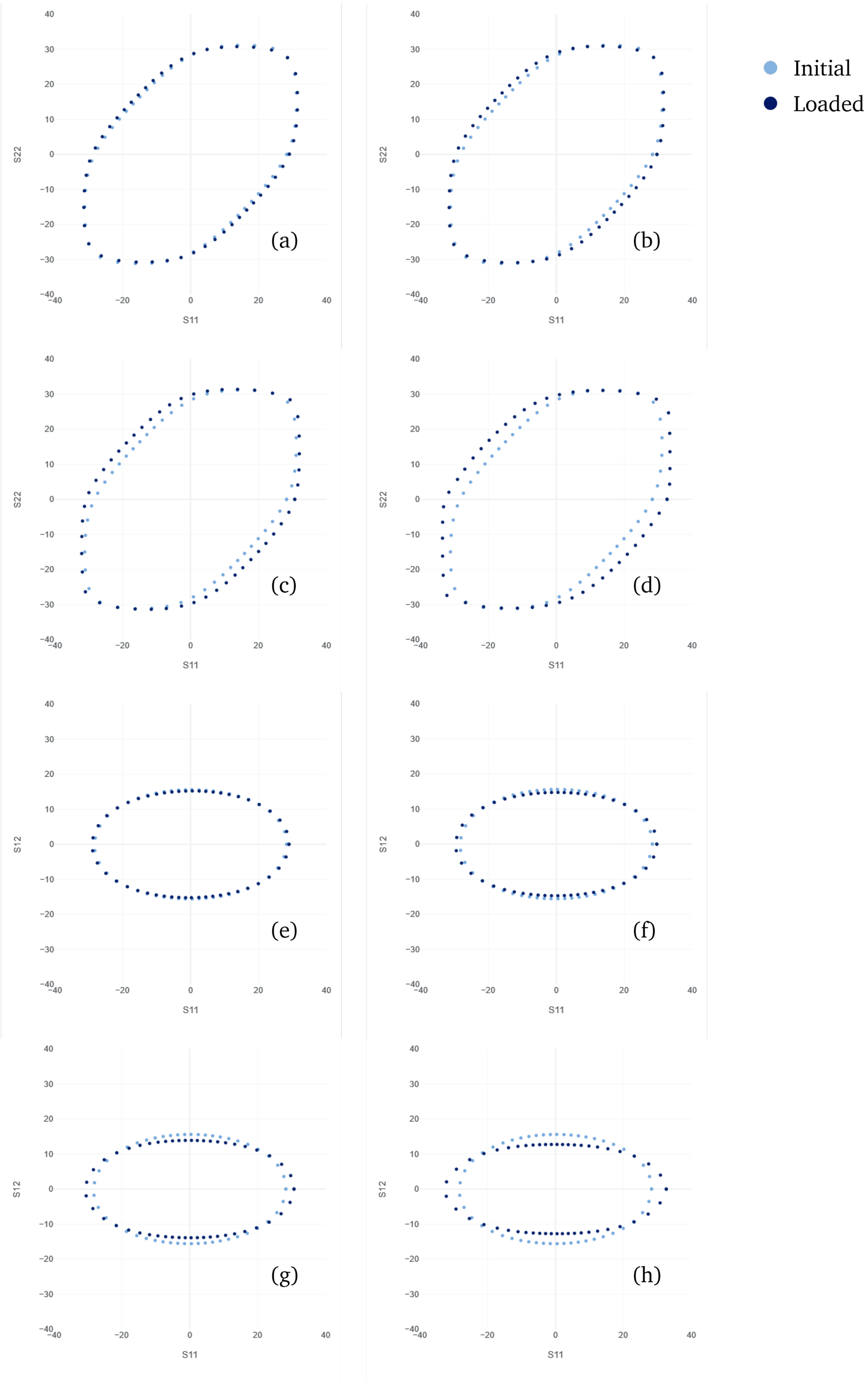


Figure 4.15: 2-dimension yield surfaces generated for FCC structure with 1000 random grains subjected to axisymmetric tension along z-axis for 11-22 axes using Taylor Model: (a) Total Von Mises Strain of 0.5, (b) Total Von Mises Strain of 1, (c) Total Von Mises Strain of 2, (d) Total Von Mises Strain of 5; and for 11-12 axes: (e) Total Von Mises Strain of 0.5, (f) Total Von Mises Strain of 1, (g) Total Von Mises Strain of 2, (h) Total Von Mises Strain of 5.

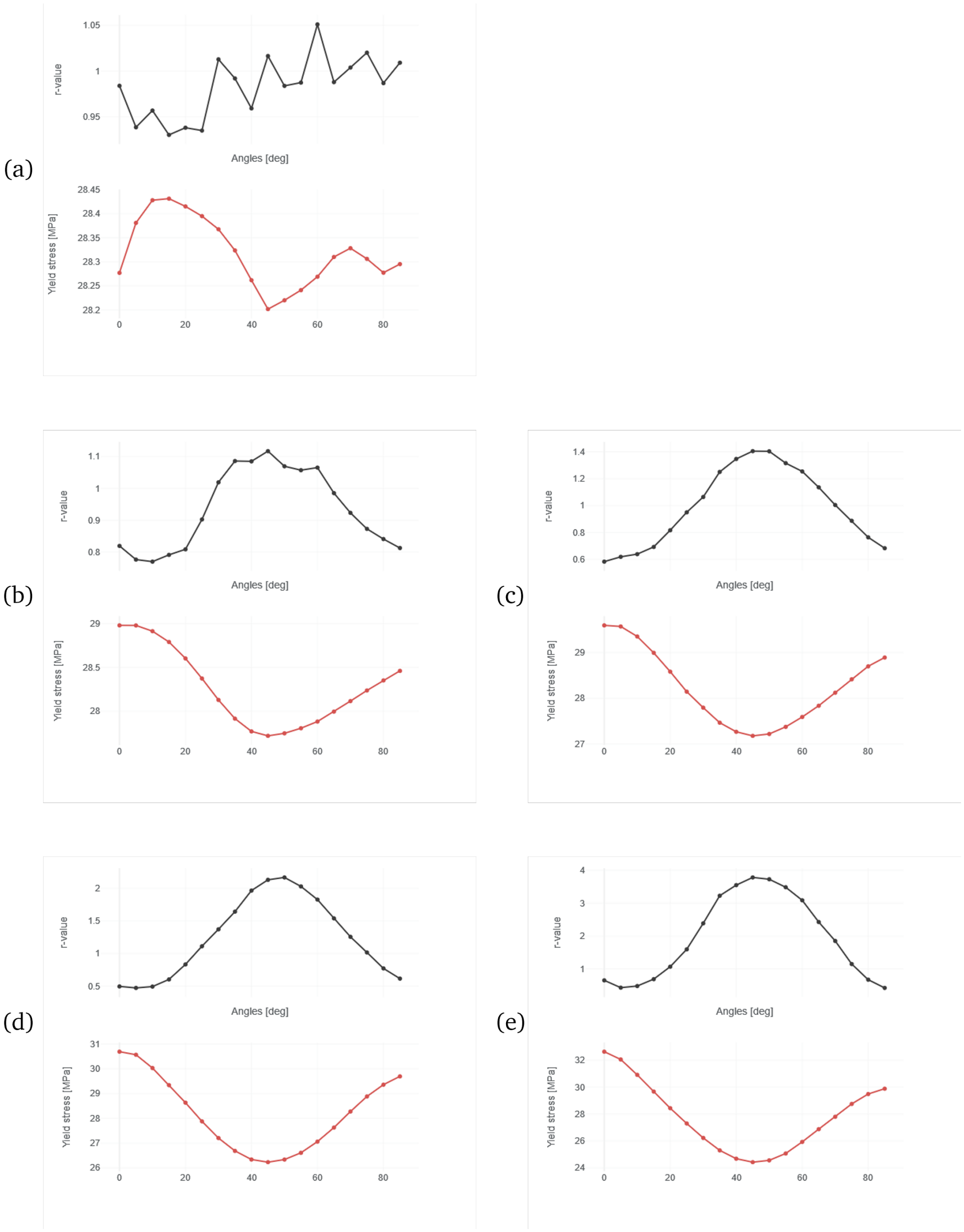


Figure 4.16: R-value and yield stress figures generated for FCC structure with 1000 random grains subjected to plane-strain compression with normal axis [001] using Alamel Model: (a) Initial polycrystal; not loaded, (b) Total Von Mises Strain of 0.5, (c) Total Von Mises Strain of 1, (d) Total Von Mises Strain of 2, (e) Total Von Mises Strain of 5.

4.4 Challenges during implementation

A lot of challenges were encountered during the implementation and deployment of the crystal plasticity toolbox to cloud, and the main ones were the following:

- Creation of the Python executable modules from Fortran code using F2PY, and importing them successfully within the Python code used for the Dash application
- Deploying the Dash application to cloud with the F2PY-generated Python executable modules from Fortran code and oneMKL [57]

At the beginning of the project, the first goal was to successfully create the Python executable modules from the existing Fortran code for crystal plasticity calculations. F2PY needs a Fortran compiler to be able to generate the modules and it was important to select a compatible compiler, not just with the operating system but also with the existing Fortran codes. Originally, a machine with a Mac operating system and an Apple M1 silicon processor was utilized, however it had compatibility issues with Intel® Fortran, which is only compatible with computers with Intel® Processors. It was also attempted to utilize the GFortran [58] compiler in Mac, however issues with linking the Linear Algebra PACKage (LAPACK) [59] libraries, utilized in the `crystal_plasticity.f90` Fortran code, were encountered during compilation. With this, it was decided to move development to a Windows-based computer. Using a machine with Windows operating system, Python executable modules were successfully generated with F2PY using the Intel® Fortran compiler, with the detailed steps discussed in Section 3.4. The generated modules, in *.pyd* format were successfully imported to the Python code in Dash, thus, these were utilized during the initial implementation of the required functionalities and debugging of the application. At this point, the application was only deployed locally in the Windows machine.

The next goal was to deploy the Dash application to cloud. The initial attempt for the cloud deployment used the files of the built application that was working locally, which included the Python executable modules in *.pyd* format. The steps provided by Microsoft Azure on how to deploy a Python app to Azure App Service[60] were followed, however the build always failed since Azure App Service was Linux-based, thus the Python executable modules in *.pyd* format were incompatible and could not be imported successfully. Due to this, it was decided to try to generate Linux-compatible Python executable modules, or those with *.so* format, and try to deploy these with the application instead. A machine with Ubuntu operating system was used, and the first attempt to generate the Python executable modules in the Linux environment utilized Intel® Fortran as the compiler. However, when trying to deploy the web application to Azure using the *.so* files generated with Intel® Fortran, the error "*libifport.so.5 not found*" was encountered and the web application failed to run. Another attempt was done, wherein GFortran was utilized as the compiler, however the generated *.so* files still caused an error during deployment which is "*libgfortran.so.5 not found*".

It was decided to proceed with using Docker in the deployment to Azure to try to solve the errors related to the missing `libgfortran` [61] and `libifport` [62] libraries. An article in Medium detailing the steps on how to deploy a Dash application using a Docker image to Azure [48] was found, however, the same errors were encountered when trying to run the Docker images, both in the local machine and in the cloud. There was no sufficient information found in Stackoverflow or in Google on how to solve these aforementioned errors, but there was a recently introduced AI chatbot on November 2022 called ChatGPT [63], which was able to provide a solution to the problem. A chat with this AI chatbot provided solutions to the error, as can be seen in the conversation screenshots in Figure A.1 and Figure A.2 in Appendix A. ChatGPT was also able to solve the problem which was encountered pertaining to the missing oneMKL in the deployment as can be seen in Figure A.3 in Appendix A. Together, the solutions provided by ChatGPT and the steps provided by Medium article, the Dash application was successfully deployed to Azure using Docker.

Chapter 5

Conclusion

5.1 Conclusion

The crystal plasticity toolbox, originally consisting of Python and Fortran code with very limited user base, is now made accessible as a web application with user-friendly GUI. It is deployed to cloud in Azure with the help of Docker, which allows packaging of the application and its dependencies into a container, ensuring that the application will run consistently across different environments. Dash is utilized for the creation of the GUI, and the existing Fortran codes for crystal plasticity models are utilized through the creation of Python executable modules with the help of F2PY. This proves that legacy Fortran codes are viable to be utilized for web application development. Dash has also been proven to be able to support data visualization requirements for Taylor and Alamel crystal plasticity models. The toolbox is able to generate pole figures in both point and contour plots, inverse pole figures in static and animated versions, ODF figures, yield surfaces, and R-value figures.

5.2 Future work

The cloud-based crystal plasticity toolbox opens a lot of possibilities for future work, one of which is the possibility of adding yield surface fitting for the generation of the yield surface. Another is that the pole figure contour plots are currently generated using a grid approach and this can be improved by using kernel density functions. Also, both the ODF and pole figures are still generated through Matplotlib, which currently lacks interactive functionalities when used with Dash, thus converting these to Plotly graphs can be considered. And, since the upload of *.ang* files has already been introduced in the application, EBSD grain visualization can also be considered as an added functionality.

General improvements for the application can also be done, such as for the performance, in terms of speed. Although this was originally included the non-functional requirements, time did not permit for further performance improvements, specially for the generation of the yield surface and R-values figures. The Python version used for the current version of the application is 3.7.5, and the upgrade to a more recent version of Python is something that needs to done. Also, the export of the output results can also be implemented so that further processing of data can be done, possibly in other external applications, if desired.

One possibility that can also be explored for the future of this application is to check how this can be utilized to possibly contribute to the efforts of the European Materials and Modelling Ontology (EMMO), which provides a structured and standardized way to organize concepts and relationships within the materials science and engineering domain. It serves as a common language and knowledge framework for describing materials, processes, properties, and models.

References

[1] A. S. Khan, *Continuum theory of plasticity*. New York ; Chichester: Wiley, 1995.

[2] Z. Jiang, J. Zhao and H. Xie, *Microforming Technology*. Academic Press, 2017, pp. 87–108.

[3] G. Sachs, 'The plastic deformation mode of polycrystals,' *Z. Verein Deut. Ing*, vol. 72, p. 734, 1928.

[4] G. I. Taylor, 'Plastic strain in metals,' *Journal of the Institute of Metals*, vol. 62, pp. 307–324, 1938.

[5] P. Van Houtte, S. Li, M. Seefeldt and L. Delannay, 'Deformation texture prediction: From the taylor model to the advanced lamel model,' *International journal of plasticity*, vol. 21, no. 3, pp. 589–624, 2005.

[6] R. Lebensohn and C. Tomé, 'A self-consistent anisotropic approach for the simulation of plastic deformation and texture development of polycrystals: Application to zirconium alloys,' *Acta metallurgica et materialia*, vol. 41, no. 9, pp. 2611–2624, 1993.

[7] D. D. Tjahjanto, P. Eisenlohr and F. Roters, 'A novel grain cluster-based homogenization scheme,' *Modelling and simulation in materials science and engineering*, vol. 18, no. 1, 2010.

[8] D. Peirce, R. Asaro and A. Needleman, 'Material rate dependence and localized deformation in crystalline solids,' *Acta metallurgica*, vol. 31, no. 12, pp. 1951–1976, 1983.

[9] F. Roters, M. Diehl, P. Shanthraj, P. Eisenlohr, C. Reuber, S. Wong, T. Maiti, A. Ebrahimi, T. Hochrainer, H.-O. Fabritius, S. Nikolov, M. Friák, N. Fujita, N. Grilli, K. Janssens, N. Jia, P. Kok, D. Ma, F. Meier, E. Werner, M. Stricker, D. Weygand and 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*, vol. 158, pp. 420–478, 2019.

[10] K. Chung and M.-G. Lee, *Basics of Continuum Plasticity*. Singapore: Springer, 2018.

[11] D. Hull and D. Bacon, 'Chapter 3 - movement of dislocations,' in *Introduction to Dislocations (Fifth Edition)*, Fifth Edition, Oxford: Butterworth-Heinemann, 2011, pp. 43–62.

[12] A. G. Jackson, 'Slip systems,' in *Handbook of Crystallography: For Electron Microscopists and Others*. New York, NY: Springer New York, 1991.

[13] Y. Guilhem, S. Basseville, F. Curtit, J.-M. Stéphan and G. Cailletaud, 'Numerical investigations of the free surface effect in three-dimensional polycrystalline aggregates,' *Computational Materials Science*, vol. 70, pp. 150–162, 2013.

[14] W. Wang and W. Wen, 'Modeling in crystal plasticity: From theory to application,' in *Encyclopedia of Materials: Metals and Alloys*, F. G. Caballero, Ed., Oxford: Elsevier, 2022, pp. 552–560.

[15] T. Mánik, *Plastic-anisotropy modelling : Texture, strain-rate and strain-path changes*, Trondheim, 2013.

[16] T. Mánik and B. Holmedal, 'Additional relaxations in the alamel texture model,' *Materials Science and Engineering: A*, vol. 580, pp. 349–354, 2013.

[17] P. Van Houtte, 'On the equivalence of the relaxed taylor theory and the bishop-hill theory for partially constrained plastic deformation of crystals,' *Materials Science and Engineering*, vol. 55, no. 1, pp. 69–77, 1982.

[18] K. Zhang, 'Multi-level modelling of plastic anisotropy of aluminium alloys using crystal plasticity models and advanced yield functions,' 2014.

[19] J. F. W. Bishop and R. Hill, 'Xlvi. a theory of the plastic distortion of a polycrystalline aggregate under combined stresses.,' *Philosophical Magazine Series 1*, vol. 42, pp. 414–427, 1951.

[20] J. F. W. Bishop and R. Hill, 'Cxxviii. a theoretical derivation of the plastic properties of a polycrystalline face-centred metal,' *Philosophical Magazine Series 1*, vol. 42, pp. 1298–1307, 1951.

[21] F. Roters, P. Eisenlohr, L. Hantcherli, D. Tjahjanto, T. Bieler and D. Raabe, 'Overview of constitutive laws, kinematics, homogenization and multiscale methods in crystal plasticity finite-element modeling: Theory, experiments, applications,' *Acta Materialia*, vol. 58, no. 4, pp. 1152–1211, 2010.

[22] S. Suwas and R. K. Ray, 'Representation of texture,' in *Crystallographic Texture of Materials*. London: Springer London, 2014, pp. 11–38.

[23] H.-J. Bunge, '11 - some results of texture analysis,' in *Texture Analysis in Materials Science*, H.-J. Bunge, Ed., Butterworth-Heinemann, 1982, pp. 226–278.

[24] F. Barlat, D. J. Lege and J. C. Brem, 'A six-component yield function for anisotropic materials,' eng, *International journal of plasticity*, vol. 7, no. 7, pp. 693–712, 1991.

[25] J. Bishop and R. Hill, 'Xlvi. a theory of the plastic distortion of a polycrystalline aggregate under combined stresses.,' *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 42, no. 327, pp. 414–427, 1951.

[26] A. V. Hershey, 'The plasticity of an isotropic aggregate of anisotropic face-centered cubic crystals,' *Journal of Applied Mechanics*, vol. 21, no. 3, pp. 241–249, Jun. 2021.

[27] W. F. Hosford, 'A generalized isotropic yield criterion,' *Journal of Applied Mechanics*, vol. 39, no. 2, pp. 607–609, Jun. 1972.

[28] K. K. Mathur and P. R. Dawson, 'On modeling the development of crystallographic texture in bulk forming processes,' *International Journal of Plasticity*, vol. 5, no. 1, pp. 67–94, 1989.

[29] P. S. Foundation, *General python faq*. [Online]. Available: `https://docs.python.org/3/faq/general.html/` (visited on 12/05/2023).

[30] V. Decyk, C. Norton and H. Gardner, 'Why fortran?' *Computing in Science Engineering*, vol. 9, pp. 68–71, Aug. 2007.

[31] NumPy, *Using f2py*. [Online]. Available: `https://numpy.org/doc/stable/f2py/usage.html` (visited on 12/05/2023).

[32] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke and T. E. Oliphant, 'Array programming with NumPy,' *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[33] Plotly, *Introduction to dash*. [Online]. Available: `https://dash.plotly.com/introduction` (visited on 12/05/2023).

[34] Plotly, *Dash installation*. [Online]. Available: `https://dash.plotly.com/installation` (visited on 12/05/2023).

[35] Plotly, *Dash layout*. [Online]. Available: `https://dash.plotly.com/layouts` (visited on 12/05/2023).

[36] Faculty, *Dash bootstrap components*. [Online]. Available: `https://dash-bootstrap-components.opensource.faculty.ai/` (visited on 12/05/2023).

[37] Plotly, *Basic dash callbacks*. [Online]. Available: `https://dash.plotly.com/basic-callbacks` (visited on 12/05/2023).

[38] Microsoft, *Visual studio code*. [Online]. Available: `https://https://code.visualstudio.com/docs/` (visited on 12/05/2023).

[39] Docker, *Docker overview*. [Online]. Available: `https://docs.docker.com/get-started/overview/` (visited on 12/05/2023).

[40] Microsoft, *Get to know azure*. [Online]. Available: `https://azure.microsoft.com/en-us/explore/` (visited on 12/05/2023).

[41] Microsoft, *App service documentation*. [Online]. Available: `https://learn.microsoft.com/fil-ph/azure/app-service/` (visited on 12/05/2023).

[42] Microsoft, *Azure container registry documentation*. [Online]. Available: `https://learn.microsoft.com/en-us/azure/container-registry/` (visited on 12/05/2023).

[43] D. Voorhees, *Guide to Efficient Software Design: An MVC Approach to Concepts, Structures, and Models* (Texts in Computer Science). Springer International Publishing, 2021.

[44] T. Park, *Lux*. [Online]. Available: `https://bootswatch.com/lux/` (visited on 12/05/2023).

[45] Anaconda, *Anaconda distribution*. [Online]. Available: `https://www.anaconda.com/products/distribution` (visited on 12/05/2023).

[46] Intel, *Intel® fortran compiler*. [Online]. Available: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/fortran-compiler.html` (visited on 12/05/2023).

[47] Intel, *Get intel® oneapi math kernel library (onemkl)*. [Online]. Available: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-download.html` (visited on 12/05/2023).

[48] E. Vioque, *Deploy a dash application in azure using docker*, 2020. [Online]. Available: `https://medium.com/swlh/deploy-a-dash-application-in-azure-using-docker-ed46c4b9d2b2` (visited on 12/05/2023).

[49] Microsoft, *Visual studio code*. [Online]. Available: `https://code.visualstudio.com/` (visited on 12/05/2023).

[50] Docker, *Docker desktop*. [Online]. Available: `https://www.docker.com/products/docker-desktop/` (visited on 12/05/2023).

[51] H. A. Moradi, *Textool*. [Online]. Available: `https://gitlab.com/HassanMo/textool` (visited on 22/05/2023).

[52] M. McKerns, *Dill*. [Online]. Available: `https://pypi.org/project/dill/` (visited on 12/05/2023).

[53] P. S. Foundation, *Pickle — python object serialization*. [Online]. Available: `https://docs.python.org/3/library/pickle.html` (visited on 12/05/2023).

[54] Plotly, *Dcc.store*. [Online]. Available: `https://dash.plotly.com/dash-core-components/store` (visited on 12/05/2023).

[55] J. D. Hunter, 'Matplotlib: A 2d graphics environment,' *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[56] P. T. Inc., *Collaborative data science*, Montreal, QC, 2015. [Online]. Available: `https://plot.ly` (visited on 12/05/2023).

[57] Intel, *Get started with intel® oneapi math kernel library*. [Online]. Available: `https://www.intel.com/content/www/us/en/docs/onemkl/get-started-guide/2023-0/overview.html` (visited on 12/05/2023).

[58] I. Free Software Foundation, *Gfortran - gcc wiki*. [Online]. Available: `https://gcc.gnu.org/wiki/GFortran` (visited on 12/05/2023).

[59] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney and D. Sorensen, *LAPACK Users' Guide*, Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[60] Microsoft, *Quickstart: Deploy a python (django or flask) web app to azure - azure app service*. [Online]. Available: `https://learn.microsoft.com/nb-no/azure/app-service/quickstart-python` (visited on 12/05/2023).

[61] Anaconda, *Libgfortran*. [Online]. Available: `https://anaconda.org/conda-forge/libgfortran/files` (visited on 12/05/2023).

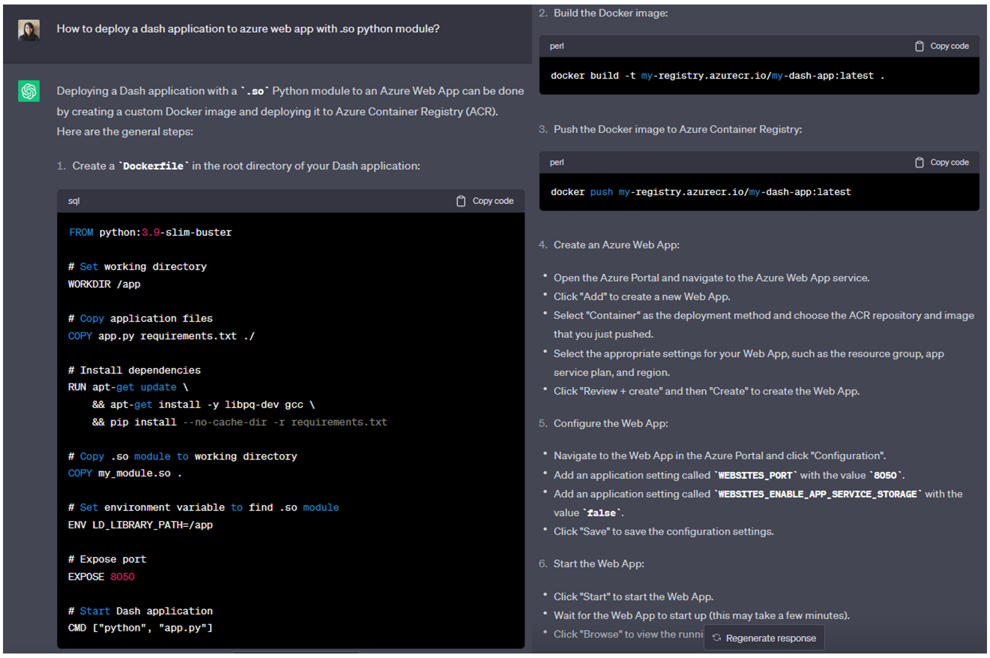
[62] Intel, *Ifport portability library*. [Online]. Available: `http://ahamodel.uib.no/intel/GUID-01E8863C-CAD1-4345-B674-F75262B7152A.html` (visited on 12/05/2023).

[63] OpenAI, *Chatgpt*. [Online]. Available: `https://chat.openai.com/` (visited on 12/05/2023).

Appendix A

Conversations with an AI (ChatGPT)

These are conversations with ChatGPT, an artificial intelligence chatbot developed by OpenAI. Some solutions to some of the challenges encountered during the implementation of the application were solved through these "conversations".



How to deploy a dash application to azure web app with .so python module?

Deploying a Dash application with a `.so` Python module to an Azure Web App can be done by creating a custom Docker image and deploying it to Azure Container Registry (ACR). Here are the general steps:

1. Create a `Dockerfile` in the root directory of your Dash application:

```
FROM python:3.9-slim-buster

# Set working directory
WORKDIR /app

# Copy application files
COPY app.py requirements.txt ./

# Install dependencies
RUN apt-get update \
    && apt-get install -y libpq-dev gcc \
    && pip install --no-cache-dir -r requirements.txt

# Copy .so module to working directory
COPY my_module.so .

# Set environment variable to find .so module
ENV LD_LIBRARY_PATH=/app

# Expose port
EXPOSE 8050

# Start Dash application
CMD ["python", "app.py"]
```

2. Build the Docker image:

```
docker build -t my-registry.azurecr.io/my-dash-app:latest .
```

3. Push the Docker image to Azure Container Registry:

```
docker push my-registry.azurecr.io/my-dash-app:latest
```

4. Create an Azure Web App:

- Open the Azure Portal and navigate to the Azure Web App service.
- Click "Add" to create a new Web App.
- Select "Container" as the deployment method and choose the ACR repository and image that you just pushed.
- Select the appropriate settings for your Web App, such as the resource group, app service plan, and region.
- Click "Review + create" and then "Create" to create the Web App.

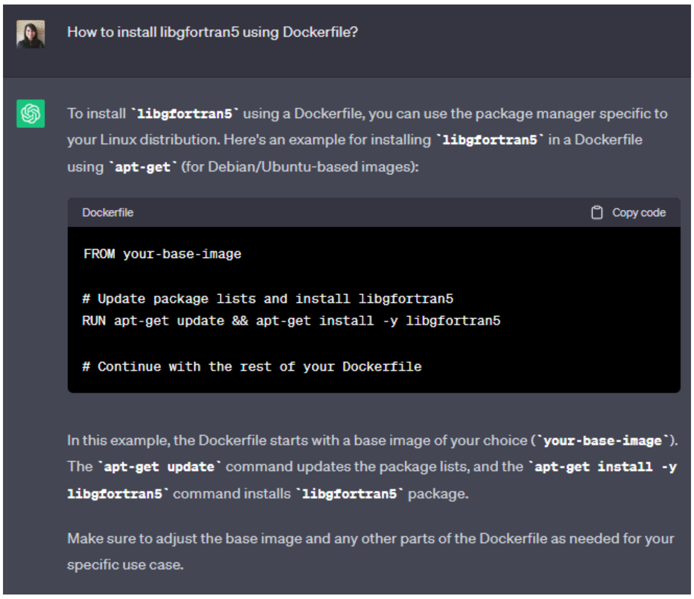
5. Configure the Web App:

- Navigate to the Web App in the Azure Portal and click "Configuration".
- Add an application setting called `WEBSITES_PORT` with the value `8050`.
- Add an application setting called `WEBSITES_ENABLE_APP_SERVICE_STORAGE` with the value `false`.
- Click "Save" to save the configuration settings.

6. Start the Web App:

- Click "Start" to start the Web App.
- Wait for the Web App to start up (this may take a few minutes).
- Click "Browse" to view the runni

Figure A.1: ChatGPT: Steps how to deploy a Dash application with .so python modules to Azure with Docker.



How to install libgfortran5 using Dockerfile?

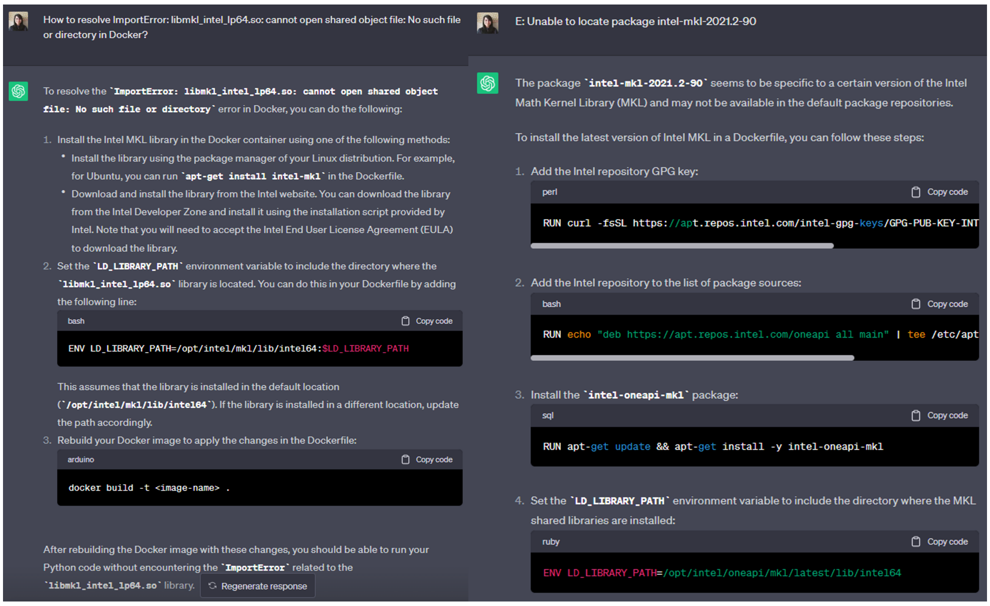
To install `libgfortran5` using a Dockerfile, you can use the package manager specific to your Linux distribution. Here's an example for installing `libgfortran5` in a Dockerfile using `apt-get` (for Debian/Ubuntu-based images):

```
FROM your-base-image

# Update package lists and install libgfortran5
RUN apt-get update && apt-get install -y libgfortran5

# Continue with the rest of your Dockerfile
```

In this example, the Dockerfile starts with a base image of your choice (`your-base-image`). The `apt-get update` command updates the package lists, and the `apt-get install -y libgfortran5` command installs `libgfortran5` package.

Make sure to adjust the base image and any other parts of the Dockerfile as needed for your specific use case.

Figure A.2: ChatGPT: Steps how to install libgfortran5 using Dockerfile.



How to resolve ImportError: libmkl_intel_lp64.so: cannot open shared object file: No such file or directory in Docker?

To resolve the `ImportError: libmkl_intel_lp64.so: cannot open shared object file: No such file or directory` error in Docker, you can do the following:

1. Install the Intel MKL library in the Docker container using one of the following methods:
   - Install the library using the package manager of your Linux distribution. For example, for Ubuntu, you can run `apt-get install intel-mkl` in the Dockerfile.
   - Download and install the library from the Intel website. You can download the library from the Intel Developer Zone and install it using the installation script provided by Intel. Note that you will need to accept the Intel End User License Agreement (EULA) to download the library.
2. Set the `LD_LIBRARY_PATH` environment variable to include the directory where the `libmkl_intel_lp64.so` library is located. You can do this in your Dockerfile by adding the following line:

```
ENV LD_LIBRARY_PATH=/opt/intel/mkl/lib/intel64:$LD_LIBRARY_PATH
```

   This assumes that the library is installed in the default location (`/opt/intel/mkl/lib/intel64`). If the library is installed in a different location, update the path accordingly.
3. Rebuild your Docker image to apply the changes in the Dockerfile:

```
docker build -t <image-name> .
```

After rebuilding the Docker image with these changes, you should be able to run your Python code without encountering the `ImportError` related to the `libmkl_intel_lp64.so` library.

E: Unable to locate package intel-mkl-2021.2-90

The package `intel-mkl-2021.2-90` seems to be specific to a certain version of the Intel Math Kernel Library (MKL) and may not be available in the default package repositories.

To install the latest version of Intel MKL in a Dockerfile, you can follow these steps:

1. Add the Intel repository GPG key:

```
RUN curl -fsSL https://apt.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INT
```

2. Add the Intel repository to the list of package sources:

```
RUN echo "deb https://apt.repos.intel.com/oneapi all main" | tee /etc/apt
```

3. Install the `intel-oneapi-mkl` package:

```
RUN apt-get update && apt-get install -y intel-oneapi-mkl
```

4. Set the `LD_LIBRARY_PATH` environment variable to include the directory where the MKL shared libraries are installed:

```
ENV LD_LIBRARY_PATH=/opt/intel/oneapi/mkl/latest/lib/intel64
```

Figure A.3: ChatGPT: Steps to install Intel® oneAPI Math Kernel Library.

Appendix B

Code listings

The code listings can also be found in: `https://gitlab.com/ntnu-physmet/master-students/crystal-plasticity-dashboard/-/tree/CP_Demo`.

B.1 Pre-requisite Installation

Code listing B.1: requirements.txt

```
# This file may be used to create an environment using:
# $ conda create --name <env> --file <this file>
# platform: win-64
matplotlib
ipywidgets
ipython
dash
dash-bootstrap-components
dash_bootstrap_templates
ipython_genutils
scipy
pandas
dill
dash-loading-spinners
gunicorn
quadprog
```

Code listing B.2: Dockerfile.prod

```
FROM python:3.7

# Create a directory where the code is to be hosted
RUN mkdir /app

# Define the working directory in the container
WORKDIR /app

# Copy and install the requirements.
COPY requirements.txt /app/requirements.txt

# Install dependencies
RUN apt-get update \
```

```
    && apt-get install -y libpq-dev gcc \
    && apt-get install libgfortran5 \
    && pip install --no-cache-dir -r requirements.txt \
    && pip install ipykernel

RUN apt-get update && apt-get install -y \
    curl

RUN curl -fsSL https://apt.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-SW-PRODUCTS.PUB | apt-key add -
RUN echo "deb␣https://apt.repos.intel.com/oneapi␣all␣main" | tee /etc/apt/sources.list.d/oneAPI.list
RUN apt-get update && apt-get install -y intel-oneapi-mkl


# Copy application code to the image
COPY . /app/

# Define environment variables
ENV LD_LIBRARY_PATH=/opt/intel/oneapi/mkl/latest/lib/intel64
ENV dash_port=80
ENV dash_debug="False"

CMD ["python", "app.py"]
```

B.2 Layout Codes

Code listing B.3: Code for sidebar layout

```python
from dash import html, dcc, dash_table
import dash_bootstrap_components as dbc
import pandas as pd


# This styles the sidebar to be fixed and always shown in the left of the UI
SIDEBAR_STYLE = {
    "position": "fixed",
    "top": 0,
    "left": 0,
    "bottom": 0,
    "width": "30rem",
    "padding": "2rem␣1rem",
    "overflow": "auto",
    "background-color": "#ebebeb",
}

# This defines the layout of the input fields in the sidebar
sidebar = html.Div([

        # Sidebar Header
        html.H4("Setup"),
        html.Hr(),

        # Expandable/collapsable section in the sidebar for the grain orientation input
        dbc.Accordion([
            dbc.AccordionItem([
```

```

                # Dropdown field for the model
                dbc.Row([
                    dbc.Col([
                        dbc.Label("MODEL"),
                        dbc.Select(
                            id = 'input-cp-model',
                            options =[
                                    {'label':"FC␣Taylor", 'value':"FCTAYLOR"},
                                    {'label':"Alamel", 'value':"ALAMEL"},
                                    ],
                            value = "FCTAYLOR"
                        )
                    ])
                ]),

                # Dropdown field for the grain orientaion
                html.Br(),
                dbc.Row([
                    dbc.Col([
                        dbc.Label("GRAIN␣ORIENTATION"),
                        dbc.Select(
                            id = 'input-grain-ori-state',
                            options =[
                                # {'label':"Random", 'value':"Random"},
                                # {'label': "Specify Euler Angles (Bunge's notation
                                     in degrees)", 'value':"Specify Euler Angles (
                                    Bunge's notation in degrees)"},
                                {'label': "Input␣Texture␣Components", 'value':"
                                    Input␣Texture␣Components"},
                                {'label': "Upload␣file", 'value':"Upload␣file"},
                                ],
                            value = "Input␣Texture␣Components"
                        )
                    ])
                ]),

                # Select Texture Components Label
                dbc.Row([
                    html.Br(),
                    dbc.Label("Select␣texture␣component:")
                ], id = 'texture-component-label'),

                dbc.Row([

                    # Toggle Selection for Cube Texture Component
                    dbc.Col([
                        dbc.Checklist(
                            options=[
                                {"label": "Cube", "value": "Cube"},
                            ],
                            value=[],
                            switch=True,
                            id = 'texture-switch-cube'
                        ),
                    ], id = 'texture-component-switch-cube'),

                    # Toggle Selection for Brass Texture Component
                    dbc.Col([
                        dbc.Checklist(
```

```
                            options=[
                                {"label": "Brass", "value": "Brass"},
                            ],
                            value=[],
                            switch=True,
                            id = ’texture-switch-brass’
                        ),
                    ], id = ’texture-component-switch-brass’),

                    # Toggle Selection for S Texture Component
                    dbc.Col([
                        dbc.Checklist(
                            options=[
                                {"label": "S", "value": "S"},
                            ],
                            value=[],
                            switch=True,
                            id = ’texture-switch-s’
                        ),
                    ], id = ’texture-component-switch-s’),
                ]),

                dbc.Row([

                    # Toggle Selection for Goss Texture Component
                    dbc.Col([
                        dbc.Checklist(
                            options=[
                                {"label": "Goss", "value": "Goss"},
                            ],
                            value=[],
                            switch=True,
                            id = ’texture-switch-goss’
                        ),
                    ], id = ’texture-component-switch-goss’),

                    # Toggle Selection for Copper Texture Component
                    dbc.Col([
                        dbc.Checklist(
                            options=[
                                {"label": "Copper", "value": "Copper"},
                            ],
                            value=[],
                            switch=True,
                            id = ’texture-switch-copper’
                        ),
                    ], id = ’texture-component-switch-copper’),

                    # Toggle Selection for Random Texture Component
                    dbc.Col([
                        dbc.Checklist(
                            options=[
                                {"label": "Random", "value": "Random"},
                            ],
                            value=[],
                            switch=True,
                            id = ’texture-switch-random’
                        ),
                    ], id = ’texture-component-switch-random’),
                ]),
```

```

                dbc.Row([

                    # Toggle Selection for Fiber Texture Component
                    dbc.Col([
                        dbc.Checklist(
                            options=[
                                {"label": "Fiber", "value": "Fiber"},
                            ],
                            value=[],
                            switch=True,
                            id = 'texture-switch-fiber'
                        ),
                    ], width = 4, id = 'texture-component-switch-fiber'),

                    # Toggle Selection for User Input Texture Component
                    dbc.Col([
                        dbc.Checklist(
                            options=[
                                {"label": "User␣Input", "value": "User␣Input"},
                            ],
                            value=[],
                            switch=True,
                            id = 'texture-switch-user-input'
                        ),
                    ], id = 'texture-component-switch-user-input'),


                ]),

                # Input for Texture Components

                # Input for Cube Texture Components
                dbc.Row([
                    dbc.Card([
                        dbc.CardBody([
                            dbc.Row([
                                dbc.Col([
                                    dbc.Badge("i",id = 'info-cube', class_name="badge␣rounded-pill␣bg-info", text_color="white",style={'margin-top': '15px'}),
                                    dbc.Popover("Texture:␣90,90,45", body = True, target = 'info-cube', trigger ='hover')
                                ]),
                                dbc.Col([
                                    dbc.Label("Cube",style={'margin-top': '15px'})
                                ], width = 3),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = 'sm',type = 'number',min = 0,max = 100,step=0.1,id = 'spread-cube',value=0),
                                        dbc.Label("Spread")
                                    ]),
                                ], width=3),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = 'sm',type = 'number',min = 0,max = 10000,step =1,id = 'number-grains-cube',value=0),
```

```
                                        dbc.Label("#␣of␣Grains")
                                    ]),
                                ], width=5,style={’margin-left’: ’6px’}),
                            ], className="g-0"),
                        ]),
                    ], style = {’margin’: ’0’}),
                ], id = ’texture-component-input-cube’),

                # Input for Goss Texture Components
                dbc.Row([
                    dbc.Card([
                        dbc.CardBody([
                            dbc.Row([
                                dbc.Col([
                                    dbc.Badge("i",id = ’info-goss’, class_name="badge␣rounded-pill␣bg-info", text_color="white",style={’margin-top’: ’15px’}),
                                    dbc.Popover("Texture:␣90,90,45", body = True, target = ’info-goss’, trigger =’hover’)
                                ]),
                                dbc.Col([
                                    dbc.Label("Goss",style={’margin-top’: ’15px’})
                                ], width = 3),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step=0.1,id = ’spread-goss’,value=0),
                                        dbc.Label("Spread")
                                    ]),
                                ], width=3),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 10000,step =1,id = ’number-grains-goss’,value=0),
                                        dbc.Label("#␣of␣Grains")
                                    ]),
                                ], width=5,style={’margin-left’: ’6px’}),
                            ], className="g-0"),
                        ]),
                    ]),
                ], id = ’texture-component-input-goss’),

                # Input for Brass Texture Components
                dbc.Row([
                    dbc.Card([
                        dbc.CardBody([
                            dbc.Row([
                                dbc.Col([
                                    dbc.Badge("i",id = ’info-brass’, class_name="badge␣rounded-pill␣bg-info", text_color="white",style={’margin-top’: ’15px’}),
                                    dbc.Popover("Texture:␣55,90,45", body = True, target = ’info-brass’, trigger =’hover’)
                                ]),
                                dbc.Col([
                                    dbc.Label("Brass",style={’margin-top’: ’15px’})
                                ], width = 3),
                                dbc.Col([
```

```
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step =0.1,id = ’spread-brass’,value=0),
                                        dbc.Label("Spread")
                                    ]),
                                ], width=3),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 10000,step =1,id = ’number-grains-brass’,value=0),
                                        dbc.Label("#␣of␣Grains")
                                    ]),
                                ], width=5,style={’margin-left’: ’6px’}),
                            ], className="g-0"),
                        ]),
                    ]),
                ], id = ’texture-component-input-brass’),

                # Input for Copper Texture Components
                dbc.Row([
                    dbc.Card([
                        dbc.CardBody([
                            dbc.Row([
                                dbc.Col([
                                    dbc.Badge("i",id = ’info-copper’, class_name="badge␣rounded-pill␣bg-info", text_color="white",style={’margin-top’: ’15px’}),
                                    dbc.Popover("Texture:␣90,35,45", body = True, target = ’info-copper’, trigger =’hover’)
                                ]),
                                dbc.Col([
                                    dbc.Label("Copper",style={’margin-top’: ’15px’})
                                ], width = 3),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step =0.1,id = ’spread-copper’,value=0),
                                        dbc.Label("Spread")
                                    ]),
                                ], width=3),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 10000,step =1,id = ’number-grains-copper’,value=0),
                                        dbc.Label("#␣of␣Grains")
                                    ]),
                                ], width=5,style={’margin-left’: ’6px’}),
                            ], className="g-0"),
                        ]),
                    ]),
                ], id = ’texture-component-input-copper’),


                # Input for S Texture Components
                dbc.Row([
```

```
                    dbc.Card([
                        dbc.CardBody([

                            dbc.Row([
                                dbc.Col([
                                    dbc.Badge("i",id = ’info-s’, class_name="badge␣rounded-pill␣bg-info", text_color="white", style={’margin-top’: ’15px’}),
                                    dbc.Popover("Texture:␣59,37,63", body = True, target = ’info-s’, trigger =’hover’)
                                ]),
                                dbc.Col([
                                    dbc.Label("S",style={’margin-top’: ’15px’})
                                ], width = 3),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step =0.1,id = ’spread-s’, value=0),
                                        dbc.Label("Spread")
                                    ]),
                                ], width=3),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 10000,step =1,id = ’number-grains-s’,value=0),
                                        dbc.Label("#␣of␣Grains")
                                    ]),
                                ], width=5,style={’margin-left’: ’6px’}),
                            ], className="g-0"),

                        ]),
                    ]),
                ],id = ’texture-component-input-s’),

                # Input for Random Texture Components
                dbc.Row([
                    dbc.Card([
                        dbc.CardBody([
                            dbc.Row([
                                dbc.Col([
                                    dbc.Badge("i",id = ’info-random’, class_name="badge␣rounded-pill␣bg-info", text_color="white",style={’margin-top’: ’15px’}),
                                    dbc.Popover("Texture:␣Random", body = True, target = ’info-random’, trigger =’hover’)
                                ]),
                                dbc.Col([
                                    dbc.Label("Random",style={’margin-top’: ’15px’})
                                ], width = 3),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’string’,id = ’spread-random’,disabled = True, value =’Rand’),
                                        dbc.Label("Spread")
                                    ]),
                                ], width=3),
```

```
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 10000,step =1,id = ’number-grains-random’,value=0),
                                        dbc.Label("#␣of␣Grains")
                                    ]),
                                ], width=5,style={’margin-left’: ’6px’}),
                            ], className="g-0"),
                        ]),
                    ]),
                ], id = ’texture-component-input-random’),

                # Input for Fiber Texture Component
                dbc.Row([
                    dbc.Card([
                        dbc.CardBody([
                            dbc.Row([
                                dbc.Col([
                                    dbc.Badge("i",id = ’info-fiber’, class_name="badge␣rounded-pill␣bg-info", text_color="white",style={’margin-top’: ’15px’}),
                                    dbc.Popover("Texture:␣User␣Input", body = True, target = ’info-fiber’, trigger =’hover’)
                                ]),
                                dbc.Col([
                                    dbc.Label("Fiber",style={’margin-top’: ’15px’})
                                ], width = 3),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step =0.1,id = ’spread-fiber’,value=0),
                                        dbc.Label("Spread")
                                    ]),
                                ], width=3),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 10000,step =1,id = ’number-grains-fiber’,value=0),
                                        dbc.Label("#␣of␣Grains")
                                    ]),
                                ], width=5,style={’margin-left’: ’6px’}),
                            ],className="g-0"),

                            dbc.Row([
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step =1,id = ’euler-phi1-fiber’,value=0),
                                        dbc.Label(["\u03C6",html.Sub(1)])
                                    ]),
                                ], style={’width’:’123px’}),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step =1,id = ’euler-theta-fiber’,value=0),
```

```
                                        dbc.Label("\u03B8")
                                    ]),
                                ], style={’width’:’123px’,’margin-left’: ’6px’}),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step =1,id = ’euler-phi2-fiber’,value=0),
                                        dbc.Label(["\u03C6",html.Sub(2)])
                                    ]),
                                ], style={’width’:’123px’,’margin-left’: ’6px’}),
                            ],className="g-0",style={’margin-top’: ’6px’}),

                            dbc.Row([
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step =1,id = ’axis-x’, value=0),
                                        dbc.Label("x")
                                    ]),
                                ], style={’width’:’123px’}),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step =1,id = ’axis-y’, value=0),
                                        dbc.Label("y")
                                    ]),
                                ], style={’width’:’123px’,’margin-left’: ’6px’}),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step =1,id = ’axis-z’, value=0),
                                        dbc.Label("z")
                                    ]),
                                ], style={’width’:’123px’,’margin-left’: ’6px’}),
                            ],className="g-0",style={’margin-top’: ’6px’})
                        ]),
                    ]),
                ], id = ’texture-component-input-fiber’),

                # Input for User Input Texture Components
                dbc.Row([
                    dbc.Card([
                        dbc.CardBody([
                            dbc.Row([
                                dbc.Col([
                                    dbc.Badge("i",id = ’info-user-input’, class_name="badge␣rounded-pill␣bg-info", text_color="white",style={’margin-top’: ’15px’}),
                                    dbc.Popover("Texture:␣User␣Input", body = True, target = ’info-user-input’, trigger =’hover’)
                                ]),
                                dbc.Col([
                                    dbc.Label("User␣Input",style={’margin-top’: ’15px’})
```

```
                                ], width = 3),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step =0.1,id = ’spread-user-input’,value=0),
                                        dbc.Label("Spread")
                                    ]),
                                ], width=3),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 10000,step =1,id = ’number-grains-user-input’,value=0),
                                        dbc.Label("#␣of␣Grains")
                                    ]),
                                ], width=5,style={’margin-left’: ’6px’}),
                            ],className="g-0"),
                            dbc.Row([
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step =1,id = ’euler-phi1-user-input’,value=0),
                                        dbc.Label(["\u03C6",html.Sub(1)])
                                    ]),
                                ], style={’width’:’123px’}),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step =1,id = ’euler-theta-user-input’,value=0),
                                        dbc.Label("\u03B8")
                                    ]),
                                ], style={’width’:’123px’,’margin-left’: ’6px’}),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(size = ’sm’,type = ’number’,min = 0,max = 100,step =1,id = ’euler-phi2-user-input’,value=0),
                                        dbc.Label(["\u03C6",html.Sub(2)])
                                    ]),
                                ], style={’width’:’123px’,’margin-left’: ’6px’}),
                            ],className="g-0",style={’margin-top’: ’6px’})
                        ]),
                    ]),
                ], id = ’texture-component-input-user-input’),



                # Input field for .ang file threshold
                dbc.Row([
                    dbc.Col([
                        html.Br(),
                        dbc.FormFloating([
                            dbc.Input(
                            type = ’number’,
                            min = 0,
                            max = 1,
                            step = 0.01,
```

```
                            id = 'ang-threshold'
                        ),
                        dbc.Label("Threshold␣(0-1.0)")
                        ]),
                    ]),
                    html.Br()
                ],id = 'ang-threshold-hide'),

                # Upload file area
                dbc.Row([
                    dbc.Col([
                        dcc.Upload(
                            id = "uploaded-file",
                            children = html.Div(['Drag␣or␣select␣a␣file␣to␣upload' ]),
                            style={
                            'width': '90%',
                            'height': '110px',
                            'lineHeight': '60px',
                            'borderWidth': '1px',
                            'borderStyle': 'dashed',
                            'borderRadius': '5px',
                            'textAlign': 'center',
                            'margin': '10px'
                            },
                            multiple = False
                        ),
                        dbc.Alert(id="upload-indicator", color="success")
                    ]),
                ],id = 'upload-hide'),

                # Button to generate initial polycrystal
                dbc.Row([
                    dbc.Col([
                        dbc.Button(id='generate-polycrystal-button',
                                    className = 'btn-success',
                                    n_clicks=0,
                                    children='GENERATE␣POLYCRYSTAL',
                                    style={"margin-left": "10px"})
                    ]),
                ], id = 'generate-polycrystal-button-style', style ={'textAlign':'right', 'margin-top' : '20px'}),

            ],title = 'INITIAL'),
        ], flush = True),

        # Expandable/collapsable section in the sidebar for the loading options
        dbc.Accordion([
            dbc.AccordionItem([

                # Dropdown field for the loading condition
                dbc.Row([
                    dbc.Col([
                        dbc.Label("LOADING␣CONDITION"),
                        dbc.Select(
                            id = 'input-loading-condition',
                            options =[
                                    {'label':"Axisymmetric␣tension␣along␣z-axis", 'value':"Uniaxial␣tension"},
```

```
                                    {'label':"Plane-strain␣compression", 'value':"Plane-strain␣compression"},
                                    {'label': "Mixed␣(strain+stress)␣boundary␣condition", 'value':"Mixed␣(strain+stress)␣boundary␣condition"},
                                    {'label': "Input␣Custom␣Values", 'value':"Input␣Custom␣Values"}
                                    ],
                            value = "Uniaxial␣tension"
                        )
                    ])
                ]),

                # Matrix Input fields for the Global Velocity Gradiebt (L)
                html.Div([
                    html.Br(),
                    dbc.Row([
                        dbc.Col([
                            dbc.Row([
                                dbc.Label("GLOBAL␣VELOCITY␣GRADIENT␣(L)"),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(
                                        type = 'number',
                                        id = 'L-XX'
                                    ),
                                    dbc.Label("XX")
                                    ]),
                                ]),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(
                                        type = 'number',
                                        id = 'L-XY'
                                    ),
                                    dbc.Label("XY")
                                    ]),
                                ]),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(
                                        type = 'number',
                                        id = 'L-XZ'
                                    ),
                                    dbc.Label("XZ")
                                    ])
                                ])
                            ]),
                            html.Br(),
                            dbc.Row([
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(
                                        type = 'number',
                                        id = 'L-YX'
                                    ),
                                    dbc.Label("YX")
                                    ]),
                                ]),
                                dbc.Col([
```

```
                                    dbc.FormFloating([
                                        dbc.Input(
                                        type = ’number’,
                                        id = ’L-YY’
                                    ),
                                    dbc.Label("YY")
                                    ]),
                                ]),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(
                                        type = ’number’,
                                        id = ’L-YZ’
                                    ),
                                    dbc.Label("YZ")
                                    ])
                                ])
                            ]),
                            html.Br(),
                            dbc.Row([
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(
                                        type = ’number’,
                                        id = ’L-ZX’
                                    ),
                                    dbc.Label("ZX")
                                    ]),
                                ]),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(
                                        type = ’number’,
                                        id = ’L-ZY’
                                    ),
                                    dbc.Label("ZY")
                                    ]),
                                ]),
                                dbc.Col([
                                    dbc.FormFloating([
                                        dbc.Input(
                                        type = ’number’,
                                        id = ’L-ZZ’
                                    ),
                                    dbc.Label("ZZ")
                                    ])
                                ])
                            ])
                        ])
                    ]),
                ],id = ’loading-cond-hide’),

                # Input field for the Total Von Mises Strain
                dbc.Row([
                    dbc.Col([
                        html.Br(),
                        dbc.Label("TOTAL␣VON␣MISES␣STRAIN"),
                        dbc.Input(
                            id = ’input-von-mises-strain’,
                            type = ’number’,
```

```
                            step = 0.001,
                            value = 1
                        )
                    ])
                ]),

                # Input field for the Number of Steps
                dbc.Row([
                    dbc.Col([
                        html.Br(),
                        dbc.Label("NUMBER␣OF␣STEPS"),
                        dbc.Input(
                            id = 'input-number-steps',
                            type = 'number',
                            step = 1,
                            value = 100
                        )
                    ])
                ]),

                # Button to generate loaded polycrystal based on values entered in
                     LOAD section
                dbc.Row([
                    dbc.Col([
                        html.Br(),
                        dbc.Button(id='load-polycrystal-button',
                            className = 'btn-success',
                            n_clicks=0,
                            children='LOAD␣POLYCRYSTAL',
                            style={"margin-left": "10px"})
                    ], style={'textAlign':'right'})
                ])

            ], title = 'LOAD'),
        ], flush = True, start_collapsed=True),
    ],style=SIDEBAR_STYLE)
```

Code listing B.4: Code for content layout

```
from dash import html
import dash_bootstrap_components as dbc
import dash_loading_spinners as dls

# This styles the content to be fixed and always shown in the right of the UI
CONTENT_STYLE = {
    "margin-left": "32rem",
    "margin-right": "2rem",
    "padding": "2rem␣1rem",
    "overflow": "auto",
}

# This defines the layout of the tabs and generated graphs
content = html.Div([
            html.Div([

                # Content Header
                html.H1("CRYSTAL␣PLASTICITY␣TOOLBOX"),

                # This defined the tabs (Pole Figures, Inverse Pole Figures, ODF,
                     Yield Surface)
```

```
                dbc.Row([],id = 'loaded-warning'),
                dbc.Row([], id = 'initial-warning'),
                dbc.Row([
                    dbc.Tabs([
                        # Tab - Pole Figures
                        dbc.Tab([
                            dbc.Card([
                                dbc.Row([

                                    # Radio button for normal direction
                                    dbc.Col([
                                        dbc.Label("Normal:"),
                                        dbc.RadioItems(
                                            options=[
                                                {'label':'ND', 'value':'ND'},
                                                {'label':'TD', 'value':'TD'},
                                                {'label':'RD', 'value':'RD'}
                                            ],
                                            value = 'ND',
                                            id = 'pf-normal-radiobuttons',
                                        )
                                    ], width = 2),

                                    # Radio button for projection type
                                    dbc.Col([
                                        dbc.Label("Projection:"),
                                        dbc.RadioItems(
                                            options=[
                                                {'label':'Steriographic', 'value':'
                                                    Steriographic_Projection'},
                                                {'label':'Area', 'value':'
                                                    Area_Projection'},
                                            ],
                                            value = 'Steriographic_Projection',
                                            id = 'proj-mode',
                                        )
                                    ], width = 2),

                                    # Radio button for pole figure type
                                    dbc.Col([
                                        dbc.Label("Mode:"),
                                        dbc.RadioItems(
                                            options=[
                                                {'label':'Point␣projection', 'value
                                                    ':'Point␣projection'},
                                                {'label':'Contour', 'value':'
                                                    Contour'},
                                            ],
                                            value = 'Point␣projection',
                                            id = 'pf-mode',
                                        )
                                    ], width = 2),

                                    dbc.Col([
                                        dbc.Row([

                                            # Input for contour levels
                                            dbc.Col([
                                                dbc.Label("Contour␣Levels:"),
                                                dbc.Input(type="text", id = 'pf-
```

```
                                                    contour-levels-input'),
                                                dbc.FormText("Separate␣levels␣with␣
                                                    a␣comma,␣without␣any␣spaces"),
                                            ], width=4),

                                            # Button to re-generate pole figures
                                                based on entered contour levels
                                            dbc.Col([
                                                dbc.Button(id='pf-contour-btn',
                                                            className = 'btn-
                                                                success',
                                                            n_clicks=0,
                                                            children='RE-GENERATE␣
                                                                POLE␣FIGURES',
                                                            style={"margin-top": "
                                                                33px"})
                                            ]),

                                        ]),
                                    ], id = 'pf-contour-levels'),
                                ]),

                                # Loading icon when pole figure generation is in
                                    progress
                                dls.Pacman([
                                    dbc.Row([
                                        dbc.Col([
                                            html.Br(),
                                            dbc.Row([
                                                dbc.Label("Initial␣Pole␣Figure:"),
                                            ],id = 'pf-initial-label'),
                                            dbc.Row([],id = 'pf-image-initial'),
                                            dbc.Row([
                                                dbc.Label("Loaded␣Pole␣Figure:"),
                                            ],id = 'pf-loaded-label'),
                                            dbc.Row([],id = 'pf-image-loaded'),
                                        ])
                                    ])
                                ], color="#eba134",speed_multiplier=2, fullscreen =
                                     True,fullscreen_style={'opacity': '0.7'}),

                            ], body=True, className= 'border-0', id = 'pf-inputs'),
                        ], label = 'Pole␣Figures',
                        tab_id = 'tab-pf'
                        ),

                        # Tab - Inverse Pole Figures
                        dbc.Tab([
                            dbc.Card([
                                dbc.Row([

                                    # Radio button for static or animated IPF
                                    dbc.Col([
                                        dbc.Label("Show:"),
                                        dbc.RadioItems(
                                            options=[
                                                {'label':'Initial,␣Loaded␣&␣
                                                    Trajectory', 'value':'Initial␣
                                                    and␣Loaded␣View'},
                                                {'label':'Animated␣Trajectory', '
```

```
                                                    value’:’Animated’},
                                            ],
                                            value = ’Initial␣and␣Loaded␣View’,
                                            id = ’ipf-radiobuttons’,
                                            inline = True
                                        )
                                    ])
                                ],id = ’ipf-radiobuttons-visibility’),

                                # Checboxes for showing initial/loaded/trajectory IPF
                                dbc.Row([
                                    dbc.Col([
                                        html.Br(),
                                        dbc.Checklist(
                                            options=[
                                                {"label": "Initial", "value": 1},
                                                {"label": "Loaded", "value": 2},
                                                {"label": "Trajectory", "value": 3}
                                            ],
                                            inline=True,
                                            value=[1,2],
                                            id="ipf-checklist",
                                        )
                                    ])
                                ],id = ’ipf-checklist-visibility’),

                                # Loading icon shown when the data is still being generated in the background
                                dbc.Row([
                                    dls.Pacman(dbc.Col([],id = ’ipf-content’),color="#eba134",speed_multiplier=2, fullscreen = True,fullscreen_style={’opacity’: ’0.7’})
                                ]),

                            ], id = ’ipf-card’, body=True, className= ’border-0’)
                        ], label=’Inverse␣Pole␣Figures’, tab_id = ’tab-ipf’),

                        # Tab - ODF
                        dbc.Tab([
                            dbc.Card([
                                    dbc.Row([

                                        # Input field for the ODF levels
                                        dbc.Col([
                                            dbc.Label("ODF␣Levels"),
                                            dbc.Input(placeholder="2,4,8,12,16,20,25,30,35", type="text", id = ’odf-levels-input’),
                                            dbc.FormText("Separate␣levels␣with␣a␣comma,␣without␣any␣spaces"),
                                        ], width=4),

                                        # Button to generate the ODF graph
                                        dbc.Col([
                                            dbc.Button(id=’odf-button-state’,
                                                        className = ’btn-success’,
                                                        n_clicks=0,
                                                        children=’GENERATE␣ODF’,
                                                        style={"margin-top": "33px"
```

```
                                                            })
                                        ])

                                    ]),
                                    html.Br(),
                                    dbc.Row([

                                        # Loading icon shown when the data is still being generated in the background
                                        dls.Pacman([
                                            dbc.Col([
                                                dbc.Row([
                                                    dbc.Label("Initial␣ODF:",id = ’odf-initial-label’),
                                                ]),
                                                dbc.Row(id=’odf-initial’)
                                            ],width = 7),
                                            dbc.Col([
                                                dbc.Row([
                                                    dbc.Label("Loaded␣ODF:",id = ’odf-loaded-label’),
                                                ]),
                                                dbc.Row(id=’odf-loaded’)
                                            ],width = 7),
                                        ], color="#eba134",speed_multiplier=2, fullscreen = True,fullscreen_style={’opacity’: ’0.7’})
                                    ]),

                            ], body=True, className= ’border-0’, id = ’odf-fields’)
                        ],label=’ODF’, tab_id = ’tab-odf’),

                        # Tab - Yield Surface
                        dbc.Tab([
                            dbc.Card([
                                    dbc.Row([

                                        # Checkbox for Initial and/or Loaded Yield Surface
                                        dbc.Col([
                                            dbc.Label("Generate␣yield␣surface␣for␣the␣following:"),
                                            dbc.Checklist(
                                                options=[
                                                    {"label": "Initial", "value": 1},
                                                    {"label": "Loaded", "value": 2},
                                                ],
                                                inline=True,
                                                value=[1,2],
                                                id="ys-checklist",
                                            ),
                                            html.Br()
                                        ], id = ’ys-checklist-visibility’),
                                    ], id=’ys-generate-checklist’),
                                    dbc.Row([
                                        dbc.Col([
                                            dbc.Row([

```

```
                                                # Dropdown field for x-axis of yield surface
                                                dbc.Col([
                                                    dbc.Label("X-axis"),
                                                    dbc.Select(
                                                        id = ’ys-input-x’,
                                                        options =[
                                                            {’label’:"11", ’value’:"11"},
                                                            {’label’:"12", ’value’:"12"},
                                                            {’label’:"13", ’value’:"13"},
                                                            {’label’:"22", ’value’:"22"},
                                                            {’label’:"23", ’value’:"23"},
                                                            {’label’:"33", ’value’:"33"},
                                                            ],
                                                        value = "11"
                                                    )
                                                ], width=2),

                                                # Dropdown field for y-axis of yield surface
                                                dbc.Col([
                                                    dbc.Label("Y-axis"),
                                                    dbc.Select(
                                                        id = ’ys-input-y’,
                                                        options =[
                                                            {’label’:"11", ’value’:"11"},
                                                            {’label’:"12", ’value’:"12"},
                                                            {’label’:"13", ’value’:"13"},
                                                            {’label’:"22", ’value’:"22"},
                                                            {’label’:"23", ’value’:"23"},
                                                            {’label’:"32", ’value’:"32"},
                                                            {’label’:"33", ’value’:"33"},
                                                            ],
                                                        value = "22"
                                                    )
                                                ], width=2),

                                                # Button to generate yield surface
                                                dbc.Col([
                                                    dbc.Button(id=’ys-button-state’,
                                                                className = ’btn-success’,
                                                                n_clicks=0,
                                                                children=’GENERATE␣YIELD␣SURFACE’,
                                                                style={"margin-top"
```

```
                                             : "33px"})
                        ])
                    ])
                ])
            ]),

            # Loading icon shown when the data is still
                 being generated in the background
            dbc.Row([
                dls.Pacman(dbc.Col([],id = 'ys-content'),
                     color="#eba134",speed_multiplier=2,
                     fullscreen = True,fullscreen_style={'
                     opacity': '0.7'})
            ]),
    ], body=True, className= 'border-0', id = 'ys-fields')
],label='Yield␣Surface', tab_id = 'tab-ys'),

# Tab - R-values
dbc.Tab([
    dbc.Card([

            # Checkboxes for generating initial and/or
                 loaded r-values
            dbc.Row([
                dbc.Col([
                    dbc.Label("View␣r-values␣for␣the␣
                         following:"),
                    dbc.RadioItems(
                         options=[
                             {"label": "Initial", "value":
                                  1},
                             {"label": "Loaded", "value":
                                  2},
                        ],
                        value = 1,
                        id = 'rv-checklist',
                        inline = True
                    ),
                    html.Br()
                ], id = 'rv-checklist-visibility'),
            ], id='rv-generate-checklist'),

            dbc.Row([
                dbc.Col([
                    dbc.Row([

                        # Dropdown field for normal axis
                             for r-values
                        dbc.Col([
                            dbc.Label("Normal␣axis"),
                            dbc.Select(
                                id = 'rv-input-normal-axis'
                                     ,
                                options =[
                                    {'label':"001", 'value'
                                         :"001"},
                                    {'label':"100", 'value'
                                         :"100"},
                                    {'label':"010", 'value'
                                         :"010"},
```

```
                                                            ],
                                                        value = "001"
                                                    )
                                                ], width=2),

                                                # Button to generate r-values
                                                dbc.Col([
                                                    dbc.Button(id=’rv-button-state’,
                                                                className = ’btn-success’,
                                                                n_clicks=0,
                                                                children=’GENERATE␣R-VALUES’,
                                                                style={"margin-top" : "33px"})
                                                ])
                                            ])
                                        ])
                                    ]),

                                    # Loading icon shown when the data is still being generated in the background
                                    dbc.Row([
                                        dls.Pacman(dbc.Col([],id = ’rv-content’),color="#eba134",speed_multiplier=2,fullscreen = True,fullscreen_style={’opacity’: ’0.7’})
                                    ]),
                            ], body=True, className= ’border-0’, id = ’rv-fields’)
                        ],label=’R-values’, tab_id = ’tab-rv’),

                    ],
                        id = ’tabs’,
                        active_tab = ’tab-pf’
                    ),
                ]),
            ]),
        ],style=CONTENT_STYLE)
```

Code listing B.5: Code for common layout

```
from dash import html, dcc
from layout.sidebar.sidebar import sidebar
from layout.content.content import content
import dash_loading_spinners as dls
import dash_bootstrap_components as dbc

# This defines the common layout for the application
layout = html.Div([
                sidebar,content,
                dls.Pacman([
                    dcc.Store(id=’store_initial’),
                    dcc.Store(id=’store_loaded’),
                    dcc.Store(id=’store_pf’),
                    ],
                    color="#eba134",
                    speed_multiplier=2,
                    fullscreen=True,
                    fullscreen_style={’opacity’: ’0.7’})
```

```
                ])
```

B.3 Callback Codes

Code listing B.6: Code for sidebar callbacks

```python
from dash import Input, Output
from upload_file import save_file, check_uploaded_file

def get_callbacks_sidebar(app):

    # Callback for showing or hiding input field for texture components
    @app.callback(
    Output(’texture-component-switch-cube’,’style’),
    Output(’texture-component-switch-brass’,’style’),
    Output(’texture-component-switch-s’,’style’),
    Output(’texture-component-switch-goss’,’style’),
    Output(’texture-component-switch-copper’,’style’),
    Output(’texture-component-switch-random’,’style’),
    Output(’texture-component-switch-fiber’,’style’),
    Output(’texture-component-switch-user-input’,’style’),
    Output(’texture-component-label’,’style’),
    Input(’input-grain-ori-state’,’value’),
    )
    def show_hide_element(grain_orientation):
        """This function returns style value for the texture components section.
        The field will only be shown if the selected grain orientation value is ’Input Texture Components’
        Input: Selected value in Grain orientation field
        Output: Styling of texture components section"""
        if grain_orientation == "Input␣Texture␣Components":
            return {’display’: ’block’},{’display’: ’block’},{’display’: ’block’},{’display’: ’block’},{’display’: ’block’},{’display’: ’block’},{’display’: ’block’},{’display’: ’block’},{’display’: ’block’}
        else:
            return {’display’: ’none’,},{’display’: ’none’},{’display’: ’none’},{’display’: ’none’},{’display’: ’none’},{’display’: ’none’},{’display’: ’none’},{’display’: ’none’},{’display’: ’none’}

    # Callback to show or hide Cube texture component input section based on switch
    @app.callback(
    Output(’texture-component-input-cube’,’style’),
    Input(’texture-switch-cube’,’value’),
    Input(’input-grain-ori-state’,’value’),
    )
    def show_hide_element(texture_switch, grain_orientation):
        """This function returns style value for the Cube texture component input section.
        The section will only be shown if Cube toggle button is selected.
        Input: Cube toggle button value
        Output: Styling of the Cube texture component input section"""
        if grain_orientation != ’Input␣Texture␣Components’:
            return {’display’: ’none’}
        else:
            if texture_switch == ["Cube"]:
                return {’display’: ’block’}
            else:
```

```
            return {'display': 'none'}

# Callback to show or hide Goss texture component input section based on switch
@app.callback(
Output('texture-component-input-goss','style'),
Input('texture-switch-goss','value'),
Input('input-grain-ori-state','value'),
)
def show_hide_element(texture_switch, grain_orientation):
    """This function returns style value for the Goss texture component input section.
    The section will only be shown if Goss toggle button is selected.
    Input: Goss toggle button value
    Output: Styling of the Goss texture component input section"""
    if grain_orientation != 'Input␣Texture␣Components':
        return {'display': 'none',}
    else:
        if texture_switch == ["Goss"]:
            return {'display': 'block'}
        else:
            return {'display': 'none'}

# Callback to show or hide Brass texture component input section based on switch
@app.callback(
Output('texture-component-input-brass','style'),
Input('texture-switch-brass','value'),
Input('input-grain-ori-state','value'),
)
def show_hide_element(texture_switch, grain_orientation):
    """This function returns style value for the Brass texture component input section.
    The section will only be shown if Brass toggle button is selected.
    Input: Brass toggle button value
    Output: Styling of the Brass texture component input section"""
    if grain_orientation != 'Input␣Texture␣Components':
        return {'display': 'none'}
    else:
        if texture_switch == ["Brass"]:
            return {'display': 'block'}
        else:
            return {'display': 'none'}

# Callback to show or hide Copper texture component input section based on switch
@app.callback(
Output('texture-component-input-copper','style'),
Input('texture-switch-copper','value'),
Input('input-grain-ori-state','value'),
)
def show_hide_element(texture_switch, grain_orientation):
    """This function returns style value for the Copper texture component input section.
    The section will only be shown if Copper toggle button is selected.
    Input: Copper toggle button value
    Output: Styling of the Copper texture component input section"""
    if grain_orientation != 'Input␣Texture␣Components':
        return {'display': 'none'}
    else:
        if texture_switch == ["Copper"]:
```

```
            return {'display': 'block'}
        else:
            return {'display': 'none'}

# Callback to show or hide S texture component input section based on switch
@app.callback(
Output('texture-component-input-s','style'),
Input('texture-switch-s','value'),
Input('input-grain-ori-state','value'),
)
def show_hide_element(texture_switch, grain_orientation):
    """This function returns style value for the S texture component input section.
    The section will only be shown if S toggle button is selected.
    Input: S toggle button value
    Output: Styling of the S texture component input section"""
    if grain_orientation != 'Input␣Texture␣Components':
        return {'display': 'none'}
    else:
        if texture_switch == ["S"]:
            return {'display': 'block'}
        else:
            return {'display': 'none'}

# Callback to show or hide Random texture component input section based on switch
@app.callback(
Output('texture-component-input-random','style'),
Input('texture-switch-random','value'),
Input('input-grain-ori-state','value'),
)
def show_hide_element(texture_switch, grain_orientation):
    """This function returns style value for the Random texture component input section.
    The section will only be shown if Random toggle button is selected.
    Input: Random toggle button value
    Output: Styling of the Random texture component input section"""
    if grain_orientation != 'Input␣Texture␣Components':
        return {'display': 'none'}
    else:
        if texture_switch == ["Random"]:
            return {'display': 'block'}
        else:
            return {'display': 'none'}

# Callback to show or hide Fiber texture component input section based on switch
@app.callback(
Output('texture-component-input-fiber','style'),
Input('texture-switch-fiber','value'),
Input('input-grain-ori-state','value'),
)
def show_hide_element(texture_switch, grain_orientation):
    """This function returns style value for the Fiber texture component input section.
    The section will only be shown if Fiber toggle button is selected.
    Input: Fiber toggle button value
    Output: Styling of the Fiber texture component input section"""
    if grain_orientation != 'Input␣Texture␣Components':
        return {'display': 'none'}
```

```
        else:
            if texture_switch == ["Fiber"]:
                return {’display’: ’block’}
            else:
                return {’display’: ’none’}

    # Callback to show or hide Fiber texture component input section based on switch
    @app.callback(
    Output(’texture-component-input-user-input’,’style’),
    Input(’texture-switch-user-input’,’value’),
    Input(’input-grain-ori-state’,’value’),
    )
    def show_hide_element(texture_switch, grain_orientation):
        """This function returns style value for the User Input texture component input section.
        The section will only be shown if User Input toggle button is selected.
        Input: User Input toggle button value
        Output: Styling of the User Input texture component input section"""
        if grain_orientation != ’Input␣Texture␣Components’:
            return {’display’: ’none’}
        else:
            if texture_switch == ["User␣Input"]:
                return {’display’: ’block’}
            else:
                return {’display’: ’none’}

    # Calback to return the Spread and Number Grains value for Cube texture component
    @app.callback(
    Output(’spread-cube’, ’value’),
    Output(’number-grains-cube’, ’value’),
    Input(’texture-switch-cube’,’value’),
    Input(’spread-cube’, ’value’),
    Input(’number-grains-cube’, ’value’)
    )
    def show_hide_element(texture_switch, val1, val2):
        """This function returns the spread and number of grains value for Cube texture component
        Input: Cube toggle button value, spread, and number of grains
        Output: Cube spread and number of grains"""
        if texture_switch == []:
            return 0, 0
        else:
            return val1, val2

    # Calback to return the Spread and Number Grains value for Goss texture component
    @app.callback(
    Output(’spread-goss’, ’value’),
    Output(’number-grains-goss’, ’value’),
    Input(’texture-switch-goss’,’value’),
    Input(’spread-goss’, ’value’),
    Input(’number-grains-goss’, ’value’)
    )
    def show_hide_element(texture_switch,val1, val2):
        """This function returns the spread and number of grains value for Goss texture component
        Input: Goss toggle button value, spread, and number of grains
        Output: Goss spread and number of grains"""
```

```
        if texture_switch == []:
            return 0, 0
        else:
            return val1, val2

    # Calback to return the Spread and Number Grains value for Brass texture
        component
    @app.callback(
    Output('spread-brass', 'value'),
    Output('number-grains-brass', 'value'),
    Input('texture-switch-brass','value'),
    Input('spread-brass', 'value'),
    Input('number-grains-brass', 'value')
    )
    def show_hide_element(texture_switch, val1, val2):
        """This function returns the spread and number of grains value for Brass
            texture component
        Input: Brass toggle button value, spread, and number of grains
        Output: Brass spread and number of grains"""
        if texture_switch == []:
            return 0, 0
        else:
            return val1, val2

    # Calback to return the Spread and Number Grains value for Copper texture
        component
    @app.callback(
    Output('spread-copper', 'value'),
    Output('number-grains-copper', 'value'),
    Input('texture-switch-copper','value'),
    Input('spread-copper', 'value'),
    Input('number-grains-copper', 'value')
    )
    def show_hide_element(texture_switch, val1, val2):
        """This function returns the spread and number of grains value for Brass
            texture component
        Input: Copper toggle button value, spread, and number of grains
        Output: Copper spread and number of grains"""
        if texture_switch == []:
            return 0,0
        else:
            return val1, val2

    # Calback to return the Spread and Number Grains value for S texture component
    @app.callback(
    Output('spread-s', 'value'),
    Output('number-grains-s', 'value'),
    Input('texture-switch-s','value'),
    Input('spread-s', 'value'),
    Input('number-grains-s', 'value')
    )
    def show_hide_element(texture_switch, val1, val2):
        """This function returns the spread and number of grains value for S
            texture component
        Input: S toggle button value, spread, and number of grains
        Output: S spread and number of grains"""
        if texture_switch == []:
            return 0, 0
        else:
            return val1, val2
```

```

# Calback to return the Spread and Number Grains value for Random texture component
@app.callback(
Output(’spread-random’, ’value’),
Output(’number-grains-random’, ’value’),
Input(’texture-switch-random’,’value’),
Input(’spread-random’, ’value’),
Input(’number-grains-random’, ’value’)
)
def show_hide_element(texture_switch, val1, val2):
    """This function returns the spread and number of grains value for Random texture component
    Input: Random toggle button value, spread, and number of grains
    Output: Random spread and number of grains"""
    if texture_switch == []:
        return ’-’, 0
    else:
        return val1, val2

# Calback to return the spread, number of grains, Euler angles and axis values for Fiber texture component
@app.callback(
Output(’spread-fiber’, ’value’),
Output(’number-grains-fiber’, ’value’),
Output(’euler-phi1-fiber’, ’value’),
Output(’euler-theta-fiber’, ’value’),
Output(’euler-phi2-fiber’, ’value’),
Output(’axis-x’, ’value’),
Output(’axis-y’, ’value’),
Output(’axis-z’, ’value’),
Input(’texture-switch-fiber’,’value’),
Input(’spread-fiber’, ’value’),
Input(’number-grains-fiber’, ’value’),
Input(’euler-phi1-fiber’, ’value’),
Input(’euler-theta-fiber’, ’value’),
Input(’euler-phi2-fiber’, ’value’),
Input(’axis-x’, ’value’),
Input(’axis-y’, ’value’),
Input(’axis-z’, ’value’),
)
def show_hide_element(texture_switch, val1, val2, tex1, tex2, tex3, axis_x, axis_y, axis_z):
    """This function returns the spread, number of grains, Euler angles and axis values for Fiber texture component
    Input: Fiber toggle button value, spread, number of grains, Euler angles and axis values
    Output: Fiber spread, number of grains, Euler angles and axis"""
    if texture_switch == []:
        return 0, 0, 0, 0, 0, 0, 0, 0
    else:
        return val1, val2, tex1, tex2, tex3, axis_x, axis_y, axis_z

# Calback to return the spread, number of grains and Euler angles values for User Inpur texture component
@app.callback(
Output(’spread-user-input’, ’value’),
Output(’number-grains-user-input’, ’value’),
Output(’euler-phi1-user-input’, ’value’),
Output(’euler-theta-user-input’, ’value’),
```

```
    Output('euler-phi2-user-input', 'value'),
    Input('texture-switch-user-input','value'),
    Input('spread-user-input', 'value'),
    Input('number-grains-user-input', 'value'),
    Input('euler-phi1-user-input', 'value'),
    Input('euler-theta-user-input', 'value'),
    Input('euler-phi2-user-input', 'value')
    )
    def show_hide_element(texture_switch, val1, val2, tex1, tex2, tex3):
        """This function returns the spread, number of grains and Euler angles values for User Input texture component
        Input: User Input toggle button value, spread, number of grains and Euler angles
        Output: User Input spread, number of grains and Euler angles"""
        if texture_switch == []:
            return 0, 0, 0, 0, 0
        else:
            return val1, val2, tex1, tex2, tex3

    # Callback for showing or hiding file upload field
    @app.callback(
    Output('upload-hide','style'),
    Input('input-grain-ori-state','value'),
    )
    def show_hide_element(grain_orientation):
        """This function returns style value for the upload field.
        The field will only be shown if the selected grain orientation value is 'Upload file'
        Input: Selected value in Grain orientation field
        Output: Styling of number of upload field"""
        if grain_orientation == "Upload␣file":
            return {'display': 'block'}
        else:
            return {'display': 'none'}

    # Callback for showing the file is uploaded successfully or not
    @app.callback(
        Output("upload-indicator", "children"),
        Output("upload-indicator", "color"),
        Output("ang-threshold-hide","style"),
        Input("uploaded-file", "filename"),
        Input("uploaded-file", "contents"),
        Input("input-grain-ori-state", "value"),
    )
    def file_upload(uploaded_filename, uploaded_file_content, selected_grain_ori_state):
        """This function returns the status of the uploaded file (successful, error, not yet uploaded).
        Input: uploaded file name, content, selected grain orientation value
        Output: Message showing the status of the uploaded file"""
        if uploaded_filename is not None and uploaded_file_content is not None:
            if selected_grain_ori_state == 'Upload␣file':
                if uploaded_filename.endswith('.ori'):
                    save_file(uploaded_filename, uploaded_file_content)
                    result = check_uploaded_file(uploaded_filename)
                    return result, "success", {'display': 'none'}
                elif uploaded_filename.endswith('.ang'):
                    save_file(uploaded_filename, uploaded_file_content)
                    result = check_uploaded_file(uploaded_filename)
                    return result, "success", {'display': 'block'}
```

```
                else:
                    return "Only␣.ori␣or␣.ang␣files␣are␣allowed.␣Please␣try␣again.", "danger", {'display': 'none'}
            else:
                return "No␣uploaded␣files␣yet.", "warning", {'display': 'none'}
        else:
            return "No␣uploaded␣files␣yet.", "warning", {'display': 'none'}

    # Call back for showing or hiding input fields for custom loading conditions
    @app.callback(
    Output('loading-cond-hide','style'),
    Input('input-loading-condition','value'),
    )
    def show_hide_element(loading_condition):
        """This function returns style value for custom values field.
        The field will only be shown if the selected loading condition value is 'Input Custom Values'
        Input: Selected value in loading condition field
        Output: Styling of custom values field"""
        if loading_condition == "Input␣Custom␣Values":
            return {'display': 'block'}
        else:
            return {'display': 'none'}
```

Code listing B.7: Code for content callbacks

```
from dash import html, dcc, Input, Output, State
from dash.exceptions import PreventUpdate
import dash_bootstrap_components as dbc
import plotly.graph_objects as go
import plotly.express as px
from plotly.subplots import make_subplots
import pandas as pd
import numpy as np
from datasource import create_initial_polycrystal, generate_odf, create_loaded_polycrystal, generate_inversepolefigure, generate_ys, generate_rv
import TexTool_Driver
import dill

def get_callbacks_content(app):

    # Callback to show or hide messagge "Update setup to generate plots" when app is initiliaze for the first time
    @app.callback(
        Output('initial-warning','children'),
        Input('store_initial','data'),
    )
    def render_tab_content(data):
        """This function shows or hides the info message 'Update setup to generate plots'
        Input: Initial data
        Output: Show or hide info message 'Update setup to generate plots'"""
        if data is  None:
            return html.Br(), dbc.Alert("Update␣setup␣to␣generate␣plots.", color="warning")
        else:
            pass

    # Callback to show or hide messagge "Update setup to generate plots" when app is initiliaze for the first time
```

```
    @app.callback(
        Output(’loaded-warning’,’children’),
        Input(’store_initial’,’data’),
        Input(’store_loaded’,’data’),
    )
    def render_tab_content(initial_data, loaded_data):
        """This function shows or hides the info message ’Update setup to generate plots’
        Input: Initial data and Loaded data
        Output: Show or hide info message ’Update setup to generate plots’"""
        if initial_data is not None:
            if loaded_data is  None:
                return html.Br(), dbc.Alert("Update␣setup␣to␣generate␣plots.", color="warning")
            else:
                pass
        else:
            pass

    # Call back to generate and store data for initial polycrystal
    @app.callback(
        Output(’store_initial’,’data’),
        Input(’generate-polycrystal-button’,’n_clicks’),
        State(’input-grain-ori-state’,’value’),
        State(’input-cp-model’, ’value’),
        #Cube
        State(’spread-cube’, ’value’),
        State(’number-grains-cube’,’value’),
        #goss
        State(’spread-goss’, ’value’),
        State(’number-grains-goss’,’value’),
        #brass
        State(’spread-brass’, ’value’),
        State(’number-grains-brass’,’value’),
        #copper
        State(’spread-copper’, ’value’),
        State(’number-grains-copper’,’value’),
        #s
        State(’spread-s’, ’value’),
        State(’number-grains-s’,’value’),
        #random
        State(’spread-random’, ’value’),
        State(’number-grains-random’,’value’),
        #fiber
        State(’spread-fiber’, ’value’),
        State(’number-grains-fiber’,’value’),
        State(’euler-phi1-fiber’,’value’),
        State(’euler-theta-fiber’,’value’),
        State(’euler-phi2-fiber’,’value’),
        State(’axis-x’, ’value’),
        State(’axis-y’, ’value’),
        State(’axis-z’, ’value’),
        #user input
        State(’spread-user-input’, ’value’),
        State(’number-grains-user-input’,’value’),
        State(’euler-phi1-user-input’,’value’),
        State(’euler-theta-user-input’,’value’),
        State(’euler-phi2-user-input’,’value’),
        #upload
        State(’uploaded-file’,’filename’),
```

```
    State('ang-threshold','value'),
prevent_initial_call=True)
def generate_initial_polycrystal(n_clicks, selected_grain_ori_state, cp_model,
                                 spread_cube, number_grains_cube,
                                 spread_goss, number_grains_goss,
                                 spread_brass, number_grains_brass,
                                 spread_copper, number_grains_copper,
                                 spread_s, number_grains_s,
                                 spread_random, number_grains_random,
                                 spread_fiber, number_grains_fiber,
                                 euler_phi1_fiber, euler_theta_fiber, euler_phi2_fiber,
                                 axis_x, axis_y, axis_z,
                                 spread_user_input, number_grains_user_input,
                                 euler_phi1_user_input, euler_theta_user_input, euler_phi2_user_input,
                                 uploaded_filename, threshold):
    """This function generates and stores data for initial polycrystal when generate polycrystal button is clicked
    Input: Generate polycrystal button click, details for selected texture component or uploaded file
    Output: Initial polycrystal data"""
    if n_clicks is None:
        raise PreventUpdate
    else:
        store_initial = create_initial_polycrystal(selected_grain_ori_state,
                                                     cp_model,
                                                     spread_cube, number_grains_cube,
                                                     spread_goss, number_grains_goss,
                                                     spread_brass, number_grains_brass,
                                                     spread_copper, number_grains_copper,
                                                     spread_s, number_grains_s,
                                                     spread_random, number_grains_random,
                                                     spread_fiber, number_grains_fiber,
                                                     euler_phi1_fiber, euler_theta_fiber, euler_phi2_fiber,
                                                     axis_x, axis_y, axis_z,
                                                     spread_user_input, number_grains_user_input,
                                                     euler_phi1_user_input, euler_theta_user_input, euler_phi2_user_input,
                                                     uploaded_filename, threshold)
        return store_initial

# Call back to generate and store data for loaded polycrystal
@app.callback(
    Output('store_loaded','data'),
    Input('load-polycrystal-button','n_clicks'),
    State('store_initial','data'),
    State('input-loading-condition','value'),
```

```
        State('L-XX','value'),
        State('L-XY','value'),
        State('L-XZ','value'),
        State('L-YX','value'),
        State('L-YY','value'),
        State('L-YZ','value'),
        State('L-ZX','value'),
        State('L-ZY','value'),
        State('L-ZZ','value'),
        State('input-von-mises-strain', 'value'),
        State('input-number-steps', 'value'),
    prevent_initial_call=True)
    def generate_loaded_polycrystal(n_clicks, initial_data, selected_loading_state,
                                    L_XX, L_XY, L_XZ, L_YX, L_YY, L_YZ,  L_ZX, L_ZY
                                        , L_ZZ,
                                    tot_von_mises_strain, number_steps):
        """This function generates and stores data for loaded polycrystal when load
             polycrystal button is clicked
        Input: Initial polycrystal data, loading condition selected, global
            velocity gradient values, total von mises stress value, number of steps
             value
        Output: Loades polycrystal data"""
        if n_clicks is None:
            raise PreventUpdate
        else:
            store_loaded = create_loaded_polycrystal(initial_data,
                 selected_loading_state,
                                        L_XX, L_XY, L_XZ, L_YX, L_YY, L_YZ,  L_ZX,
                                            L_ZY, L_ZZ,
                                        tot_von_mises_strain, number_steps)
            return store_loaded

    # Callback to generate and show inverse pole figures
    @app.callback(
        Output('ipf-content','children'),
        Input('store_initial', 'data'),
        Input('store_loaded', 'data'),
        Input('ipf-radiobuttons','value'),
        Input('ipf-checklist','value'),
    prevent_initial_call=True)
    def gen_initial_ipf(initial_data, loaded_data, ipf_rb_value, ipf_cl_value):
        """This function the generates and shows the Inverse Pole Figure
        Input: Initial polycrystal data, Loaded polycrystal data, IPF radio buttons
             value, IPF checkboxes value
        Output: Inverse Pole figure"""
        if initial_data is not None:

            ipf_figure = go.Figure(
                data=[go.Scatter(x = [], y = [], mode='markers', marker=dict(color
                    = '#00256e',size=3), showlegend=False)],
                layout = {'xaxis': {'visible': False,
                                    'showticklabels': False},
                        'yaxis': {'visible': False,
                                    'showticklabels': False}
                }
            )

            ipf_figure.update_layout(xaxis_range=[-0.05, 0.5],yaxis_range=[-0.05,
                 0.5],height=700, width=700)

```

```
            # border for ipf
            x3 = 1./np.sqrt(3.)/(1./np.sqrt(3.)+1.)
            x2 = 1./np.sqrt(2.)/(1./np.sqrt(2.)+1.)
            ipf_trace1=dict(type='scatter',
                                        x=[0., x2],
                                        y=[0., 0.],
                                        marker=dict(color='Black', size =3),
                                        showlegend=False)

            ipf_trace2=dict(type='scatter',
                                        x=[0., x3],
                                        y=[0., x3],
                                        marker=dict(color='Black', size =3),
                                        showlegend=False)

            # plot arc
            a2 = np.arange(0., 0.263, 0.001)
            ipf_trace3=dict(type='scatter',
                                        x=(1.+x2)*np.cos(a2)-1.,
                                        y=(1.+x2)*np.sin(a2),
                                        mode='markers',
                                        marker=dict(color='Black', size =3),
                                        showlegend=False)

            ipf_figure = go.Figure(
                data=[go.Scatter(x = [], y = [], mode='markers', marker=dict(color = '#00256e',size=3), showlegend=False)],
                layout = {'xaxis': {'visible': False,
                                    'showticklabels': False},
                        'yaxis': {'visible': False,
                                    'showticklabels': False}
                }
            )
            ipf_figure.update_layout(xaxis_range=[-0.05, 0.5],yaxis_range=[-0.05, 0.5],height=700, width=700)

            if loaded_data is None:
                ipf_start = generate_inversepolefigure(loaded_data)
                ipf_figure.add_trace(ipf_trace1)
                ipf_figure.add_trace(ipf_trace2)
                ipf_figure.add_trace(ipf_trace3)
                ipf_figure.add_trace(ipf_start)

            elif loaded_data == 1:

                fig_ipf_trajectory, ipf_start, ipf_end, ipf_trajectory = generate_inversepolefigure(loaded_data)

                if ipf_rb_value == 'Animated':
                    ipf_figure = fig_ipf_trajectory

                elif ipf_rb_value == 'Initial␣and␣Loaded␣View':

                    ipf_figure.add_trace(ipf_trace1)
                    ipf_figure.add_trace(ipf_trace2)
                    ipf_figure.add_trace(ipf_trace3)

                    if ipf_cl_value != []:
                        if 1 in ipf_cl_value:
                            ipf_figure.add_trace(ipf_start)
```

```
                        if 2 in ipf_cl_value:
                            ipf_figure.add_trace(ipf_end)
                        if 3 in ipf_cl_value:
                            ipf_figure.add_trace(ipf_trajectory)

            ipf_figure.add_annotation(text=’111’, x=0.37, y=0.38, showarrow=False)
            ipf_figure.add_annotation(text=’101’, x=0.43, y=0, showarrow=False)
            ipf_figure.add_annotation(text=’001’, x=0, y=0.03, showarrow=False)

            return dcc.Graph(figure=ipf_figure)

    # Callback to show or hide UI elements in the Inverse Pole Figure tab
    @app.callback(
        Output(’ipf-card’,’style’),
        Output(’ipf-radiobuttons-visibility’,’style’),
        Output(’ipf-checklist-visibility’,’style’),
        Input(’store_initial’,’data’),
        Input(’store_loaded’,’data’),
        Input(’ipf-radiobuttons’,’value’)
    )
    def show_hide_element(initial_data, loaded_data, ipf_radiobutton_value):
        """This function controls if which UI element in the inverse pole figure tab will be shown or hidden
        Input: Initial polycrystal data, Loaded polycrystal data, IPF radio button value
        Output: IPF card style, IPF radio buttons style, IPF checkboxes style"""
        if initial_data is not None:
            if loaded_data is not None:
                if ipf_radiobutton_value == "Initial␣and␣Loaded␣View":
                    return {’display’: ’block’}, {’display’: ’block’}, {’display’: ’block’}
                else:
                    return {’display’: ’block’}, {’display’: ’block’}, {’display’: ’none’}
            else:
                return {’display’: ’block’}, {’display’: ’none’}, {’display’: ’none’}
        else:
            return {’display’: ’none’}, {’display’: ’none’}, {’display’: ’none’}

    # Callback to generate and show ODF figure
    @app.callback(
        Output(’odf-initial’,’children’),
        Output(’odf-loaded’,’children’),
        Input(’odf-button-state’,’n_clicks’),
        State(’store_initial’,’data’),
        State(’store_loaded’,’data’),
        State(’odf-levels-input’,’value’),
    prevent_initial_call=True
    )
    def generate_initial_odf(n_clicks, initial_data, loaded_data, odf_levels):
        """This function the generates and shows the ODF figure
        Input: Generate ODF button click, Initial polycrystal data, Loaded polycrystal data, ODF levels value
        Output: ODF figure"""

        default_levels = [2,4,8,12,16,20,25,30,35]

        if initial_data is not None:

```

```
            if odf_levels == None:
                odf_levels_list = default_levels
            elif odf_levels == ’’:
                odf_levels_list = default_levels
            elif odf_levels == []:
                odf_levels_list = default_levels
            else:
                odf_levels_strlist = list(odf_levels.split(","))
                odf_levels_list = [eval(i) for i in odf_levels_strlist]

            initial_odf, loaded_odf = generate_odf(odf_levels_list, loaded_data)

            if loaded_data is None:

                return html.Img(src = initial_odf), loaded_odf

            elif loaded_data == 1:

                return html.Img(src = initial_odf), html.Img(src = loaded_odf)

        else:
            return None, None

    # Callback to show or hide UI elements in the ODF tab
    @app.callback(
        Output(’odf-fields’,’style’),
        Input(’store_initial’,’data’),
        Input(’store_loaded’,’data’),
    )
    def show_hide_element(initial_data, loaded_data):
        """This function controls if which UI element in the ODF tab will be shown or hidden
        Input: Initial polycrystal data, Loaded polycrystal data
        Output: ODF level style"""
        if initial_data is not None:
            if loaded_data is not None:
                return {’display’: ’block’}
            else:
                return {’display’: ’block’}
        else:
            return {’display’: ’none’}

    # Callback to generate and show inverse pole figures
    @app.callback(
        Output(’pf-image-initial’,’children’),
        Output(’pf-image-loaded’,’children’),
        Input(’pf-mode’, ’value’),
        Input(’proj-mode’, ’value’),
        Input(’pf-normal-radiobuttons’, ’value’),
        Input(’store_initial’,’data’),
        Input(’store_loaded’,’data’),
        Input(’pf-contour-btn’, ’n_clicks’),
        State(’pf-contour-levels-input’, ’value’),
    prevent_initial_call=True
    )
    def show_hide_element(mode, proj_mode, pf_normal, store_initial, store_loaded, n_clicks, pf_contour_levels):
        """This function the generates and shows the pole figures
        Input: Pole figure tye, projection type, normal value, initial polycrystal data, loaded polycrystal data
```

```
        Output: Initial pole figure, Loaded pole figure"""
        if n_clicks is (None or 0):

            if store_initial is not None:
                with open('orientations_initial.pkl', 'rb') as f:
                    initial_orientations = dill.load(f)
                pf_img_initial = TexTool_Driver.Plot_PF(mode, proj_mode, pf_normal,
                     pf_levels_list=None, EulersAll= initial_orientations)
            else:
                pf_img_initial = None

            if store_loaded is not None:
                with open('orientations_loaded.pkl', 'rb') as f:
                    loaded_orientations = dill.load(f)
                pf_img_loaded = TexTool_Driver.Plot_PF(mode, proj_mode, pf_normal,
                     pf_levels_list=None, EulersAll= loaded_orientations)
            else:
                pf_img_loaded = None

        else:
            if pf_contour_levels == None:
                pf_levels_list = None
            elif pf_contour_levels == '':
                pf_levels_list = None
            elif pf_contour_levels == []:
                pf_levels_list = None
            else:
                pf_levels_strlist = list(pf_contour_levels.split(","))
                pf_levels_list = [eval(i) for i in pf_levels_strlist]

            if store_initial is not None:
                with open('orientations_initial.pkl', 'rb') as f:
                    initial_orientations = dill.load(f)
                pf_img_initial = TexTool_Driver.Plot_PF(mode, proj_mode, pf_normal,
                     pf_levels_list, EulersAll = initial_orientations)
            else:
                pf_img_initial = None

            if store_loaded is not None:
                with open('orientations_loaded.pkl', 'rb') as f:
                    loaded_orientations = dill.load(f)
                pf_img_loaded = TexTool_Driver.Plot_PF(mode, proj_mode, pf_normal,
                     pf_levels_list, EulersAll = loaded_orientations)
            else:
                pf_img_loaded = None

        return html.Img(src = pf_img_initial), html.Img(src = pf_img_loaded)

    # Callback to show or hide UI elements in the PF tab
    @app.callback(
        Output('pf-inputs','style'),
        Input('store_initial','data'),
        Input('store_loaded','data'),
    )
    def show_hide_element(initial_data, loaded_data):
        """This function controls if which UI element in the PF tab will be shown
             or hidden
        Input: Initial polycrystal data, Loaded polycrystal data
        Output: PF input style"""
        if initial_data is not None:
```

```
            if loaded_data is not None:
                return {'display': 'block'}
            else:
                return {'display': 'block'}

        else:
            return {'display': 'none'}

    # Callback reset button clicks for generate pole figure for contour plots
    @app.callback(
        Output('pf-contour-btn', 'n_clicks'),
        Input('pf-mode', 'value')
    )
    def reset_button(pf_mode):
        """This function reset button clicks for generate pole figure for contour plots. This would only reset if pole figure type is NOT contour.
        Input: Pole figure type
        Output: Reset button click"""
        if pf_mode != 'Contour':
            return 0
        else:
            pass

    # Callback to show or hide PF Contour Levels
    @app.callback(
        Output('pf-contour-levels','style'),
        Input('pf-mode','value'),
    )
    def show_hide_element(pf_mode):
        """This function controls to show or hide the Contour levels field in PF tab. This would only show if Contour is selected.
        Input: Pole figure type
        Output: Contour levels field style"""
        if pf_mode != 'Contour':
            return {'display': 'none'}
        else:
            return {'display': 'block'}

    # Callback to generate and show Yield Surface
    @app.callback(
        Output('ys-content','children'),
        Input('ys-button-state','n_clicks'),
        State('store_initial', 'data'),
        State('store_loaded', 'data'),
        State('ys-input-x','value'),
        State('ys-input-y','value'),
        State('ys-checklist', 'value')
    )
    def gen_ys(n_clicks, initial_data, loaded_data, ys_input_x, ys_input_y, ys_cl_value):
        """This function the generates and shows the Yield Surface
        Input: Generate yield surface button click, Initial polycrystal data, Loaded polycrystal data, x-axis value, y-axis value, yield surface checbox value
        Output: Yield surface graph"""
        if initial_data is not None:

            plot_axes = [0,0]
            plot_axes[0] = ys_input_x
            plot_axes[1] = ys_input_y
```

```
            plot_axes_tup = tuple(plot_axes)

            if loaded_data is None:

                ys_xvalues_initial, ys_yvalues_initial, ys_xvalues_loaded, ys_yvalues_loaded= generate_ys(plot_axes_tup, loaded_data)

                df_ys = pd.DataFrame(ys_xvalues_initial)
                df_ys[1] = ys_yvalues_initial
                fig_ys = px.scatter(df_ys, x = 0, y = 1,
                                    labels={
                                        "0": "S"+plot_axes_tup[0],
                                        "1": "S"+plot_axes_tup[1],
                                    })
                fig_ys.update_layout(xaxis_range=[-40,40],yaxis_range=[-40,40], height=600, width=600)
                fig_ys.update_traces(marker=dict(color=’#00256e’))

                return dcc.Graph(figure = fig_ys)

            if loaded_data == 1:
                fig_ys = go.Figure(
                    data=[go.Scatter(x = [], y = [], mode=’markers’, showlegend=False)],
                    layout = {’xaxis’: {’visible’: True,
                                        ’showticklabels’: True},
                            ’yaxis’: {’visible’: True,
                                        ’showticklabels’: True}
                    }
                )
                fig_ys.update_layout(xaxis_range=[-40,40],yaxis_range=[-40,40], height=600, width=600)

                ys_xvalues_initial, ys_yvalues_initial, ys_xvalues_loaded, ys_yvalues_loaded= generate_ys(plot_axes_tup, loaded_data)

                if ys_cl_value != []:
                    if 1 in ys_cl_value:
                        df_ys_initial = pd.DataFrame(ys_xvalues_initial)
                        df_ys_initial[1] = ys_yvalues_initial
                        fig_ys_initial = go.Scatter(x = df_ys_initial[0], y = df_ys_initial[1], mode=’markers’, marker=dict(color = ’#78abde’,size=5), name = "Initial", showlegend=True)
                        fig_ys.add_trace(fig_ys_initial)
                    if 2 in ys_cl_value:
                        df_ys_loaded = pd.DataFrame(ys_xvalues_loaded)
                        df_ys_loaded[1] = ys_yvalues_loaded
                        fig_ys_loaded = go.Scatter(x = df_ys_loaded[0], y = df_ys_loaded[1], mode=’markers’, marker=dict(color = ’#00256e’,size=5), name = "Loaded", showlegend=True)
                        fig_ys.add_trace(fig_ys_loaded)

                fig_ys.update_xaxes(title_text = "S"+ plot_axes_tup[0])
                fig_ys.update_yaxes(title_text = "S"+plot_axes_tup[1])

                return dcc.Graph(figure = fig_ys)

        else:
            return None

```

```

    # Callback to show or hide UI elements in the Yield Surface tab
    @app.callback(
        Output('ys-fields','style'),
        Output('ys-checklist-visibility', 'style'),
        Input('store_initial','data'),
        Input('store_loaded','data'),

    )
    def show_hide_element(initial_data, loaded_data):
        """This function controls if which UI element in the Yield Surface tab will be shown or hidden
        Input: Initial polycrystal data, Loaded polycrystal data
        Output: Yield surface x-axis and y-axis fields style, Yield surface warning message style, Yield surface checbox style"""
        if initial_data is not None:
            if loaded_data is not None:
                return {'display': 'block'},{'display': 'block'}
            else:
                return {'display': 'block'},{'display': 'none'}
        else:
            return {'display': 'none'},{'display': 'none'}


    # Callback to generate and show R-values
    @app.callback(
        Output('rv-content','children'),
        Input('rv-button-state','n_clicks'),
        State('store_initial', 'data'),
        State('store_loaded', 'data'),
        State('rv-input-normal-axis','value'),
        State('rv-checklist', 'value')
    )
    def gen_rvalues(n_clicks, initial_data, loaded_data, normal_axis_input, rv_radiobutton_value):
        """This function the generates and shows the R-values
        Input: Generate r-values button click, Initial polycrystal data, Loaded polycrystal data, normal axis, r-values checkbox value
        Output: R-values graph"""
        if initial_data is not None:

            fig = make_subplots(rows=2, cols=1,shared_xaxes=True)
            angles = np.arange(0,90,5)

            if loaded_data is None:

                rlist_initial, ylist_initial,rlist_loaded, ylist_loaded= generate_rv(normal_axis_input, loaded_data)

                fig.append_trace(go.Scatter(
                    x=angles,
                    y=rlist_initial,
                ), row=1, col=1)

                fig.append_trace(go.Scatter(
                    x=angles,
                    y=ylist_initial,
                ), row=2, col=1)

            if loaded_data == 1:
```

```

                rlist_initial, ylist_initial,rlist_loaded, ylist_loaded=generate_rv(normal_axis_input, loaded_data)

                if rv_radiobutton_value == 1:

                    fig.append_trace(go.Scatter(
                        x=angles,
                        y=rlist_initial,
                    ), row=1, col=1)

                    fig.append_trace(go.Scatter(
                        x=angles,
                        y=ylist_initial,
                    ), row=2, col=1)

                    fig.update_layout(height=600, width=600, showlegend=False, title_text="R-values␣for␣initial␣polycrystal",
                                    xaxis=dict(title=’Angles␣[deg]’),
                                    yaxis1=dict(title=’r-value’),
                                    yaxis2=dict(title=’Yield␣stress␣[MPa]’))

                if rv_radiobutton_value == 2:

                    fig.append_trace(go.Scatter(
                        x=angles,
                        y=rlist_loaded,
                    ), row=1, col=1)

                    fig.append_trace(go.Scatter(
                        x=angles,
                        y=ylist_loaded,
                    ), row=2, col=1)


                    fig.update_layout(height=600, width=600, showlegend=False, title_text="R-values␣for␣loaded␣polycrystal",
                                    xaxis=dict(title=’Angles␣[deg]’, side=’bottom’),
                                    yaxis1=dict(title=’r-value’),
                                    yaxis2=dict(title=’Yield␣stress␣[MPa]’))



            return dcc.Graph(figure = fig)

        else:
            return None

    # Callback to show or hide UI elements in the R-values tab
    @app.callback(
        Output(’rv-fields’,’style’),
        Output(’rv-checklist-visibility’, ’style’),
        Input(’store_initial’,’data’),
        Input(’store_loaded’,’data’),
    )
    def show_hide_element(initial_data, loaded_data):
        """This function controls if which UI element in the R-values tab will be shown or hidden
        Input: Initial polycrystal data, Loaded polycrystal data
```

```
        Output: R-values fields style"""
        if initial_data is not None:
            if loaded_data is not None:
                return {’display’: ’block’},{’display’: ’block’}
            else:
                return {’display’: ’block’},{’display’: ’none’}
        else:
            return {’display’: ’none’},{’display’: ’none’}

    # Callback to show or hide Pole Figure and ODF Initial or Loaded labels
    @app.callback(
        Output(’pf-initial-label’,’style’),
        Output(’pf-loaded-label’,’style’),
        Output(’odf-initial-label’,’style’),
        Output(’odf-loaded-label’,’style’),
        Input(’store_initial’,’data’),
        Input(’store_loaded’,’data’),
    )
    def show_hide_labels(initial_data, loaded_data):
        if initial_data is not None:
            if loaded_data is not None:
                return {’display’: ’block’},{’display’: ’block’},{’display’: ’block’},{’display’: ’block’}
            else:
                return {’display’: ’block’},{’display’: ’none’},{’display’: ’block’},{’display’: ’none’}
        else:
            return {’display’: ’none’},{’display’: ’none’},{’display’: ’none’},{’display’: ’none’}

    # Callback for clearing the initial polycrystal data everytime the generate polycrystal button is clicked
    @app.callback(
        Output(’store_loaded’,’clear_data’),
        Input(’generate-polycrystal-button’,’n_clicks’)
    )
    def clear_store_data(n_click):
        """This function clears the initial polycrystal data when the generate polycrystal button is clicked.
        Input: Generate polycrystal button click
        Output: Trigger to clear initial stored data"""
        if n_click is not None and n_click > 0:
            return True
        else:
            return False
```

B.4 Model Component Codes

Code listing B.8: datasource.py

```
import numpy as np
import crystal_plasticity_module as cp
import odflib
import plotly.express as px
import plotly.graph_objects as go
import pandas as pd
import TexTool_Driver
```

```
import dill

# Storage path for uploaded files
UPLOAD_DIRECTORY = "/uploads/"

FCC_Texture_Comps={ ’Cube’             : ’0,0,0’,
                    ’Rotated␣Cube’     : ’0,0,45’,
                    ’Copper’           : ’90,35,45’,
                    ’Rotated␣Copper’   : ’0,35,45’,
                    ’Inverse␣Copper’   : ’x,x,x’,
                    ’Copper␣Twin’      : ’90,74,45’,
                    ’Brass’            : ’55,90,45’,
                    ’Inverse␣Brass’    : ’0,35,45’,
                    ’Goss’             : ’90,90,45’,
                    ’Rotated␣Goss’     : ’0,90,45’,
                    ’Goss␣Twin’        : ’90,25,45’,
                    ’S’                : ’59,37,63’,
                    ’A’                : ’35,90,45’,
                    ’G/B’              : ’70,90,45’
                     }

# Function that triggers the creation of the initial polycvrystal data and returns
     flag if initial polycrystal data is generated
def create_initial_polycrystal(selected_grain_ori_state,
                               cp_model,
                               spread_cube, number_grains_cube,
                               spread_goss, number_grains_goss,
                               spread_brass, number_grains_brass,
                               spread_copper, number_grains_copper,
                               spread_s, number_grains_s,
                               spread_random, number_grains_random,
                               spread_fiber, number_grains_fiber,
                               euler_phi1_fiber, euler_theta_fiber,
                                    euler_phi2_fiber,
                               axis_x, axis_y, axis_z,
                               spread_user_input, number_grains_user_input,
                               euler_phi1_user_input, euler_theta_user_input,
                                    euler_phi2_user_input,
                               uploaded_filename, threshold):
    """
    Input: Grain Orientaion source, Crystal Plasticity model, Details for selected
         texture component or uplaoded file
    Output: Flag if initial polycrystal data is generated. Returns ’None’ if is is
         NOT generated and returns 1 if it is generated. This also pickles the
         initial polycrystal data and grain orientation.
    """
    if selected_grain_ori_state is None:
        return None

    else:

        if selected_grain_ori_state == "Input␣Texture␣Components":
            number_grains = [number_grains_cube, number_grains_goss,
                 number_grains_brass, number_grains_copper, number_grains_s]
            spread = [spread_cube, spread_goss,spread_brass,spread_copper,spread_s]
            tex_comp = [’Cube’, ’Goss’, ’Brass’, ’Copper’, ’S’]
            tbl = np.array([])

            n = 0
            for tc in tex_comp:
```

```
                row = ’’
                if number_grains[n] != 0:
                    row = ’>␣’ + tc + ’|’ + str(number_grains[n]) + ’|’ + str(
                         spread[n]) + ’|’ + FCC_Texture_Comps[tc] + ’|’
                    tbl=np.append(tbl,row)
                n += 1

            if number_grains_random != 0:
                random = ’Random␣Distribution|’ + str(number_grains_random) + ’|||’
                tbl=np.append(tbl,random)

            if number_grains_user_input != 0:
                user_input = ’>␣UserInput|’ + str(number_grains_user_input) + ’|’ +
                      str(spread_user_input) + ’|’ + str(euler_phi1_user_input) + ’,
                     ’ + str(euler_theta_user_input) + ’,’ + str(
                     euler_phi2_user_input) + ’|’
                tbl=np.append(tbl,user_input)

            orientations = []
            orientations = TexTool_Driver.Generate_BTN(tbl)

            if number_grains_fiber != 0:
                Naxi = int(np.round(number_grains_fiber*(3/5), decimals=0))
                print(’Naxi:’, Naxi)
                euler_angles_deg_fiber = [euler_phi1_fiber, euler_theta_fiber,
                     euler_phi2_fiber]
                axis = [axis_x, axis_y, axis_z]
                print(’axis:’, axis)
                ori_fiber = cp.fibre_spread(euler_angles_deg_fiber, axis,
                     spread_fiber, Naxi, number_grains_fiber)
                if orientations is None:
                    orientations = ori_fiber
                else:
                    orientations = np.append(orientations, ori_fiber, axis = 0)

        elif selected_grain_ori_state == "Upload␣file":
            if uploaded_filename.endswith(’.ori’):
                orientations = []
                file_ori = UPLOAD_DIRECTORY + uploaded_filename
                orientations = np.loadtxt(file_ori, skiprows=3, usecols=(0,1,2))

            elif uploaded_filename.endswith(’.ang’):
                orientations = []
                file_ang = UPLOAD_DIRECTORY + uploaded_filename
                ori1 = np.loadtxt(file_ang, skiprows=103, usecols=(0,1,2))
                ori1 = np.rad2deg(ori1)
                ori2 = []
                threshold_list = np.loadtxt(file_ang, skiprows=103, usecols=(6))

                i = 0
                while i < len(ori1):
                    if threshold_list[i]>=threshold:
                        ori2.append(ori1[i])
                    i+=1

                orientations = ori2

        # No elasticity
        elasticity = None

```

```
        # choose the crystallography of the grains
        crystal_structure = ’FCC_111’

        # Taylor model without grain interactions
        grain_interaction = cp_model

        hardening_law = {’model’                : ’RIGID_PLASTIC’,    # rigid plasticity with a given critical stress
                        ’hardening_parameters’ : [10.],              # CRSS [MPa]
                        ’relax_penalties’      : [0., 0., 0.]}       # [MPa]

        # Create polycrystal
        polycrystal_initial = cp.Polycrystal(crystal_structure, orientations, elasticity, hardening_law, grain_interaction)

        with open(’polycrystal_initial.pkl’, ’wb’) as f:
            dill.dump(polycrystal_initial, f)

        with open(’orientations_initial.pkl’, ’wb’) as f:
            dill.dump(orientations, f)

        return 1

# Function that triggers the creation of the loaded polycvrystal data and returns flag if initial polycrystal data is generated
def create_loaded_polycrystal(initial_data, selected_loading_state,
                            L_XX, L_XY, L_XZ, L_YX, L_YY, L_YZ,  L_ZX, L_ZY, L_ZZ,
                            tot_von_mises_strain, number_steps):
    """
    Input: Initial Polycrystal daya, Loading mode, global velocity gradient, total Von Mises strain, Number of steps
    Output: Flag if loaded polycrystal data is generated. Returns ’None’ if is is NOT generated and returns 1 if it is generated. This also pickles the loaded polycrystal data.
    """
    if initial_data is None:
        pass
    elif initial_data is not None:
        if selected_loading_state is None:
            return None
        else:
            if selected_loading_state == ’Uniaxial␣tension’:
                # prescribed global velocity gradient
                L = np.array([[-0.5,   0.0,  0.0],
                            [ 0.0,  -0.5,  0.0],
                            [ 0.0,   0.0,  1.0]])
                # prescribed stress direction components
                S_direction = None
                # prescribed absolute stress components
                S_absolute = None
                # bool array for indicating known (True) and unknown (False) terms in both vel. gradient and stress tensor
                iL = True
                iS_direction = False
                iS_absolute = False
            elif selected_loading_state == ’Plane-strain␣compression’:
                L = np.array([[1.0,   0.0,  0.0],
                [0.0,   0.0,  0.0],
                [0.0,   0.0, -1.0]])
                # prescribed stress direction components
```

```
                S_direction = None
                # prescribed absolute stress components
                S_absolute = None
                # bool array for indicating known (True) and unknown (False) terms in both vel. gradient and stress tensor
                iL = True
                iS_direction = False
                iS_absolute = False

            elif selected_loading_state == 'Mixed␣(strain+stress)␣boundary␣condition':
                # prescribed global velocity gradient
                L = None
                # prescribed stress direction components
                ang = np.deg2rad(0.)
                sd1 = 2./3.*np.cos(ang) - 1./3.*np.sin(ang)
                sd2 = 2./3.*np.sin(ang) - 1./3.*np.cos(ang)
                S_direction = [[sd1,     0.,    0.],
                            [    0., sd2,    0.],
                            [    0.,     0., -sd1-sd2]]
                # prescribed absolute stress components
                S_absolute = [[ 0., 0., 0.],
                            [ 0., 0., 0.],
                            [ 0., 0., 0.]]

                # bool array for indicating known (True) and unknown (False) terms in both vel. gradient and stress
                iL = False
                iS_direction = [[  True, False, False],
                                [ False,  True, False],
                                [ False, False,  True]]
                iS_absolute = [[False,  True,  True],
                            [ True, False,  True],
                            [ True,  True, False]]

            elif selected_loading_state == 'Input␣Custom␣Values':
                # prescribed global velocity gradient
                L = np.array([[L_XX, L_XY, L_XZ],
                            [L_YX, L_YY, L_YZ],
                            [L_ZX, L_ZY, L_ZZ]])
                L[L==None]=0
                S_direction = None
                S_absolute = None

                # bool array for indicating known (True) and unknown (False) terms in both vel. gradient and stress tensor
                if not np.any(L):
                    iL = False
                else:
                    iL = True

                iS_direction = False
                iS_absolute = False

            if number_steps == None:
                Nsteps = 100
            elif number_steps != None:
                Nsteps = number_steps

            D = deformation_rate(L)
```

```
            t_max = tot_von_mises_strain/(np.sqrt((3/2)*D))
            dt = t_max/Nsteps

            # number of computation steps
            Nsteps = Nsteps
            # time increment
            dt = dt
            # run with elasticity or without
            run_elasticity = False
            # set export options
            results_output_options = {'grain_results'       : ['euler_angles','sliprates','crss','relaxation','stress_loc'],
                                    'polycrystal_results' : ['average_stress','average_slip'],
                                    'number_of_outputs'   : 100}
            dofortran = True

            with open('polycrystal_initial.pkl', 'rb') as f:
                polycrystal_initial = dill.load(f)

            # Load polycrystal
            polycrystal_initial.load(L, S_direction, S_absolute, iL, iS_direction, iS_absolute, Nsteps, dt, run_elasticity, results_output_options, dofortran)

            loaded_orientations = polycrystal_initial.ori_loaded_val

            with open('polycrystal_loaded.pkl', 'wb') as f:
                dill.dump(polycrystal_initial, f)

            with open('orientations_loaded.pkl', 'wb') as f:
                dill.dump(loaded_orientations, f)

            return 1

# Function that generates data and figures for initial and loaded inverse pole figures
def generate_inversepolefigure(loaded_data):
    """
    Input: Loaded polycrystal data
    Output: Inverse pole figure
    """
    # border for ipf
    x3 = 1./np.sqrt(3.)/(1./np.sqrt(3.)+1.)
    x2 = 1./np.sqrt(2.)/(1./np.sqrt(2.)+1.)
    ipf_trace1=dict(type='scatter',
                                x=[0., x2],
                                y=[0., 0.],
                                marker=dict(color='Black', size =3),
                                showlegend=False)

    ipf_trace2=dict(type='scatter',
                                x=[0., x3],
                                y=[0., x3],
                                marker=dict(color='Black', size =3),
                                showlegend=False)
    # plot arc
    a2 = np.arange(0., 0.263, 0.001)
    ipf_trace3=dict(type='scatter',
                                x=(1.+x2)*np.cos(a2)-1.,
```

```
                                y=(1.+x2)*np.sin(a2),
                                mode=’markers’,
                                marker=dict(color=’Black’, size =3),
                                showlegend=False)

    inversepolefigure_start = []
    inversepolefigure_end = []
    inversepolefigure_trajectory = []

    if loaded_data is None:

        with open(’polycrystal_initial.pkl’, ’rb’) as f:
            polycrystal_initial = dill.load(f)

        ipf_data = polycrystal_initial.plot_orientations_plotly(plot_type=’IPF’)
        df_ipf_data = pd.DataFrame(ipf_data)
        df_ipf_data_transposed = df_ipf_data.transpose()

        for i in range(len(df_ipf_data_transposed)):
            inversepolefigure_start.append([i,df_ipf_data_transposed[0][i][0], df_ipf_data_transposed[1][i][0]])

        df_inversepolefigure_start = pd.DataFrame(inversepolefigure_start, columns =[’grain’,’x’, ’y’])

        ipf_start = go.Scatter(x = df_inversepolefigure_start.loc[:,"x"], y = df_inversepolefigure_start.loc[:,"y"], mode=’markers’, marker=dict( color = ’#78abde’,size=3), showlegend=False)

        return ipf_start

    elif loaded_data == 1:

        with open(’polycrystal_loaded.pkl’, ’rb’) as f:
            polycrystal_loaded = dill.load(f)


        ipf_data = polycrystal_loaded.plot_orientations_plotly(plot_type=’IPF’)
        df_ipf_data = pd.DataFrame(ipf_data)
        df_ipf_data_transposed = df_ipf_data.transpose()

        for i in range(len(df_ipf_data_transposed)):
            inversepolefigure_start.append([i,df_ipf_data_transposed[0][i][0], df_ipf_data_transposed[1][i][0]])
            inversepolefigure_end.append([i,df_ipf_data_transposed[0][i][-1], df_ipf_data_transposed[1][i][-1]])
            for j in range(len(df_ipf_data_transposed[0][0])):
                inversepolefigure_trajectory.append([i,j,df_ipf_data_transposed[0][i][j],df_ipf_data_transposed[1][i][j]])

        df_inversepolefigure_trajectory = pd.DataFrame(inversepolefigure_trajectory, columns=[’grain’,’iteration’, ’x’, ’y’])

        # IPF trajectory figure
        fig_ipf_trajectory = px.scatter(df_inversepolefigure_trajectory, x="x", y="y", animation_frame="iteration")
        fig_ipf_trajectory.update_xaxes(showticklabels=False, showgrid=False, visible = False) # Hide x axis ticks
        fig_ipf_trajectory.update_yaxes(showticklabels=False, showgrid=False, visible = False) # Hide y axis ticks
```

```
        fig_ipf_trajectory.update_traces(marker=dict(size=3, color = ’#00256e’))
        fig_ipf_trajectory.add_trace(ipf_trace1)
        fig_ipf_trajectory.add_trace(ipf_trace2)
        fig_ipf_trajectory.add_trace(ipf_trace3)
        fig_ipf_trajectory.add_annotation(text=’111’, x=0.37, y=0.38, showarrow=False)
        fig_ipf_trajectory.add_annotation(text=’101’, x=0.43, y=0, showarrow=False)
        fig_ipf_trajectory.add_annotation(text=’001’, x=0, y=0.03, showarrow=False)
        fig_ipf_trajectory.update_layout(xaxis_range=[-0.05, 0.5],yaxis_range=[-0.05, 0.5],height=700, width=700)

        df_inversepolefigure_start = pd.DataFrame(inversepolefigure_start, columns=[’grain’,’x’, ’y’])
        df_inversepolefigure_end = pd.DataFrame(inversepolefigure_end, columns=[’grain’,’x’, ’y’])
        df_inversepolefigure_trajectory = pd.DataFrame(inversepolefigure_trajectory, columns=[’grain’,’iteration’, ’x’, ’y’])

        ipf_start = go.Scatter(x = df_inversepolefigure_start.loc[:,"x"], y = df_inversepolefigure_start.loc[:,"y"], mode=’markers’, marker=dict(color = ’#78abde’,size=3), showlegend=False)
        ipf_end = go.Scatter(x = df_inversepolefigure_end.loc[:,"x"], y = df_inversepolefigure_end.loc[:,"y"], mode=’markers’, marker=dict(color = ’#00256e’,size=3), showlegend=False)
        ipf_trajectory = go.Scatter(x = df_inversepolefigure_trajectory.loc[:,"x"], y = df_inversepolefigure_trajectory.loc[:,"y"], mode=’markers’, marker=dict(color = ’#00d900’,size=1), showlegend=False)

        return fig_ipf_trajectory, ipf_start, ipf_end, ipf_trajectory

# Function that generates ODF figures
def generate_odf(odf_levels_list, loaded_data):
    """
    Input: ODF levels, Loaded Polycrystal data
    Output: Initial ODF image, Loaded ODF image
    """
    with open(’polycrystal_initial.pkl’, ’rb’) as f:
        polycrystal_initial = dill.load(f)

    odf_data = polycrystal_initial.plot_orientations_plotly(plot_type=’ODF’)
    ori = odflib.Orientations(angles=odf_data)
    odf = odflib.ODF(orientations=ori)
    initial_odf = odf.save_plotly(boundaries=odf_levels_list)

    if loaded_data is None:

        return initial_odf, None

    if loaded_data == 1:

        with open(’polycrystal_loaded.pkl’, ’rb’) as f:
            polycrystal_loaded = dill.load(f)

        odf_data = polycrystal_loaded.plot_orientations_plotly(plot_type=’ODF’)
        ori = odflib.Orientations(angles=odf_data)
        odf = odflib.ODF(orientations=ori)
        loaded_odf = odf.save_plotly(boundaries=odf_levels_list)

        return initial_odf, loaded_odf

```

```
# Function that generates Yield Surface graphs
def generate_ys(plot_axes, loaded_data):
    """
    Input: Axes for the yield surface, Loaded polycrystal data
    Output: Yield Surface graph
    """
    xij = [int(x)-1 for x in plot_axes[0]]
    xij.sort()
    xij = tuple(xij)
    yij = [int(x)-1 for x in plot_axes[1]]
    yij.sort()
    yij = tuple(yij)
    xi, xj = xij
    yi, yj = yij

    if loaded_data is None:

        with open(’polycrystal_initial.pkl’, ’rb’) as f:
            polycrystal_initial = dill.load(f)

        YL_initial, r_initial, num_initial = polycrystal_initial.yield_locus(
             locus_type=’2D’, number_of_points=50, plot_axes=plot_axes)
        ys_xvalues_initial = YL_initial[xi,xj,:]
        ys_yvalues_initial = YL_initial[yi,yj,:]

        ys_xvalues_loaded = 0
        ys_yvalues_loaded = 0

    if loaded_data == 1:

        with open(’polycrystal_initial.pkl’, ’rb’) as f:
            polycrystal_initial = dill.load(f)

        with open(’polycrystal_loaded.pkl’, ’rb’) as f:
            polycrystal_loaded = dill.load(f)


        YL_initial, r_initial, num_initial = polycrystal_initial.yield_locus(
             locus_type=’2D’, number_of_points=50, plot_axes=plot_axes)
        ys_xvalues_initial = YL_initial[xi,xj,:]
        ys_yvalues_initial = YL_initial[yi,yj,:]

        YL_loaded, r_loaded, num_loaded = polycrystal_loaded.yield_locus(locus_type
             =’2D’, number_of_points=50, plot_axes=plot_axes)
        ys_xvalues_loaded = YL_loaded[xi,xj,:]
        ys_yvalues_loaded = YL_loaded[yi,yj,:]

    return ys_xvalues_initial, ys_yvalues_initial, ys_xvalues_loaded,
         ys_yvalues_loaded


# Function that generates R-values graphs
def generate_rv(normal_axis_input, loaded_data):
    """
    Input: Normal axis, Loaded polycrystal data
    Output: R-values graph and Yield stress graph
    """
    angles = np.arange(0,90,5)
    normal_axis_string = list(normal_axis_input)
    normal_axis = [int(x) for x in normal_axis_string]
```

```

    if loaded_data is None:
        with open(’polycrystal_initial.pkl’, ’rb’) as f:
            polycrystal_initial = dill.load(f)

        rlist_initial, ylist_initial = [], []

        if normal_axis == [0,0,1]:
            for ang in np.deg2rad(angles):
                tensile_axis = [np.cos(ang),np.sin(ang),0]
                r, y = polycrystal_initial.getRvalues(tensile_axis, normal_axis)
                rlist_initial.append(r)
                ylist_initial.append(y)

        if normal_axis == [1,0,0]:
            for ang in np.deg2rad(angles):
                tensile_axis = [0,np.sin(ang),np.cos(ang)]
                r, y = polycrystal_initial.getRvalues(tensile_axis, normal_axis)
                rlist_initial.append(r)
                ylist_initial.append(y)

        if normal_axis == [0,1,0]:
            for ang in np.deg2rad(angles):
                tensile_axis = [np.cos(ang),0,np.sin(ang)]
                r, y = polycrystal_initial.getRvalues(tensile_axis, normal_axis)
                rlist_initial.append(r)
                ylist_initial.append(y)

        rlist_loaded = 0
        ylist_loaded = 0

    if loaded_data == 1:

        with open(’polycrystal_initial.pkl’, ’rb’) as f:
            polycrystal_initial = dill.load(f)

        rlist_initial, ylist_initial = [], []

        if normal_axis == [0,0,1]:
            for ang in np.deg2rad(angles):
                tensile_axis = [np.cos(ang),np.sin(ang),0]
                r, y = polycrystal_initial.getRvalues(tensile_axis, normal_axis)
                rlist_initial.append(r)
                ylist_initial.append(y)

        if normal_axis == [1,0,0]:
            for ang in np.deg2rad(angles):
                tensile_axis = [0,np.sin(ang),np.cos(ang)]
                r, y = polycrystal_initial.getRvalues(tensile_axis, normal_axis)
                rlist_initial.append(r)
                ylist_initial.append(y)

        if normal_axis == [0,1,0]:
            for ang in np.deg2rad(angles):
                tensile_axis = [np.cos(ang),0,np.sin(ang)]
                r, y = polycrystal_initial.getRvalues(tensile_axis, normal_axis)
                rlist_initial.append(r)
                ylist_initial.append(y)

        with open(’polycrystal_loaded.pkl’, ’rb’) as f:
```

```
            polycrystal_loaded = dill.load(f)

        rlist_loaded, ylist_loaded = [], []

        if normal_axis == [0,0,1]:
            for ang in np.deg2rad(angles):
                tensile_axis = [np.cos(ang),np.sin(ang),0]
                r, y = polycrystal_loaded.getRvalues(tensile_axis, normal_axis)
                rlist_loaded.append(r)
                ylist_loaded.append(y)

        if normal_axis == [1,0,0]:
            for ang in np.deg2rad(angles):
                tensile_axis = [0,np.sin(ang),np.cos(ang)]
                r, y = polycrystal_loaded.getRvalues(tensile_axis, normal_axis)
                rlist_loaded.append(r)
                ylist_loaded.append(y)

        if normal_axis == [0,1,0]:
            for ang in np.deg2rad(angles):
                tensile_axis = [np.cos(ang),0,np.sin(ang)]
                r, y = polycrystal_loaded.getRvalues(tensile_axis, normal_axis)
                rlist_loaded.append(r)
                ylist_loaded.append(y)

    return rlist_initial, ylist_initial,rlist_loaded, ylist_loaded

# Function to calculate deformation rate from gradient velocity
def deformation_rate(L):
    """
    Input: Global velocity gradient (L) matrix
    Output: D:D
    """
    L_T = np.transpose(L)
    D = (1/2)*(L+L_T)
    D_D = (((D[0,0])**2) + ((D[1,1])**2) + ((D[2,2])**2)) + (2*(((D[0,1])**2)+((D[0,2])**2)+((D[1,2])**2)))
    return D_D
```

Code listing B.9: uploadfile.py

```
import base64
import os
from pickle import TRUE

UPLOAD_DIRECTORY = "/uploads/"

if not os.path.exists(UPLOAD_DIRECTORY):
    os.makedirs(UPLOAD_DIRECTORY)

#This function saves the uploaded file to the upload directory
def save_file(name, content):
    """
    Input: File name, File content
    Output: Decoded and saved file in upload directory
    """
    for file_name in os.listdir(UPLOAD_DIRECTORY):
        file = UPLOAD_DIRECTORY + file_name
        if os.path.isfile(file):
            os.remove(file)
```

```

    # Decode and store a file uploaded with Plotly Dash.
    data = content.encode("utf8").split(b";base64,")[1]
    with open(os.path.join(UPLOAD_DIRECTORY, name), "wb") as fp:
        fp.write(base64.decodebytes(data))

#This function checks is the file is uploaded successfully and returns a message
def check_uploaded_file(uploaded_filename):
    """
    Input: Uploaded filename
    Output: Message whether the file upload was successful or not
    """
    if os.path.exists(os.path.join(UPLOAD_DIRECTORY, uploaded_filename)):
        return "File␣uploaded␣successfully:␣",uploaded_filename
    else:
        return "No␣files␣uploaded."
```

B.5 Main Dash Application

Code listing B.10: app.py

```
import dash
import dash_bootstrap_components as dbc
from dash_bootstrap_templates import load_figure_template
from layout.layout import layout
from layout.content.content_callbacks import get_callbacks_content
from layout.sidebar.sidebar_callbacks import get_callbacks_sidebar
import os

app = dash.Dash(__name__, external_stylesheets=[dbc.themes.LUX])
server=app.server

load_figure_template("lux")
get_callbacks_content(app)
get_callbacks_sidebar(app)

# App Layout
app.layout = layout

# Run app
if __name__ == '__main__':
    # app.run_server(debug=True)
    # Get port and debug mode from environment variables
    port = os.environ.get('dash_port')
    debug = os.environ.get('dash_debug')=="True"
    app.run_server(debug = debug, host="0.0.0.0", port=port)
```

B.6 Added functions in `crystal_plasticity_module.py`

Code listing B.11: plot_orientations_plotly function

```
def plot_orientations_plotly(self, plot_type='IPF', crystal_structure = 'FCC_111',
     plot_what ='trajectory'):
```

```
# This function generates the plotting data for pole figures, inverse pole figures
    or ODF which can be used in Plotly
# Input: Polycrystal grain orientations, Plot type (PF or IPF or ODF), Selected
    crystal structure and plane, Plot trajectory or start
# Ouput: Plotting data for pole figures, inverse pole figures or ODF for the
    polycrystal

    # For Pole Figures
    if plot_type.upper() == 'PF':
        # Lists to store x and y values for plotting; for all normal directions (TD
            , RD and ND)
        x_pf_nd = []
        x_pf_td = []
        x_pf_rd = []
        y_pf_nd = []
        y_pf_td = []
        y_pf_rd = []
        crys_struc = crystal_structure

        for grain in self.grains:
            result = grain.plot_orientation_plotly('PF', crystal_structure =
                crys_struc)
            x_pf_nd.extend(result[0])
            x_pf_td.extend(result[1])
            x_pf_rd.extend(result[2])
            y_pf_nd.extend(result[3])
            y_pf_td.extend(result[4])
            y_pf_rd.extend(result[5])

        # Return x and y data for plotting
        return x_pf_nd, x_pf_td, x_pf_rd, y_pf_nd, y_pf_td, y_pf_rd

    # For Inverse Pole Figures
    elif plot_type.upper() == 'IPF':
        plot_what_input = plot_what
        x_ipf = []
        y_ipf = []
        x_list = [0]
        y_list = [0]
        for grain in self.grains:
            result = grain.plot_orientation_plotly('IPF', plot_what =
                plot_what_input)
            x_list[0]=result[0]
            y_list[0]=result[1]
            x_ipf.extend(x_list)
            y_ipf.extend(y_list)
        return x_ipf, y_ipf

    # For ODF
    elif plot_type.upper() == 'ODF':
        angs = np.zeros((int(self.Ngrains),3))
        for i, g in enumerate(self.grains):
            angs[i,:] = matrix2ang(g.Q)
        return angs
```

Code listing B.12: plot_orientation_plotly function

```
def plot_orientation_plotly(self, plot_type='IPF', crystal_structure = 'FCC_111',
    plot_what='trajectory'):
```

```
# This function generates the plotting data for one grain for pole figures or inverse pole figures which can be used in Plotly
# Input: Grain orientation, Plot type (PF orIPF), Selected crystal structure and plane, Plot trajectory or start
# Ouput: Plotting data for pole figures or inverse pole figures for one grain

    # For Pole Figures
    if plot_type == 'PF':

        if crystal_structure == 'FCC_111':

            planes111 = 1./np.sqrt(3.)*np.array([[ 1., 1., 1.],
                                                [-1., 1., 1.],
                                                [ 1.,-1., 1.],
                                                [-1.,-1., 1.],
                                                [-1.,-1.,-1.],
                                                [ 1.,-1.,-1.],
                                                [-1., 1.,-1.],
                                                [ 1., 1.,-1.],
                                                ])

            planes111rot_ND = np.dot(self.Q.T, planes111.T)
            planes111rot_RD = np.dot(self.Q.T, planes111.T)
            planes111rot_TD = np.dot(self.Q.T, planes111.T)

            proj_nr_111 = 8

            # projection of axis vector to the circle plane to get stereographic projection
            proj111_ND = np.zeros((2,proj_nr_111))
            proj111_RD = np.zeros((2,proj_nr_111))
            proj111_TD = np.zeros((2,proj_nr_111))

            # for ND as normal in PF
            for i in range(proj_nr_111):
                # flip direction if pointing south (negative z)
                planes111rot_ND[:,i] = np.sign(planes111rot_ND[2,i])*planes111rot_ND[:,i]
                proj111_ND[0,i] = planes111rot_ND[1,i]/(planes111rot_ND[2,i]+1)
                proj111_ND[1,i] = -planes111rot_ND[0,i]/(planes111rot_ND[2,i]+1)
            x_ND = proj111_ND[0,:]
            y_ND = proj111_ND[1,:]


            # for RD as normal in PF
            for i in range(proj_nr_111):
                # flip direction if pointing south (negative y)
                planes111rot_RD[:,i] = np.sign(planes111rot_RD[0,i])*planes111rot_RD[:,i]
                proj111_RD[0,i] = planes111rot_RD[1,i]/(planes111rot_RD[0,i]+1)
                proj111_RD[1,i] = planes111rot_RD[2,i]/(planes111rot_RD[0,i]+1)
            x_RD = proj111_RD[0,:]
            y_RD = proj111_RD[1,:]

            # for TD as normal in PF
            for i in range(proj_nr_111):
                # flip direction if pointing south (negative x)
                planes111rot_TD[:,i] = np.sign(planes111rot_TD[1,i])*planes111rot_TD[:,i]
                proj111_TD[0,i] = -planes111rot_TD[2,i]/(planes111rot_TD[1,i]+1)
```

```
                proj111_TD[1,i] = -planes111rot_TD[0,i]/(planes111rot_TD[1,i]+1)
            x_TD = proj111_TD[0,:]
            y_TD = proj111_TD[1,:]

            return x_ND, x_RD, x_TD, y_ND, y_RD, y_TD

        elif crystal_structure == 'FCC_110':

            planes110 = 1./np.sqrt(2.)*np.array([[ 1., -1., 0.],
                                                    [ 1., 1., 0.],
                                                    [ -1., 0., 1.],
                                                    [ 1., 0., 1.],
                                                    [ 0., 1., -1.],
                                                    [ 0., 1., 1.],
                                                    [ -1., 1., 0.],
                                                    [ -1.,-1., 0.],
                                                    [ 1., 0., -1.],
                                                    [ -1., 0., -1.],
                                                    [ 0., -1., 1.],
                                                    [ 0., -1., -1.],
                                                    ])

            planes110rot_ND = np.dot(self.Q.T, planes110.T)
            planes110rot_RD = np.dot(self.Q.T, planes110.T)
            planes110rot_TD = np.dot(self.Q.T, planes110.T)

            proj_nr_110 = 12

            # projection of axis vector to the circle plane to get stereographic projection
            proj110_ND = np.zeros((2,proj_nr_110))
            proj110_RD = np.zeros((2,proj_nr_110))
            proj110_TD = np.zeros((2,proj_nr_110))

            # for ND as normal in PF

            for i in range(proj_nr_110):
                # flip direction if pointing south (negative z)
                if np.sign(planes110rot_ND[2,i]) != 0:
                    planes110rot_ND[:,i] = np.sign(planes110rot_ND[2,i])*planes110rot_ND[:,i]
                proj110_ND[0,i] = planes110rot_ND[1,i]/(planes110rot_ND[2,i]+1)
                proj110_ND[1,i] = -planes110rot_ND[0,i]/(planes110rot_ND[2,i]+1)
            x_ND = proj110_ND[0,:]
            y_ND = proj110_ND[1,:]

            # for RD as normal in PF
            for i in range(proj_nr_110):
                # flip direction if pointing south (negative y)
                if np.sign(planes110rot_RD[0,i]) != 0:
                    planes110rot_RD[:,i] = np.sign(planes110rot_RD[0,i])*planes110rot_RD[:,i]
                proj110_RD[0,i] = planes110rot_RD[1,i]/(planes110rot_RD[0,i]+1)
                proj110_RD[1,i] = planes110rot_RD[2,i]/(planes110rot_RD[0,i]+1)
            x_RD = proj110_RD[0,:]
            y_RD = proj110_RD[1,:]

            # for TD as normal in PF
            for i in range(proj_nr_110):
                # flip direction if pointing south (negative x)
```

```
                if np.sign(planes110rot_TD[1,i]) != 0:
                    planes110rot_TD[:,i] = np.sign(planes110rot_TD[1,i])*planes110rot_TD[:,i]
                proj110_TD[0,i] = -planes110rot_TD[2,i]/(planes110rot_TD[1,i]+1)
                proj110_TD[1,i] = -planes110rot_TD[0,i]/(planes110rot_TD[1,i]+1)

            x_TD = proj110_TD[0,:]
            y_TD = proj110_TD[1,:]

            return x_ND, x_RD, x_TD, y_ND, y_RD, y_TD

        elif crystal_structure == 'FCC_100':

            planes100 = 1./np.sqrt(1.)*np.array([[ 1., 0., 0.],
                                                    [ 0., 1., 0.],
                                                    [ 0., 0., 1.],
                                                    [-1., 0., 0.],
                                                    [ 0.,-1., 0.],
                                                    [ 0., 0.,-1.],
                                                    ])

            planes100rot_ND = np.dot(self.Q.T, planes100.T)
            planes100rot_RD = np.dot(self.Q.T, planes100.T)
            planes100rot_TD = np.dot(self.Q.T, planes100.T)

            proj_nr_100 = 6

            # projection of axis vector to the circle plane to get stereographic projection
            proj100_ND = np.zeros((2,proj_nr_100))
            proj100_RD = np.zeros((2,proj_nr_100))
            proj100_TD = np.zeros((2,proj_nr_100))

            # for ND as normal in PF
            for i in range(proj_nr_100):
                # flip direction if pointing south (negative z)
                if np.sign(planes100rot_ND[2,i]) != 0:
                    planes100rot_ND[:,i] = np.sign(planes100rot_ND[2,i])*planes100rot_ND[:,i]
                proj100_ND[0,i] = planes100rot_ND[1,i]/(planes100rot_ND[2,i]+1)
                proj100_ND[1,i] = -planes100rot_ND[0,i]/(planes100rot_ND[2,i]+1)
            x_ND = proj100_ND[0,:]
            y_ND = proj100_ND[1,:]

            # for RD as normal in PF
            for i in range(proj_nr_100):
                # flip direction if pointing south (negative y)
                if np.sign(planes100rot_RD[0,i]) != 0:
                    planes100rot_RD[:,i] = np.sign(planes100rot_RD[0,i])*planes100rot_RD[:,i]
                proj100_RD[0,i] = planes100rot_RD[1,i]/(planes100rot_RD[0,i]+1)
                proj100_RD[1,i] = planes100rot_RD[2,i]/(planes100rot_RD[0,i]+1)
            x_RD = proj100_RD[0,:]
            y_RD = proj100_RD[1,:]

            # for TD as normal in PF
            for i in range(proj_nr_100):
                # flip direction if pointing south (negative x)
                if np.sign(planes100rot_TD[1,i]) != 0:
                    planes100rot_TD[:,i] = np.sign(planes100rot_TD[1,i])*
```

```
                        planes100rot_TD[:,i]
                proj100_TD[0,i] = -planes100rot_TD[2,i]/(planes100rot_TD[1,i]+1)
                proj100_TD[1,i] = -planes100rot_TD[0,i]/(planes100rot_TD[1,i]+1)

            x_TD = proj100_TD[0,:]
            y_TD = proj100_TD[1,:]

            return x_ND, x_RD, x_TD, y_ND, y_RD, y_TD

        elif crystal_structure == 'FCC_112':

            planes112 = 1./np.sqrt(6.)*np.array([[ 1., 1., 2.],
                                                [ 1.,-1., 2.],
                                                [-1.,-1., 2.],
                                                [-1., 1., 2.],
                                                [-2.,-1., 1.],
                                                [-2., 1., 1.],
                                                [-1.,-2., 1.],
                                                [-1., 2., 1.],
                                                [ 1., 2., 1.],
                                                [ 2., 1., 1.],
                                                [ 2.,-1., 1.],
                                                [ 1.,-2., 1.],
                                                [-1.,-1.,-2.],
                                                [-1., 1.,-2.],
                                                [ 1., 1.,-2.],
                                                [ 1.,-1.,-2.],
                                                [ 2., 1.,-1.],
                                                [ 2.,-1.,-1.],
                                                [ 1., 2.,-1.],
                                                [ 1.,-2.,-1.],
                                                [-1.,-2.,-1.],
                                                [-2.,-1.,-1.],
                                                [-2., 1.,-1.],
                                                [-1., 2.,-1.],
                                                    ])

            planes112rot_ND = np.dot(self.Q.T, planes112.T)
            planes112rot_RD = np.dot(self.Q.T, planes112.T)
            planes112rot_TD = np.dot(self.Q.T, planes112.T)

            proj_nr_112 = 24

            # projection of axis vector to the circle plane to get stereographic
                projection
            proj112_ND = np.zeros((2,proj_nr_112))
            proj112_RD = np.zeros((2,proj_nr_112))
            proj112_TD = np.zeros((2,proj_nr_112))

            # for ND as normal in PF
            for i in range(proj_nr_112):
                # flip direction if pointing south (negative z)
                if np.sign(planes112rot_ND[2,i]) != 0:
                    planes112rot_ND[:,i] = np.sign(planes112rot_ND[2,i])*
                        planes112rot_ND[:,i]
                proj112_ND[0,i] = planes112rot_ND[1,i]/(planes112rot_ND[2,i]+1)
                proj112_ND[1,i] = -planes112rot_ND[0,i]/(planes112rot_ND[2,i]+1)
            x_ND = proj112_ND[0,:]
            y_ND = proj112_ND[1,:]

```

```
            # for RD as normal in PF
            for i in range(proj_nr_112):
                # flip direction if pointing south (negative y)
                if np.sign(planes112rot_RD[0,i]) != 0:
                    planes112rot_RD[:,i] = np.sign(planes112rot_RD[0,i])*
                         planes112rot_RD[:,i]
                proj112_RD[0,i] = planes112rot_RD[1,i]/(planes112rot_RD[0,i]+1)
                proj112_RD[1,i] = planes112rot_RD[2,i]/(planes112rot_RD[0,i]+1)
            x_RD = proj112_RD[0,:]
            y_RD = proj112_RD[1,:]

            # for TD as normal in PF
            for i in range(proj_nr_112):
                # flip direction if pointing south (negative x)
                if np.sign(planes112rot_TD[1,i]) != 0:
                    planes112rot_TD[:,i] = np.sign(planes112rot_TD[1,i])*
                         planes112rot_TD[:,i]
                proj112_TD[0,i] = -planes112rot_TD[2,i]/(planes112rot_TD[1,i]+1)
                proj112_TD[1,i] = -planes112rot_TD[0,i]/(planes112rot_TD[1,i]+1)

            x_TD = proj112_TD[0,:]
            y_TD = proj112_TD[1,:]

            return x_ND, x_RD, x_TD, y_ND, y_RD, y_TD

    # For Inverse Pole Figures
    elif plot_type == ’IPF’:

        x3 = 1./np.sqrt(3.)/(1./np.sqrt(3.)+1.)
        x2 = 1./np.sqrt(2.)/(1./np.sqrt(2.)+1.)

        if plot_what == ’trajectory’:
            try:
                shp = self.results.euler_angles.shape
                if shp[1] > 1:
                    noresults = False
                    plot_data = np.zeros((2,shp[1]))
                    for i in range(shp[1]):
                        Q = ang2matrix(self.results.euler_angles[:,i])
                        plot_data[:,i] = ori2IPF(Q, [0.,0.,1.], (x2, x3))
                    return plot_data
                else:
                    noresults = True
            except:
                noresults = True

            if noresults:
                # plot at least start and end orientation
                plot_data = np.zeros((2,1))
                plot_data[:,0] = ori2IPF(self.Q0, [0.,0.,1.], (x2, x3))
                return plot_data

        elif plot_what == ’start’:
            # plot at least start and end orientation
            plot_data = np.zeros((2,1))
            plot_data[:,0] = ori2IPF(self.Q0, [0.,0.,1.], (x2, x3))
            return plot_data
```

B.7 Added function in `odflib.py`

Code listing B.13: save_plotly function

```python
def save_plotly(self,**kwargs):
    """Write ODF plot to file.  Keyword arguments are passed to odffig.
    The figure instance is returned."""
    o = np.zeros((19,19,19))
    for i in range(19):
        for j in range(19):
            for k in range(19):
                o[i,j,k] = self._data[k,j,i]
    buf = io.BytesIO() # in-memory files
    fig = odffig(o, **kwargs)
    fig.savefig(buf, format = "png") # save to the above file object
    plt.close()
    data = base64.b64encode(buf.getbuffer()).decode("utf8") # encode to html elements
    return "data:image/png;base64,{}".format(data)
```

B.8 Updated function in `TexTool_Driver.py`

Code listing B.14: Plot_PF function

```python
def Plot_PF(mode, proj_mode, pf_normal, pf_levels_list, EulersAll):
    print (’Number␣of␣data␣points␣(Euler␣Angles):␣’, len(EulersAll))
    PF_planes = [[1,0,0],[1,1,0],[1,1,1]]
    fig=plt.figure(figsize=(18,6),dpi=200)
    cmapx = copy.copy(matplotlib.cm.get_cmap(’rainbow’))
    cmapx.set_under(’w’)
    # Density Mesh
    MS=20 # Number of Grids in each direction for Density mesh
    MSxy= np.linspace(-1,1,MS+1)
    MSx,MSy = np.meshgrid(MSxy,MSxy)

    for i,PF_plane in enumerate(PF_planes):
        ax=fig.add_axes([i*0.33,0,0.33, 1])
        Points=Funcs.Projection(EulersAll,PF_plane,proj_mode, pf_normal)
        if mode==’Point␣projection’:
            ax.scatter(Points[0],Points[1],marker=’.’,s=5)
            ax.axis(’off’)
            ax.set_aspect(’equal’)
            circle1 = plt.Circle((0, 0), 1, color=’black’, fill=0)
            ax.add_artist(circle1)
            ax.set_xlim([-1.3,1.3]) ; ax.set_ylim([-1.3,1.3])
            Plane_txt=’(’ + str(PF_plane[0]) + str(PF_plane[1]) + str(PF_plane[2]) + ’)’
            ax.text(-0.1,1.2,Plane_txt,fontsize=14)
            if pf_normal == ’ND’:
                ax.text(-0.03,1.02,’RD’,fontsize=12) ; ax.text(1.02,-0.02,’TD’, fontsize=12)
            if pf_normal == ’TD’:
                ax.text(-0.03,1.02,’ND’,fontsize=12) ; ax.text(1.02,-0.02,’RD’, fontsize=12)
            if pf_normal == ’RD’:
```

```
                ax.text(-0.03,1.02,'TD',fontsize=12) ; ax.text(1.02,-0.02,'ND',fontsize=12)
        if mode=='Contour':
            xm,ym,Density= Funcs.Points2Dens(Points,MS,MSx,MSy)
            maxv= np.amax(Density) ; minv= np.amin(Density) ; avv= np.average(Density)
            print('Plane,␣max,min,av␣denisty:',PF_plane, maxv,minv,avv*np.sqrt(2))
            if pf_levels_list == None:
                max_logspace = np.log(maxv)/np.log(2)
                lvls = np.logspace(0,max_logspace,num=7,endpoint = True, base = 2)
                lvls_rounded = np.round(lvls, decimals = 2)
                print('lvls:', lvls)
                print('lvls␣rounded:', lvls_rounded)
            if pf_levels_list != None:
                lvls_rounded = pf_levels_list
                print('lvls␣rounded:', lvls_rounded)
            cs=ax.contourf(xm,ym,Density, levels=lvls_rounded, vmin=0, cmap=cmapx, extend='both')
            cbar=plt.colorbar(cs, ax=ax, shrink=0.6)
            cbar.ax.tick_params(labelsize=10)
            line_colors = ['black' for l in cs.levels]
            cl = ax.contour(cs, levels=lvls_rounded, colors=line_colors)  #or cmap=cmapx
            ax.patch.set_facecolor('xkcd:mint␣green')
            ax.axis('off')
            ax.set_aspect('equal')
            circle1 = plt.Circle((0, 0), 1, color='black', fill=0)
            ax.add_artist(circle1)
            ax.set_xlim([-1.1,1.1]) ; ax.set_ylim([-1.1,1.1])
            Plane_txt='(' + str(PF_plane[0]) + str(PF_plane[1]) + str(PF_plane[2]) + ')'
            ax.set_title(Plane_txt, fontsize=14, y=1.03)
            if pf_normal == 'ND':
                ax.text(-0.03,1.02,'RD',fontsize=12) ; ax.text(1.02,-0.02,'TD',fontsize=12)
            if pf_normal == 'TD':
                ax.text(-0.03,1.02,'ND',fontsize=12) ; ax.text(1.02,-0.02,'RD',fontsize=12)
            if pf_normal == 'RD':
                ax.text(-0.03,1.02,'TD',fontsize=12) ; ax.text(1.02,-0.02,'ND',fontsize=12)
        if mode=='Contour':
            circ = patches.Circle((0, 0), 1, transform=ax.transData)
            for coll in cs.collections:
               coll.set_clip_path(circ)
            for coll in cl.collections:
               coll.set_clip_path(circ)
    buf = io.BytesIO() # in-memory files
    fig.savefig(buf, format = "png") # save to the above file object
    plt.close()
    data = base64.b64encode(buf.getbuffer()).decode("utf8") # encode to html elements
    return "data:image/png;base64,{}".format(data)
```

B.9 Updated function in `TexTool_Funcs.py`

Code listing B.15: Projection function

```
def Projection (Eulers,PF_plane,proj_mode, pf_normal):

      Family= GetFamilyOfPlanes(PF_plane)
      Prj=np.zeros(2)
      Prjs= np.empty((0,2))

      for Euleri in Eulers:
            Qi=Euler2Rot(Euleri,'deg')
            for planei in Family:
                  # Rotating plane normal vector with Qi
                  planei=np.matmul(planei,Qi)
                  if pf_normal == 'ND':
                        if planei[2]>=0:
                              # Conversion from Cartesian to Spherical coordinates
                              alpha = np.arctan2(planei[1],planei[0])
                              psi = np.arccos(planei[2])
                              Prj[0] , Prj[1] = Project(alpha, psi, proj_mode)
                              Prjs = np.append(Prjs, np.array([Prj]), axis=0)
                  elif pf_normal == 'TD':
                        if planei[0]>=0:
                              # Conversion from Cartesian to Spherical coordinates
                              alpha = np.arctan2(planei[1],planei[2])
                              psi = np.arccos(planei[0])
                              Prj[0] , Prj[1] = Project(alpha, psi, proj_mode)
                              Prjs = np.append(Prjs, np.array([Prj]), axis=0)
                  elif pf_normal == 'RD':
                        if planei[1]>=0:
                              # Conversion from Cartesian to Spherical coordinates
                              alpha = np.arctan2(planei[2],planei[0])
                              psi = np.arccos(planei[1])
                              Prj[0] , Prj[1] = Project(alpha, psi, proj_mode)
                              Prjs = np.append(Prjs, np.array([Prj]), axis=0)

      return np.transpose(Prjs)
```

Norwegian University of
Science and Technology