NTNU
Kunnskap for en bedre verden

Department of Materials Science and Engineering

Crystal Plasticity Finite Element Simulations using FEpX Open-Source Software

Peder Jørgen Nagelsaker Lexau

Master Thesis in Material Technology

Supervisors: Tomáš Mánik & Bjørn Holmedal

March, 2023

Preface

This thesis is delivered as a part of the masters program ”Industrial Chemistry and Biotechnology” at the Norwegian University of Science and Technology. The work was carried out at the Department of Materials Science and Engineering in Trondheim under the supervision of researcher Tomáš Mánik and professor Bjørn Holmedal.

The project was performed using multiple tools. A computer running the Linux operating system *Ubuntu 22.04.1 LTS* was provided by Tomáš Mánik, NTNU, and accessed with the screen sharing program *AnyDesk*. The opportunity to use the NTNU IDUN cluster was given, which was critical to perform the simulations in this thesis. The tessellations, meshes, pole figures and pictures of simulation cubes were created using the software package *Neper*. Simulations on these meshes were done with the finite element software package for polycrystal plasticity called *FEpX*, and the powerful multiphysics FEA software *Abaqus*.

All figures and illustrations have unless otherwise stated been created by the author.

I would like to extend my thanks to my supervisors for their support and directions. Thanks to the FEpX Github and the team maintaining IDUN for answering any questions. At last i would also thank my family for encouraging me through the project.

Peder Jørgen Nagelsaker Lexau

Trondheim, March 2023

Abstract

Our understanding of material properties is critical to our modern life. This comprehension has historically been found through physical experiments on materials and their macrostructures, but is difficult to achieve for the microstructures of the material. The last forty years computational simulations have been increasingly used to test the macrostructural properties of materials. As better computers have become available, the simulations have also become cheaper and less time-consuming. With better computers it has become possible to simulate the fairly expensive process of straining structures on a much smaller scale. The Crystal Plasticity Finite Element Method has risen as a good substitute for experimental data on the microstructure of materials. Its computations are fairly expensive and so sees limited use, but has become important for our understanding of the impact of micro-deformations on material failure.

Today Abaqus is one of the most widely used finite element commercial programs that are able to call crystal plasticity material models via user-defined subroutines. It is closed source, and have a relatively high cost. Free, open source alternatives have appeared the last years, and those bring with them obvious advantages. Free software is more available to the public, less-economically powerful institutions and companies, and can be of assistance to a bigger user pool. Open source means that the code the software runs on is available for the users to see, learn from, and improve upon. It can be adjusted by individuals for their specific cases, and also be merged into the software if found to be an improvement. Such software has the possibility of becoming shared projects across fields and institutions. FEpX is a finite element software package for polycrystal plasticity that is both free and open source. Comparing Abaqus to FEpX can be useful for discovering advantages and drawbacks with both programs.

Oppsummering

Vår forståelse av materialegenskaper er kritisk for vårt moderne liv. Denne forståelsen har historisk blitt laget gjennom fysiske eksperimenter på materialer og deres makrostruktur, men den er vanskelig å oppnå for materialets mikrostruktur. De siste førti årene har komputersimuleringer blitt brukt i økende grad for å teste materialers makrostrukturelle egenskaper. Mens bedre komputere har blitt tilgjengelig har disse simuleringene blitt mindre intensive og tidsforbruket har gått ned. Med bedre komputere har det blitt mulig å simulere den intensive prosessen å anstrenge strukturer på mye midre skala. Krystallplastisk finite-element metoder har dukket opp som et godt alternativ til eksperimentell data på mikrostrukturen til materialer. Komputasjonene er ganske intensive, så den ser begrenset bruk, men har blitt viktig for vår forståelse av innvirkningen av mikrodeformasjoner på materialskade.

I dag er Abaqus et av de mest brukte kommersielle programmene for finite-element metoder som kan kalle krystallplastisk materialmodeller via brukerdefinerte subrutiner. Den er closed-source, og har en relativt høy pris. Gratis open-source alternativer eksisterer, og de bringer med seg åpenbare fordeler. Gratis software er mer tilgjenelig for folk flest, mindre økonomisk stekre institusjoner og bedrifter, og kan bli brukt av en større brukermengde. Open-source betyr at koden softwaren kjører på er tilgjengelig for brukeren å se, lære fra, og forbedre. Den kan bli justert av individer for deres spesifikke tilfeller og kan bli tatt inn i softwaren om det blir oppdaget at den er en forbedring. Slik software har muligheten til å bli delte prosjekter mellom forskjellige felter og instutisjoner. FEpX er en finite-element software pakke for polykrystalplastiskitet som er både gratis og open-source. Å sammenlikne Abaqus og FEpX kan være nyttig for å oppdage fordeler og ulemper med begge programmene.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Polycrystalline materials have always been important to create products in everyday life. As knowledge about their properties and behavior have become better understood, better materials have become available. Through experiments, the texture of the material, in other words the size, quantity and orientation of its grains, have been found to have a big impact on the material's mechanical properties [2]. Simulating deformations on such polycrystalline structures have become an important source of data about the behavior of such materials under stress [13].

It has long been known long that the plastic deformation in a loaded crystalline material is highly dependent on grain structure, grain orientations, morphology, presence of constitutive particles, and others features that may be present in the microstructure [2]. Mechanical properties are dependent on the way plasticity takes place in the material, which usually leads to a directional dependency, hence, the plastic anisotropy. To a large extent, this is dictated by the texture [13]. A very heterogeneous nature of the plastic deformation has also been confirmed experimentally[4] [https://doi.org/10.1007/s11340-012-9685-2]. Micro-deformations may lead to material failure at lower strains than expected and reduce the life-span of critical parts. This is especially true for smaller components, as they have fewer grains. Fewer grains means the properties of the material are more dependent on the properties of the grains that make up the material. Through experiments and simulations, the areas of such components that are especially exposed for deformation can be found.

One of the most important methods for simulating strain and stress fields in microstructures is the Crystal Plasticity Finite-Element Method. To this day it has seen relatively little use in industrial context, as it is both resource and time consuming. By increasing the coarseness of the mesh describing the structure, and simplifying the grain structure this "price" can be reduced. Large computational time reduction can be achieved by applying very efficient spectral solver instead of the finite element method [12]. When only the material texture and the global plastic properties of a polycrystalline aggregate are of interest, as e.g. plastic anisotropy, yield surfaces, formability, so-called mean field crystal plasticity models are computationally much cheaper alternatives (see e.g. [9], [KIA ZHANG 2015 PAPER], [ALAMEL TYPE 3 PAPPER]). Mean-field models are of a statistical nature and do not bear any detailed spatial information of a given 3D microstrucutre, and thus provide no full-field results. In this thesis, the exploitation of the full-field Crystal Plasticity Finite-Element Method is in focus.

Abaqus is one such program capable of running such methods. Abaqus itself is a finite element software to solve a boundary value problem of a component, to which e.g. a static or a dynamic load is applied [15]. Via Fortran-based subroutines, Abaqus provides very important interfaces for programming large variety of fully user-defined phenomena, which are then built and recompiled together. Specifically, using a so called user-defined material subroutine (UMAT/VUMAT) allows to bridge crystal plasticity material models with the finite element method in Abaqus. Recently, Manik et al. [8] has implemented a robust and efficient algorithm for the crystal plasticity via a UMAT in the Abaqus/Standard software. Unfortunately, Abaqus has a subscription fee, which makes it unavailable for some users, and increases the threshold for others to start making use of it. Apart from the user-defined subroutines, it is a closed source software, stopping the public from seeing how it implements its finite-element method for solving the global equilibrium system of equations, and preventing experimental changes and alternative algorithms from being implemented into it. The ability to do so could help increase the effectiveness of the program, and discover better ways of performing the simulations. For example, one of the desired options to have a user-friendly access to is remeshing. As modeling of the solid mechanics is done using a deforming mesh, there are limits to how much the mesh can deform before it becomes too distored, causing the finite element method to give wrong results or the whole calculation to terminate.

Testing an alternative open-source software package that are also capable of performing the Crystal Plasticity Finite-Element Method is therefore something that should be tried. Today, several open-source software implementing crystal plasticity into a finite element method are available, Moose, WARP3D, PRISMS, FEpX [3]. DAMASK software [12] is of another kind, as it employs a spectral method using the Fast Fourier transformation, instead of the finite element method. Due to this, it can be several orders of magnitude faster.

In this work, FEpX [3] was chosen to be further explored and compared as an finite elements method-driven alternative to Abaqus, due to the following reasons:

1. It is an implicit finite element code, similarly as Abaqus/Standard, so the comparison can be performed

2. It is programmed in Fortran 90, which can make it relatively easy to try to implement the user-material subroutine developed by Manik et al. [8]

3. It has a close link to NEPER, an open-source software for both grain morphology and mesh generation, as well as used for post-processing of the FEpX results. Hence, possibly a well-integrated and on-the-fly remeshing could be achieved, which is lacking in Abaqus when user-defined material subroutines are applied and is necessary for simulating rolling textures up to medium and high strains

4. It has relative large user-base and large number of scientific publications, where FEpX is employed

5. It has well-organize web page and seemingly well-described documentation with tutorials, which can contribute to a steep learning curve

FEpX is a free, open-source program which allows the users to tweak and adjust its algorithm as needed. An experienced researcher may also find use of the program by substituting parts of its algorithm with their own. To do this however, an overview and understanding of the code-base is needed. This thesis will strive to create such an overview of FEpX, and see if a its finite-element methods for crystal plasticity can feasibly be interfaced with alternative methods for crystal plasticity, as [8].

As mentioned above, FEpX is dependent on another program, Neper [10], to create user-defined polycrystal structures using Voronoi tesselations, generate meshes, also with the possibility for export in Abaqus. NEPER is also used for post-processing of the results from FEpX, generation and visualization of pole figures, and figures of the representative volume element showing large variety of field results. Gaining an understanding of this software is therefore necessary to be able to use FEpX in any meaningful way.

It is, however, important to test FEpX's ability to run different simulation cases to determine, how well it can perform, and map any potential limitations of the software. If it is to be used as an alternative to Abaqus, users should be aware of any such limitations. User interaction friendliness, performance, simulation results, and possible limitations and pitfalls are all very important aspects when comparing FEpX to Abaqus and will be addressed in this thesis.

1.2 Objectives

The scope of the thesis is defined in this section. The questions and objectives listed will be revisited in Chapter 6.

- What is the code-structure of FEpX, and how easy is it to substitute it with other algorithms for crystal plasticity?
- What are the advantages of using FEpX compared to Abaqus?
- What are the limitations of FEpX?

To answer these questions, the given objectives will be fulfilled.

1. Create a overview of the FEpX code-base.
2. Understand how Neper works, and use it to create tessellations and meshes for FEpX and Abaqus.
3. Use FEpX to simulate different standard cases and try to find any advantages or limitations with the program.
4. Run similar simulations on FEpX and Abaqus and compare the results.
5. See how FEpX handles simulations of a plane-strain compression. Which strains can be achieved before the mesh get distorted and hinder further progress of the simulation?

1.3 Contributions

This thesis contributes to the field of material simulations, and more specifically Finite-Element Crystal Plasticity Methods. The following points are introduced in this paper:

- Use of Neper to generate polycrystal structures, and meshing them.
- An overview of the code-base of FEpX.
- Several material deformation tests using FEpX.
- A comparison of similar simulations run on both FEpX and Abaqus.

By creating an overview of the code-base of FEpX, it is easier to see which parts and modulations are responsible for which logic. This overview makes it simpler to do changes and updates at the correct areas of the code. To substitute other algorithms in place for the native ones, it is important to know where the substitutions should connect to the native code, and what variables are expected in and out.

By testing FEpX in different simulation scenarios, advantages and limitations of FEpX have been found. Since FEpX is thightly bound to Neper, its ability to generate polycrystal structures and meshes have also been tested. How the features of FEpX impact the user experience will be described in this thesis.

To see if FEpX is to be used as a substitute or replacement for Abaqus, their performance and results have been compared. By running similar simulation cases on the two programs, a rudimentary comparison has been made.

1.4 Outline

The thesis is divided into six chapters. Chapter 1 introduces the motivation, objective, and contributions of the thesis, and Chapter 2 describes the relevant theory. Chapter 3 explains the software used and their function, while Chapter 4 contains the recipe for installing them. After that is a section about the simulations, their parameters and the files used. In Chapter 5 The results of the different simulations are presented and compared, and later discussed. Lastly Chapter 6 concludes the thesis and brings up possible future work to expand upon what has been performed in this thesis.

Chapter 2

Theory

2.1 Texture

Texture is the statistical orientation of the polycrystalline structures in a material. Most materials like metals, ceramics, and minerals are made up of grains, or crystallites, with their specific crystal structure. Orientation in this context is the relation between the atomic planes of the crystal structure and a fixed reference. The orientation of these compared to the sample affect many of the material's properties. The grain direction in almost all materials are not random, but have a predisposition for some orientations. These orientations may affect the material properties by up to 50% [5]. It is therefore highly important to understand and predict the effect of crystallite orientation in material technology.

The understanding of texture dives into how changing the texture affects the material's properties, but also how changes in the material may affect the texture. This can be done by measuring the orientations of a physical material before and after it is exposed for stress. The ways to analyse the texture of materials have until recently been focusing on

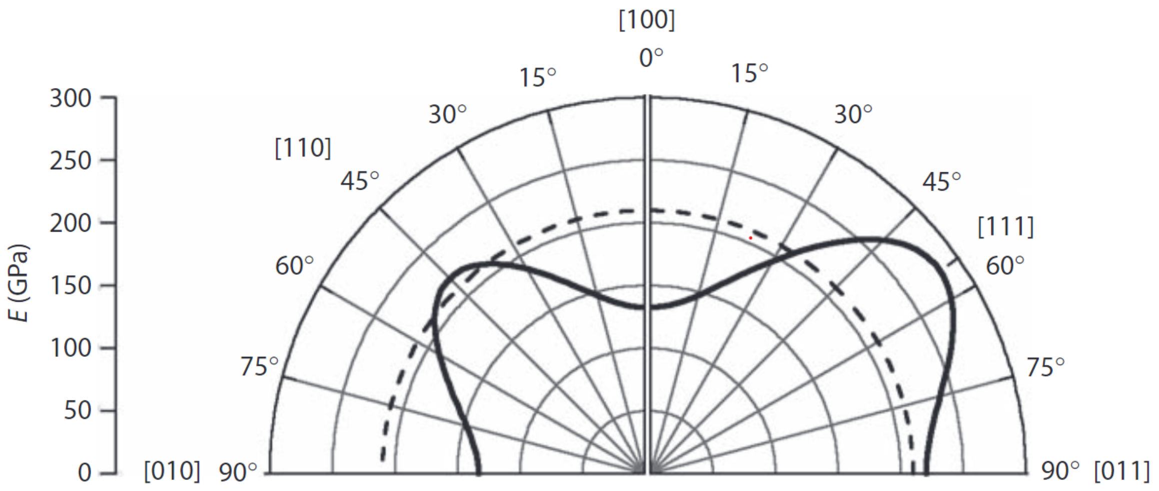


Figure 1: Elastic (Young's) modulus E of single-crystalline iron as a function of crystal direction. Clearly, the values strongly differ from the well-known bulk modulus of 210 GPa (dotted line), which is only obtained in isotropic, that is, texture-free material. Image borrowed from [5]

the macrotexture, by using x-ray or neutron diffraction. With transmission and scanning electron microscopes, it has become possible to analyse individual orientations related to the microstructure of the material. These orientations of individual grains affecting smaller parts of the material, instead of the material as a whole, is called microtexture. By understanding the effects of microtexture, cracks, fatigue and failure in vulnerable parts of the material can be better prevented.

As analysing the orientation of individual grains in a material can quickly become expensive and cumbersome, alternative ways to understand the microtexture have been created. Today there are several different modulation programs where properties of a microstructure can be simulated before, during, and after being exposed to a force [5].

2.1.1 Pole Figure

When talking about the importance of knowing the orientation of grains, it is also vital to be able to easily show their orientation distribution. To describe a 3D vector in a crystal, it can be imagined as a point on a reference sphere. That is a sphere with standard radius of 1 encompassing the crystal. Any 3D vector in the circle will move towards a point on the sphere surface, and so all vectors on the crystal can be presented as points, or poles, on the sphere. If the sphere is instead attached to an external reference, for example the sample, the crystallographic orientation of the crystals in the sample can be presented in this fashion. The position of the pole on the sphere is dependent on two angles. Angle α which determines the distance from the north pole, in other words its "latitude", and angle β which determines the pole's rotation around the sphere, its "longitude". Mark that the crystals can still rotate around its "pole" and so it still has one degree of freedom.

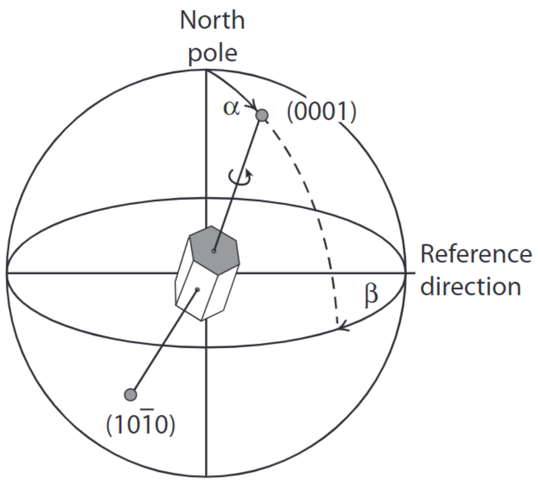


Figure 2: A reference sphere with a hexagonal crystal inside. The orientation of the basal plane (0001) is described on the sphere with the angles α and β. As the crystal can rotate around the pole, to get an unambiguous orientation, the pole position of another plane (10$\bar{1}$0) is needed. Image borrowed from [5]

A 3D sphere is not the best way to present data, and so the representation of the 3D orientation information is projected onto a 2D plane. This is called a stereographic projection, where the sphere is projected on its equatorial plane. The strength of such a projection is that equal angles between lines on the sphere surface will still be equal after being projected on the plane. The 2D projection of the reference sphere creates a circle named a *pole figure*. When creating a pole figure it is important to reference it to the sample's orientation. For rolling symmetry the reference directions of the sample are the sheet normal direction ND, the rolling direction RD, and the transverse direction TD.

The pole figures relation to these directions vary, but often the north pole, or the center of the pole figure, is set as the ND, while RD is set where β is 0°. Other deformation modes will require other references, but a 3D coordinate system is required.

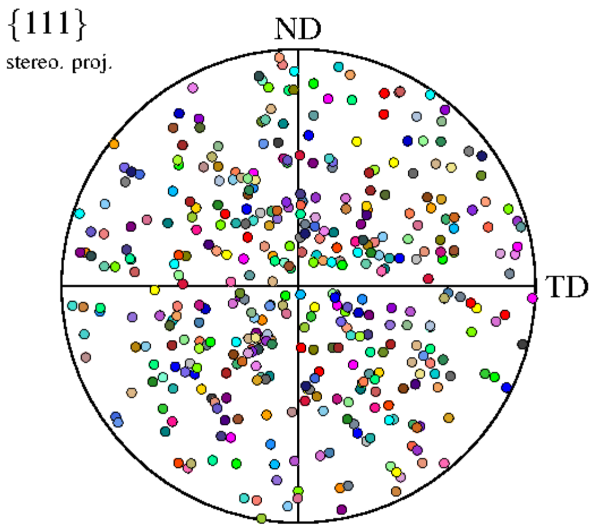


Figure 3: An example of a pole figure created with Neper. Each dot is a the orientation of a grain in a structure also generated in Neper.

The crystal coordinate system C_C given by the crystal axes $\{\mathrm{c}_1\mathrm{c}_2\mathrm{c}_3\}$ is projected into the specimen coordinate system C_S given by the specimen axes $\{\mathrm{s}_1\mathrm{s}_2\mathrm{s}_3\}$. The relation between the two coordinate systems can be expressed as

$$C_C = gC_S \tag{1}$$

A vector parallel to the pole of one of the crystal frames (XYZ) can be expressed in the C_S coordinate system as

$$R = s_1 \sin(\alpha)\cos(\beta) + s_2 \sin(\alpha)\sin(\beta) + s_3 \cos(\alpha) \tag{2}$$

and in the C_C coordinate system as

$$R = \frac{1}{N}(c_1 X + c_2 Y + c_3 Z) \tag{3}$$

where $N = \sqrt{X^2 + Y^2 + Z^2}$ to normalize the vector R. Scalar multiplication of Equation 2 and Equation 3 by the s vectors in relation to Equation 1 gives

$$\begin{bmatrix} \sin(\alpha)\cos(\beta) \\ \sin(\alpha)\sin(\beta) \\ \cos(\alpha) \end{bmatrix} = \frac{1}{N} \begin{bmatrix} g_{11} & g_{21} & g_{31} \\ g_{12} & g_{22} & g_{32} \\ g_{13} & g_{23} & g_{33} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{4}$$

Equation 4 gives nine equivalent expressions to find the angles α and β for a pole from the orientation matrix g.

As described earlier, one pole is not enough to unambiguously determine the orientation of a crystal. In Figure 2, the crystal can sill rotate around its own pole, so another poles is needed. To completely derive the orientation matrix g in Equation 4 two or three poles may be needed, depending on the crystal or pole symmetry [5].

2.1.2 The Orientation Distribution Function

When projecting the 3D orientation distribution on a 2D projection plane some information about the system will be lost. It is therefor often impossible to recreate the texture with only a pole figure, without some uncertainty. Poles of different orientations may overlap on the figure, linking certain volumes to their orientations hard. The intensities of the poles may also not clearly link them to the orientations they are describing. Lastly pole figures from experiments rarely record their entire range, and so limits the ability to fully describe the texture further.

These uncertainties can be reduced by using an orientation distribution function ODF, which allows quantitative evaluations of the texture. An ODF is a probability density functions for grain orientations. In a microstructure with grain orientations expressed through the euler angles ϕ_1, Φ, ϕ_2, has a sample volume V, and dV is the volume of all crystallites with the orientation g in the angular element dg. The ODF of this structure can be defined as $f(g)$ in the given equations:

$$\frac{dV}{V} = f(g)dg \tag{5}$$

$$dg = \frac{1}{8\pi^2}\sin(\Phi)d\phi_1 d\Phi d\phi_2 \tag{6}$$

$$\oint f(g)dg = 1 \tag{7}$$

The pole figure is defined mathematically as the volume fraction dV/V of crystals that have their direction h parallel to the sample direction y:

$$\frac{dV}{V} = \frac{1}{4\pi}P_h(y)dy \tag{8}$$

where $y = \{\alpha, \beta\}$ and $dy = \sin(\alpha)d\alpha d\beta$. α and β are the pole figure angles discussed in section 2.1.1. This definition bases the pole figure on directions $h = h_1h_2h_3$ in the direct crystal lattice, while experimental pole figures are often based on the crystal direction h in the reciprocal crystal lattice. This is not a big difference for cubic crystal structures, but for other structures it must be accounted for.

Pole figures are often normalized so that pole densities can be described as compared to median random density. Pole figure data is therefore normalized with this equation:

$$\frac{1}{4\pi}\oint P_h(y)dy = 1 \tag{9}$$

Figure 4 is a visual representation of how a pole represented by a direction y in a pole figure $P_h(y)$ fits with a region in the ODF $f(g)$ that contains all rotations with angle γ about this direction y. Said in another way, all rotations with angle γ in the ODF will show up with direction y on the pole figure. As an equation it is described as:

$$P_h(y) = \frac{1}{2\pi}\int_{\gamma=0}^{2\pi} f(g)d\gamma \tag{10}$$

where $y = \{\alpha, \beta\}$ and $g = \{\phi_1, \Phi, \phi_2\}$. The factor $1/2\pi$ comes from the normalization conditions from Equation 6 and Equation 9. Equation 10 is known as the fundamental equation of ODF computation, and is needed to calculate the ODF. The ODF can't be solved by a single pole figure, and several pole figures are needed to get the missing information. There are no analytical solutions to inversed pole figures, but some mathematical attempts have been proposed [5].

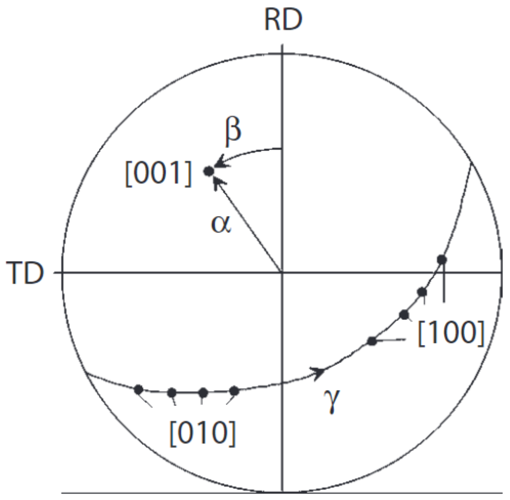


Figure 4: Representation of the fundamental equation of pole figure inversion in a 100 pole figure. Image borrowed from [5]

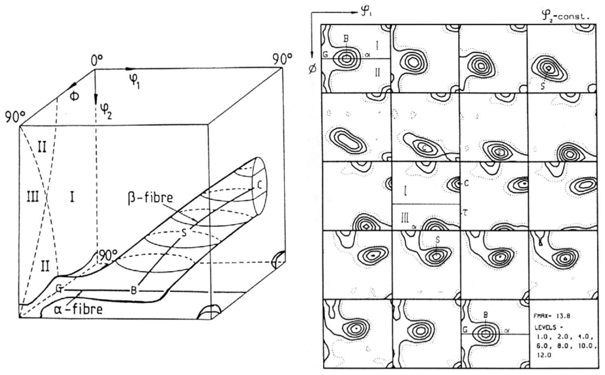


Figure 5: Schematic picture of a β-fiber - A typical texture of cold rolled aluminium. Image borrowed from

[6]

2.2 Crystal Plasticity Finite-Element Model

The Crystal Plasticity Finite-Element Models are approaches made to simulate both microstructures and macrostructures under stress. More specifically, they are used to make microstructure-based mechanical predictions, and engineering design and performance simulations for anisotropic materials. A more technical term is ”continuum-based variational formulations for describing the elastic-plastic deformations of anisotropic heterogeneous crystalline matter”. As can be discerned from this description, the models use continuous equations to simulate both elastic and plastic deformations on materials made up of crystalline structures. The applications for such models are many, and across diverse fields. A short list can count applications like microbeam bending, multiscale predicitions of rolling textures, damage prediction on micro- and macrostructures, recrystallization, and more.

Crystal properties like shape change, texture, strength, strain hardening, and deformation-induced surface roughening are not independent of orientation [13]. In other words crystals are mechanically anistropic, which means their mechanical parameters are tensor qualities.

The origin of the instant strain-rate sensitivity is that the critical shear stress depends on the shear rate of its slip system. The viscoplastic power law is given as [KILDE Overview .42. Texture Development and Strain-Hardening in Rate Dependent Polycrystals R. J. Asaro and A. Needleman]

$$\tau_{c\alpha} = \tau_\alpha^0 \left(\frac{|\dot{\gamma_\alpha}|}{\dot{\gamma_0}} \right)^{\frac{1}{m}} \tag{11}$$

Chapter 3

Methods

3.1 Neper

Neper [10] is a program that allows the generation of polycrystals, and create meshes. It may also be used to process the results from simulations, and make visualizations. The software package is free and open source, and was created by Romain Quey at CNRS and Mines Saint-Etienne. The meshes it creates can be used by finite-element simulations programs like FEpX or Abaqus. After the simulations, Neper can process the results and visualize them.

To use Neper it can be freely downloaded from GitHub, but it can only run on Linux-like systems. As it is written in C and C++, compilation can be done using CMake. The software has some dependencies that are needed depending on what you want to create. GMSH is for example required for meshing jobs, while Asymptote is used for creating pole diagrams.

When using Neper, simply enter command lines into a command window, or have the commands run through a shell script. The program will automatically use as many threads as are available, but efficiency drops after 16.

Neper is divided into four main modules: Generation, Meshing, Simulation, and Visualization.

The Generation-Module is used for creating tessellations from polycrystals it generates. There is a great variety for how the tessellations can be generated. These can be standard tessellations, tessellations generated from grain cell properties, or multiscale tessellations. An especially helpful feature is that the tessellations can be regularized, where smaller parts of it like edges and faces are removed and absorbed into the bigger overarching features. This can greatly simplify the meshes created from the tessellation, and make simulations on the mesh easier. The module also allows periodicity, and different types of orientation descriptors of the grain crystals.

The Meshing-Module creates meshes of the polycrystals described in the tessellation-file. It may also create meshes from other mesh-files, called re-meshing. The meshing process can create two different mesh types; free meshes that follow the boundaries of the crystals, and mapped meshes that generate square or cubic elements which are independent of the grain structure. Meshes created from this module can be of many different types, but the ones used in this thesis are msh-files for use by FEpX, and inp-files that are needed by Abaqus.

Simulation-Module is the smallest module and is tightly connected to the FEpX software. This module processes the results from FEpX and collects the raw data into a directory where it can be used by other processes. For example the Visualization module to create diagrams or pictures of the results, or it can be used to re-mesh the material after the simulation.

For visualizing tessellations, meshes or simulation results, there is the Visualization-Module. The module can take internal data, data from external files or a simulation directory, and create PNG-images or VTK-files. By changing the input parameters, highly different output can be created.

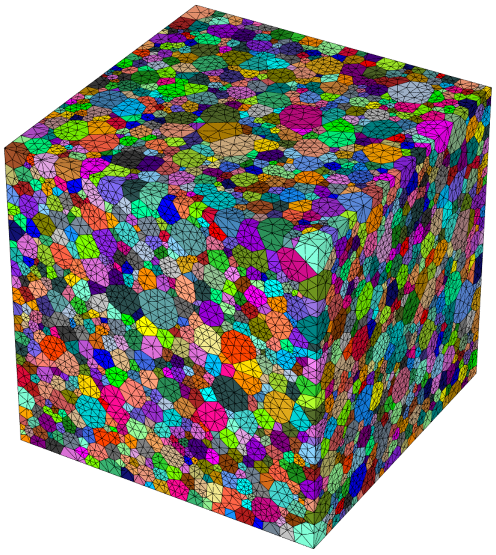


Figure 6: Example of a meshed tessellation cube created in Neper. It has 17445 grains, its morphology comes from grain-growth, and regularization is on. For the meshing, the characteristic length is 10, which means the mesh is as coarse as the program allows. This can be seen by how the mesh coarness varies by the grain size.

3.1.1 Generation-Module

The Generation-Module, accessed by writing -T, generates the tessellations of structures that can be used by other programs for simulations. The tessellations can be created in 2D or 3D, and in three different shapes; cubic, cylindrical or spherical, all with finite dimensions. Tessellations with non-convex domains may be created by cutting a tessellation with geometric structures like spheres, cylinders, squares, etc, and by adjusting the dimensions of both the tessellation and the cutting parts. The number of crystals can either be set from input, or be calculated internally based on grain growth.

When generating the tessellations, their morphological cell properties can be selected as an input parameter. These can be Voronoi which is the default setting, grain growth like those found in metals, or using custom parameters to design their size, sphericity, centroid, shape distribution and individual cell shapes. Other global properties like cell aspect ratio and columnar axis, may be set as well. Cell groups can also be established, where the different phases of a multiphased material is linked to each group.

The cells of a tessellation can be divided into subdivisions. These multiscale tessellations can be used to describe complicated microstructures, like lamellars or internal crystal structures. The internal tesselaltions can again have their own internal tessellations and so on. Meshing of these multiscale tessellations works the same way as a standard tessellation.

To avoid boundary effect, or reduce the size of the representative volume element, periodicity along the faces of the tessellation can be induced. This can be a full periodicity, where the top face matches the bottom face, the left face matches the right face, and the innermost face matches the outermost face. Semi-periodicity can also be done, where only one pair, or two pairs of faces match.

The orientation of the different crystals can be set to a certain degree. They can either be set to random, which is the default setting where the orientation of the cells is distributed randomly according to a uniform distribution. The other setting is uniform, where the orientations are distributed uniformly and there will be no local orientation clusters. If setting the orientations uniformly, the crystal symmetry must also be stated. The orientation of the cells in the tess-file is described with rodrigues-vectors by default, but other orientation descriptions are also supported. This includes euler-bunge, euler-kocks, euler-roe, rotmat, axis-angle, and quaternions.

Importantly the tessellations can be regularized for optimizing meshes, and in turn simulations on them. Small crystals or sides with tiny edges or faces have little impact on the cell morphologies, but can complicate the mesh by representing strong constraints on it. More nodes and elements are needed in the mesh to describe small edges and faces in the tessellation. This in turn means the simulations on the mesh will work harder, and increases the chances it will fail when one of those elements are strained too much. By regularizing the tessellation, small edges and faces are removed, which simplifies the mesh, and any simulations running on the mesh will be easier as well.

3.1.2 Meshing-Module

By writing -M the meshing module can be accessed. This creates a mesh based on either an already existing, mesh, a tesselation, or another collection of data which simulates a material. These meshes are very useful for simulations of materials, as the behaviour of the material can be translated into data around the nodes and elements in the meshes. The finer the mesh, the more nodes and elements it contains, and the more data can be created and processed by the simulation.

3.1.3 Simulation-Module

After a simulation has run successfully by another program, the Simulation module can be used to convert the data. This data is then moved into a separate folder and converted into a file-type that Neper can use to visualize the data. This is data about the nodes and elements in the mesh, where each file can describe, for example, the position of the node after each step of the simulation, the stress at the nodes location, the plastic equivalent elasticity, etc.

3.1.4 Visualization-Module

Both before and after a simulation, it can be useful to visualize the data to make sure it looks like it is supposed to. The Visualization module can create a PNG from tesselation-files, mesh-files, tesselation- and mesh-files combined, the folder of simulation results created from the Simulation module, and many other inputs. It has a wide variety of settings to control the output of the picture, like camera distance and angle, rotation of the material, size of the picture, etc. When visualizing data from a finished simulation, it can show the data as a color scheme on the material, with a scale picture, and allows customization of the scale parameters. The module can also create other visualizations, like Pole Figures, and 2D-images.

3.2 FEpX

FEpX [3] is a software package created to solve finite-element polycrystal plasticity methods on Linux-like operating systems. Its development started in the late 1990's, by Paul

Dawson and the Deformation Process Laboratory at Cornell University. Later development was lead by Matthew Kasemer and the Advanced Computational Materials Engineering Laboratory at the University of Alabama. It models both global and local mechanical behaviour in finite polycrystalline solid structures as aggregates of grains. Every grain of the structure is made up of elements, which behavior is determined by the local behavior of that volume of the crystal. The element behaviors it models are:

- Movement and rotations resolving finite strains.
- Anisotropic elasticity from crystal symmetry.
- Anisotropic plasticity from slip confined to dominant slip systems.
- Change in state variables for crystal lattice orientations and slip system strengths.

3.2.1 Code Structure

The FEpX algorithms are written in Fortran 90. It's a programming language used for over sixty years in computationally intensive areas like numerical weather prediction, finite element analysis, computational fluid dynamics, geophysics, computationally physics, crystallography, and computationally chemistry [7]. In Fortran, the code is distributed in different files called modules, where each module has its own responsibilities. Each module can itself contain subroutines that can be called upon from either inside the same module, or other modules depending on its accessibility.

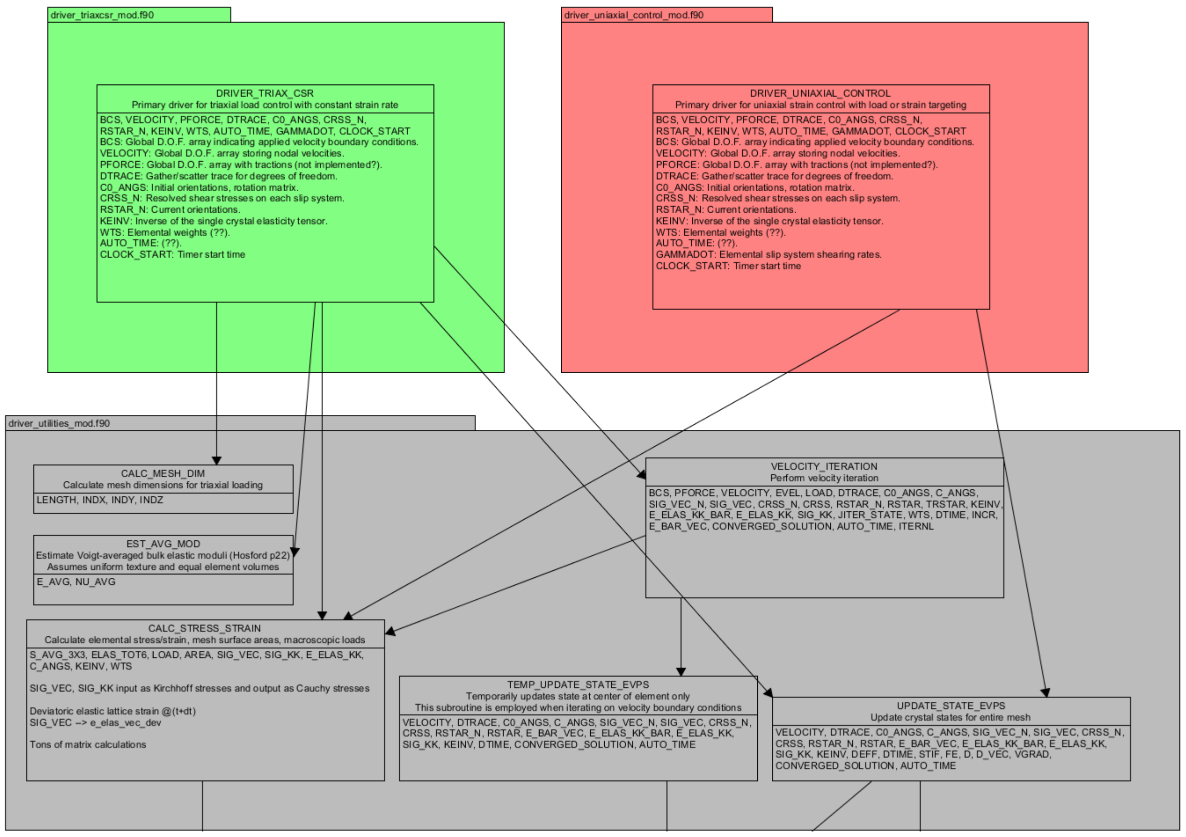

driver_triaxcsr_mod.f90
DRIVER_TRIAX_CSR
Primary driver for triaxial load control with constant strain rate
BCS, VELOCITY, PFORCE, DTRACE, C0_ANGS, CRSS_N, RSTAR_N, KEINV, WTS, AUTO_TIME, GAMMADOT, CLOCK_START
BCS: Global D.O.F. array indicating applied velocity boundary conditions.
VELOCITY: Global D.O.F. array storing nodal velocities.
PFORCE: Global D.O.F. array with tractions (not implemented?).
DTRACE: Gather/scatter trace for degrees of freedom.
C0_ANGS: Initial orientations, rotation matrix.
CRSS_N: Resolved shear stresses on each slip system.
RSTAR_N: Current orientations.
KEINV: Inverse of the single crystal elasticity tensor.
WTS: Elemental weights (??).
AUTO_TIME: (??).
CLOCK_START: Timer start time
driver_uniaxial_control_mod.f90
DRIVER_UNIAXIAL_CONTROL
Primary driver for uniaxial strain control with load or strain targeting
BCS, VELOCITY, PFORCE, DTRACE, C0_ANGS, CRSS_N, RSTAR_N, KEINV, WTS, AUTO_TIME, GAMMADOT, CLOCK_START
BCS: Global D.O.F. array indicating applied velocity boundary conditions.
VELOCITY: Global D.O.F. array storing nodal velocities.
PFORCE: Global D.O.F. array with tractions (not implemented?).
DTRACE: Gather/scatter trace for degrees of freedom.
C0_ANGS: Initial orientations, rotation matrix.
CRSS_N: Resolved shear stresses on each slip system.
RSTAR_N: Current orientations.
KEINV: Inverse of the single crystal elasticity tensor.
WTS: Elemental weights (??).
AUTO_TIME: (??).
GAMMADOT: Elemental slip system shearing rates.
CLOCK_START: Timer start time
driver_utilities_mod.f90
CALC_MESH_DIM
Calculate mesh dimensions for triaxial loading
LENGTH, INDX, INDY, INDZ
EST_AVG_MOD
Estimate Voigt-averaged bulk elastic moduli (Hosford p22)
Assumes uniform texture and equal element volumes
E_AVG, NU_AVG
CALC_STRESS_STRAIN
Calculate elemental stress/strain, mesh surface areas, macroscopic loads
S_AVG_3X3, ELAS_TOT6, LOAD, AREA, SIG_VEC, SIG_KK, E_ELAS_KK, C_ANGS, KEINV, WTS
SIG_VEC, SIG_KK input as Kirchhoff stresses and output as Cauchy stresses
Deviatoric elastic lattice strain @(t+dt)
SIG_VEC --> e_elas_vec_dev
Tons of matrix calculations
VELOCITY_ITERATION
Perform velocity iteration
BCS, PFORCE, VELOCITY, EVEL, LOAD, DTRACE, C0_ANGS, C_ANGS, SIG_VEC_N, SIG_VEC, CRSS_N, CRSS, RSTAR_N, RSTAR, TRSTAR, KEINV, E_ELAS_KK_BAR, E_ELAS_KK, SIG_KK, JITER_STATE, WTS, DTIME, INCR, E_BAR_VEC, CONVERGED_SOLUTION, AUTO_TIME, ITERNL
TEMP_UPDATE_STATE_EVPS
Temporarily updates state at center of element only
This subroutine is employed when iterating on velocity boundary conditions
VELOCITY, DTRACE, C0_ANGS, C_ANGS, SIG_VEC_N, SIG_VEC, CRSS_N, CRSS, RSTAR_N, RSTAR, E_BAR_VEC, E_ELAS_KK_BAR, E_ELAS_KK, SIG_KK, KEINV, DTIME, CONVERGED_SOLUTION, AUTO_TIME
UPDATE_STATE_EVPS
Update crystal states for entire mesh
VELOCITY, DTRACE, C0_ANGS, C_ANGS, SIG_VEC_N, SIG_VEC, CRSS_N, CRSS, RSTAR_N, RSTAR, E_BAR_VEC, E_ELAS_KK_BAR, E_ELAS_KK, SIG_KK, KEINV, DEFF, DTIME, STIF, FE, D, D_VEC, VGRAD, CONVERGED_SOLUTION, AUTO_TIME

Figure 7: An overview over the driver modules. The triaxial driver and the uniaxial driver gets called depending on what kind of simulation is running, and they in turn make calls to the necessary subroutines in the utilities driver.

In FEpX, all simulations are divided into either triaxial or uniaxial simulations. As such, all calls to lower end subroutines can be traced back to two drivers. The triaxial driver *driver_triaxcsr_mod.f90* and the uniaxial driver *driver_uniaxial_control_mod.f90*, as can be seen in Figure 7. They in turn make calls to the utilities driver *driver_utilities_mod.f90* to access necessary subroutines.

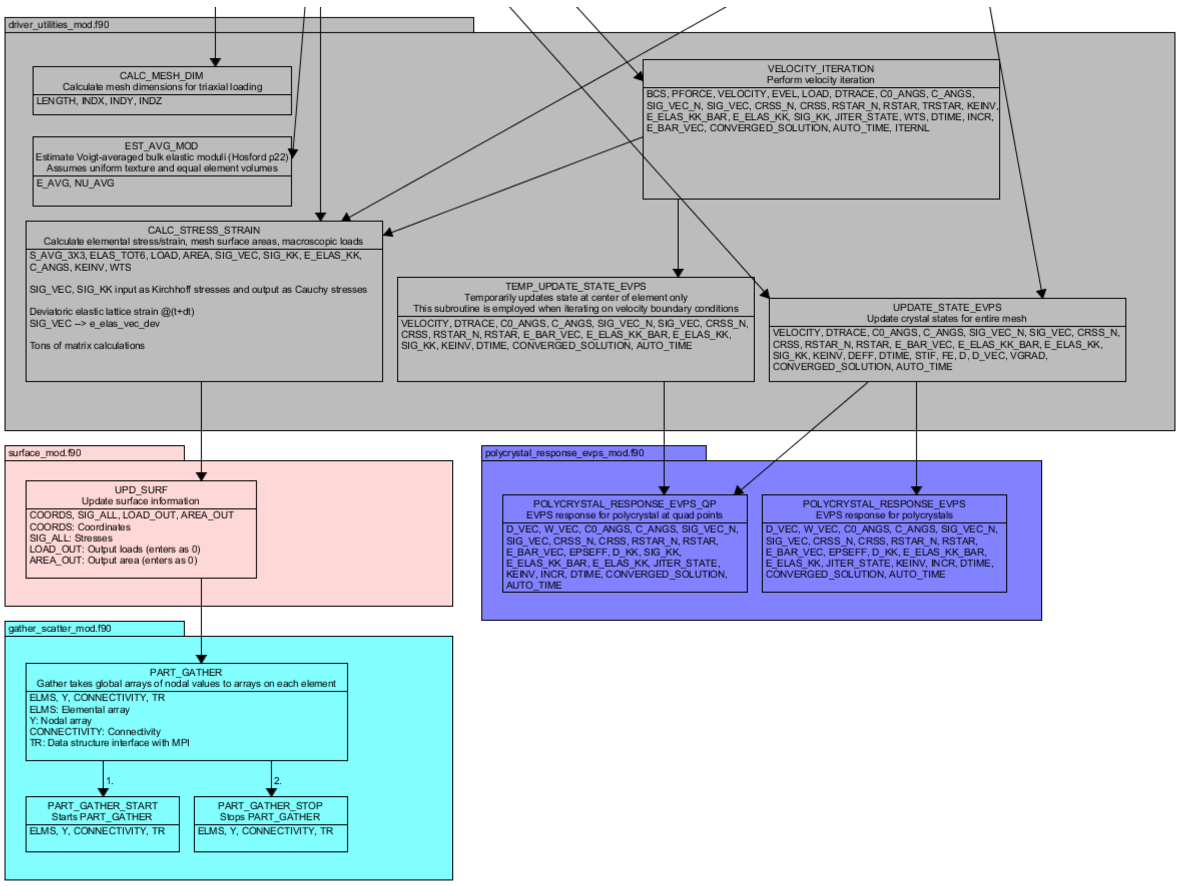

driver_utilities_mod.f90
CALC_MESH_DIM
Calculate mesh dimensions for triaxial loading
LENGTH, INDX, INDY, INDZ
EST_AVG_MOD
Estimate Voigt-averaged bulk elastic moduli (Hosford p22)
Assumes uniform texture and equal element volumes
E_AVG, NU_AVG
CALC_STRESS_STRAIN
Calculate elemental stress/strain, mesh surface areas, macroscopic loads
S_AVG_3X3, ELAS_TOT6, LOAD, AREA, SIG_VEC, SIG_KK, E_ELAS_KK, C_ANGS, KEINV, WTS
SIG_VEC, SIG_KK input as Kirchhoff stresses and output as Cauchy stresses
Deviatoric elastic lattice strain @(t+dt)
SIG_VEC --> e_elas_vec_dev
Tons of matrix calculations
VELOCITY_ITERATION
Perform velocity iteration
BCS, PFORCE, VELOCITY, EVEL, LOAD, DTRACE, C0_ANGS, C_ANGS, SIG_VEC_N, SIG_VEC, CRSS_N, CRSS, RSTAR_N, RSTAR, TRSTAR, KEINV, E_ELAS_KK_BAR, E_ELAS_KK, SIG_KK, JITER_STATE, WTS, DTIME, INCR, E_BAR_VEC, CONVERGED_SOLUTION, AUTO_TIME, ITERNL
TEMP_UPDATE_STATE_EVPS
Temporarily updates state at center of element only
This subroutine is employed when iterating on velocity boundary conditions
VELOCITY, DTRACE, C0_ANGS, C_ANGS, SIG_VEC_N, SIG_VEC, CRSS_N, CRSS, RSTAR_N, RSTAR, E_BAR_VEC, E_ELAS_KK_BAR, E_ELAS_KK, SIG_KK, KEINV, DTIME, CONVERGED_SOLUTION, AUTO_TIME
UPDATE_STATE_EVPS
Update crystal states for entire mesh
VELOCITY, DTRACE, C0_ANGS, C_ANGS, SIG_VEC_N, SIG_VEC, CRSS_N, CRSS, RSTAR_N, RSTAR, E_BAR_VEC, E_ELAS_KK_BAR, E_ELAS_KK, SIG_KK, KEINV, DEFF, DTIME, STIF, FE, D, D_VEC, VGRAD, CONVERGED_SOLUTION, AUTO_TIME
surface_mod.f90
UPD_SURF
Update surface information
COORDS, SIG_ALL, LOAD_OUT, AREA_OUT
COORDS: Coordinates
SIG_ALL: Stresses
LOAD_OUT: Output loads (enters as 0)
AREA_OUT: Output area (enters as 0)
polycrystal_response_evps_mod.f90
POLYCRYSTAL_RESPONSE_EVPS_QP
EVPS response for polycrystal at quad points
D_VEC, W_VEC, C0_ANGS, C_ANGS, SIG_VEC_N, SIG_VEC, CRSS_N, CRSS, RSTAR_N, RSTAR, E_BAR_VEC, EPSEFF, D_KK, SIG_KK, E_ELAS_KK_BAR, E_ELAS_KK, JITER_STATE, KEINV, INCR, DTIME, CONVERGED_SOLUTION, AUTO_TIME
POLYCRYSTAL_RESPONSE_EVPS
EVPS response for polycrystals
D_VEC, W_VEC, C0_ANGS, C_ANGS, SIG_VEC_N, SIG_VEC, CRSS_N, CRSS, RSTAR_N, RSTAR, E_BAR_VEC, EPSEFF, D_KK, E_ELAS_KK_BAR, E_ELAS_KK, JITER_STATE, KEINV, INCR, DTIME, CONVERGED_SOLUTION, AUTO_TIME
gather_scatter_mod.f90
PART_GATHER
Gather takes global arrays of nodal values to arrays on each element
ELMS, Y, CONNECTIVITY, TR
ELMS: Elemental array
Y: Nodal array
CONNECTIVITY: Connectivity
TR: Data structure interface with MPI
1.
2.
PART_GATHER_START
Starts PART_GATHER
ELMS, Y, CONNECTIVITY, TR
PART_GATHER_STOP
Stops PART_GATHER
ELMS, Y, CONNECTIVITY, TR

Figure 8: Here it can be seen how the utilities driver starts the calculations for different variables and states, but the module needs to call upon subroutines in other modules for more specific calculations.

The utilities driver contains the subroutines needed to calculate different variables as the simulation iterates through the time steps. This is where calculations for different variables are started. Variables like mesh dimensions, mesh elemental properties like stress/strain, mesh surface areas, macroscopic load, velocities, and crystal states start their calculation process here. The calculating subroutines often need to call upon other modules for the more specific calculations however, as putting all the necessary algorithms in just the driver module would make unnecessarily messy. As can be seen in Figure 8, the utilities driver is itself reliant on more specialized modules like the surface module *surface_mod.f90*, and the polycrystal elastic-viscoplastic response module *polycrystal_response_evps_mod.f90*.

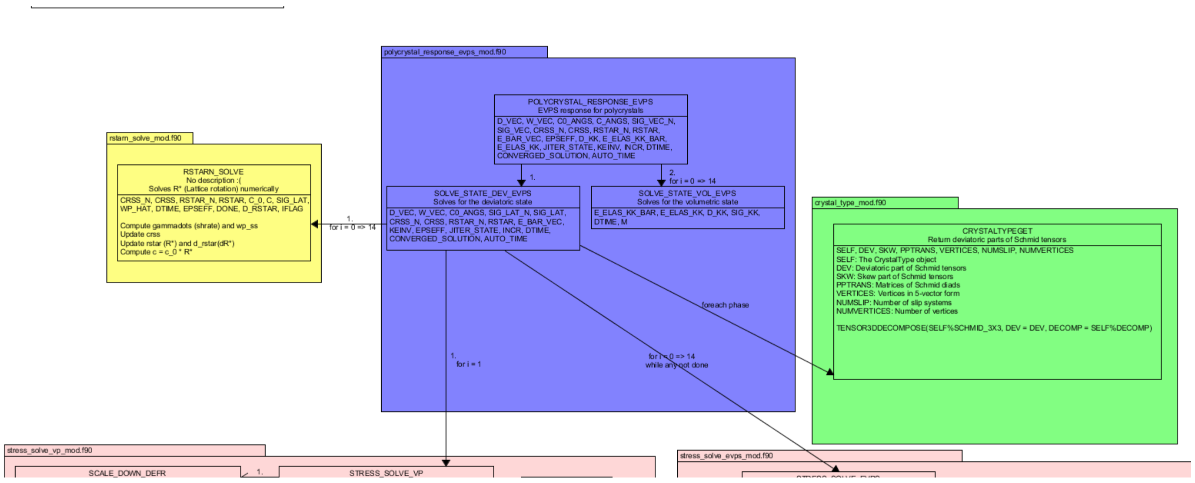

polycrystal_response_evps_mod.f90
POLYCRYSTAL_RESPONSE_EVPS
EVPS response for polycrystals
D_VEC, W_VEC, C0_ANGS, C_ANGS, SIG_VEC_N, SIG_VEC, CRSS_N, CRSS, RSTAR_N, RSTAR, E_BAR_VEC, EPSEFF, D_KK, E_ELAS_KK_BAR, E_ELAS_KK, JITER_STATE, KEINV, INCR, DTIME, CONVERGED_SOLUTION, AUTO_TIME
1.
2. for i = 0 => 14
SOLVE_STATE_DEV_EVPS
Solves for the deviatoric state
D_VEC, W_VEC, C0_ANGS, SIG_LAT_N, SIG_LAT, CRSS_N, CRSS, RSTAR_N, RSTAR, E_BAR_VEC, KEINV, EPSEFF, JITER_STATE, INCR, DTIME, CONVERGED_SOLUTION, AUTO_TIME
SOLVE_STATE_VOL_EVPS
Solves for the volumetric state
E_ELAS_KK_BAR, E_ELAS_KK, D_KK, SIG_KK, DTIME, M
rstarn_solve_mod.f90
RSTARN_SOLVE
No description :(
Solves R* (Lattice rotation) numerically
CRSS_N, CRSS, RSTAR_N, RSTAR, C_0, C, SIG_LAT, WP_HAT, DTIME, EPSEFF, DONE, D_RSTAR, IFLAG
Compute gammadots (shrate) and wp_ss
Update crss
Update rstar (R*) and d_rstar(dR*)
Compute c = c_0 * R*
1. for i = 0 => 14
foreach phase
1. for i = 1
for i = 0 => 14 while any not done
crystal_type_mod.f90
CRYSTALTYPEGET
Return deviatoric parts of Schmid tensors
SELF, DEV, SKW, PPTRANS, VERTICES, NUMSLIP, NUMVERTICES
SELF: The CrystalType object
DEV: Deviatoric part of Schmid tensors
SKW: Skew part of Schmid tensors
PPTRANS: Matrices of Schmid diads
VERTICES: Vertices in 5-vector form
NUMSLIP: Number of slip systems
NUMVERTICES: Number of vertices
TENSOR3DDECOMPOSE(SELF%SCHMID_3X3, DEV = DEV, DECOMP = SELF%DECOMP)
stress_solve_vp_mod.f90
SCALE_DOWN_DEFR
1.
STRESS_SOLVE_VP
stress_solve_evps_mod.f90

Figure 9: The calculations for elastic-viscoplastic response for polycrystals is reliant on stress solve modules. When handling the deviatoric state it iterates over 15 integration points per element. The viscoplastic stress is solved for the first integration point, while the elastic-viscoplastic stress is solved for all points afterwards.

The module handling elastic-viscoplastic response for polycrystals is itself divided into two states it needs to solve for each element. The deviatoric state and the volumetric state. For both states it iterates over 15 integration points. For the deviatoric state the stress must be solved by using the viscoplastic stress solving module *stress_solve_vp_mod.f90* for the first integration point, and then by using the elastic-viscoplastic stress solving module *stress_solve_evps_mod.f90* for all superseding points. Figure 9 shows a simple overview of the subroutines of the elastic-viscoplastic response for polycrystals module, and their dependencies.

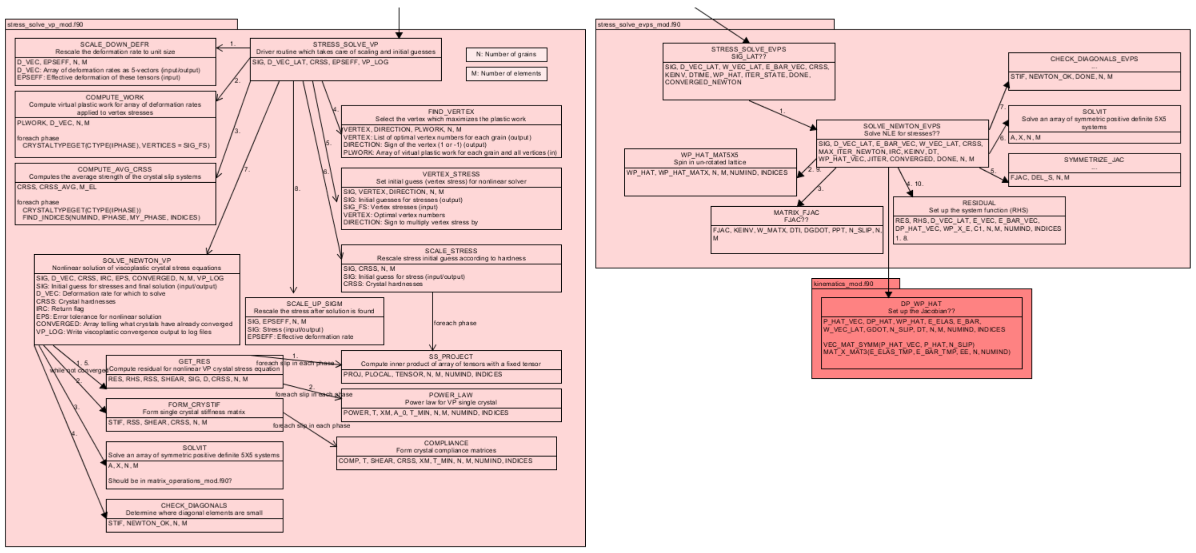

stress_solve_vp_mod.f90
STRESS_SOLVE_VP
Driver routine which takes care of scaling and initial guesses
SIG, D_VEC_LAT, CRSS, EPSEFF, VP_LOG
N: Number of grains
M: Number of elements
SCALE_DOWN_DEFR
Rescale the deformation rate to unit size
D_VEC, EPSEFF, N, M
D_VEC: Array of deformation rates as 5-vectors (input/output)
EPSEFF: Effective deformation of these tensors (input)
COMPUTE_WORK
Compute virtual plastic work for array of deformation rates applied to vertex stresses
PLWORK, D_VEC, N, M
foreach phase
CRYSTALTYPEGET(CTYPE(IPHASE), VERTICES = SIG_FS)
COMPUTE_AVG_CRSS
Computes the average strength of the crystal slip systems
CRSS, CRSS_AVG, M_EL
foreach phase
CRYSTALTYPEGET(CTYPE(IPHASE))
FIND_INDICES(NUMIND, IPHASE, MY_PHASE, INDICES)
FIND_VERTEX
Select the vertex which maximizes the plastic work
VERTEX, DIRECTION, PLWORK, N, M
VERTEX: List of optimal vertex numbers for each grain (output)
DIRECTION: Sign of the vertex (1 or -1) (output)
PLWORK: Array of virtual plastic work for each grain and all vertices (in)
VERTEX_STRESS
Set initial guess (vertex stress) for nonlinear solver
SIG, VERTEX, DIRECTION, N, M
SIG: Initial guesses for stresses (output)
SIG_FS: Vertex stresses (input)
VERTEX: Optimal vertex numbers
DIRECTION: Sign to multiply vertex stress by
SCALE_STRESS
Rescale stress initial guess according to hardness
SIG, CRSS, N, M
SIG: Initial guess for stress (input/output)
CRSS: Crystal hardnesses
SOLVE_NEWTON_VP
Nonlinear solution of viscoplastic crystal stress equations
SIG, D_VEC, CRSS, IRC, EPS, CONVERGED, N, M, VP_LOG
SIG: Initial guess for stresses and final solution (input/output)
D_VEC: Deformation rate for which to solve
CRSS: Crystal hardnesses
IRC: Return flag
EPS: Error tolerance for nonlinear solution
CONVERGED: Array telling what crystals have already converged
VP_LOG: Write viscoplastic convergence output to log files
SCALE_UP_SIGM
Rescale the stress after solution is found
SIG, EPSEFF, N, M
SIG: Stress (input/output)
EPSEFF: Effective deformation rate
foreach phase
GET_RES
Compute residual for nonlinear VP crystal stress equation
RES, RHS, RSS, SHEAR, SIG, D, CRSS, N, M
while not converged
foreach slip in each phase
SS_PROJECT
Compute inner product of array of tensors with a fixed tensor
PROJ, PLOCAL, TENSOR, N, M, NUMIND, INDICES
POWER_LAW
Power law for VP single crystal
POWER, T, XM, A_0, T_MIN, N, M, NUMIND, INDICES
FORM_CRYSTIF
Form single crystal stiffness matrix
STIF, RSS, SHEAR, CRSS, N, M
COMPLIANCE
Form crystal compliance matrices
COMP, T, SHEAR, CRSS, XM, T_MIN, N, M, NUMIND, INDICES
SOLVIT
Solve an array of symmetric positive definite 5X5 systems
A, X, N, M
Should be in matrix_operations_mod.f90?
CHECK_DIAGONALS
Determine where diagonal elements are small
STIF, NEWTON_OK, N, M
stress_solve_evps_mod.f90
STRESS_SOLVE_EVPS
SIG_LAT??
SIG, D_VEC_LAT, W_VEC_LAT, E_BAR_VEC, CRSS, KEINV, DTIME, WP_HAT, ITER_STATE, DONE, CONVERGED_NEWTON
SOLVE_NEWTON_EVPS
Solve NLE for stresses??
SIG, D_VEC_LAT, E_BAR_VEC, W_VEC_LAT, CRSS, MAX_ITER_NEWTON, IRC, KEINV, DT, WP_HAT_VEC, JITER, CONVERGED, DONE, N, M
CHECK_DIAGONALS_EVPS
STIF, NEWTON_OK, DONE, N, M
SOLVIT
Solve an array of symmetric positive definite 5X5 systems
A, X, N, M
SYMMETRIZE_JAC
FJAC, DEL_S, N, M
WP_HAT_MAT5X5
Spin in un-rotated lattice
WP_HAT, WP_HAT_MATX, N, M, NUMIND, INDICES
MATRIX_FJAC
FJAC??
FJAC, KEINV, W_MATX, DTI, DGDOT, PPT, N_SLIP, N, M
RESIDUAL
Set up the system function (RHS)
RES, RHS, D_VEC_LAT, E_VEC, E_BAR_VEC, DP_HAT_VEC, WP_X_E, C1, N, M, NUMIND, INDICES
kinematics_mod.f90
DP_WP_HAT
Set up the Jacobian??
P_HAT_VEC, DP_HAT, WP_HAT, E_ELAS, E_BAR, W_VEC_LAT, GDOT, N_SLIP, DT, N, M, NUMIND, INDICES
VEC_MAT_SYMM(P_HAT_VEC, P_HAT, N_SLIP)
MAT_X_MAT3(E_ELAS_TMP, E_BAR_TMP, EE, N, NUMIND)

Figure 10: Here it can be seen how the stress solving modules solve for each integration point by calling upon their own specialized subroutines. Only one subroutine in the modules are public and can be called by other modules, while all the other subroutines exist as internal helping subroutines.

The stress solving modules do not make use of any other modules, excluding the generic matrix operations module *matrix_operations_mod.f90*, and one call to the kinematics module *kinematics_mod.f90.* They are relatively self contained modules that solve the viscoplastic and elastic-viscoplastic stress for integration points in the elements of the mesh. In the viscoplastic stress solve module the input is first restructured to fit the internal subroutines. It then solves for the stress by using nonlinear viscoplastic crystal stress equations repeatedly until it converges on a solution. Lastly the stress solution is rescaled to better fit the expected output. The elastic-viscoplastic stress solving module is much simpler, but is dependent on the solution found in the other module to work properly. In Figure 10 an overview of the interconnected subroutines in the two modules can be observed.

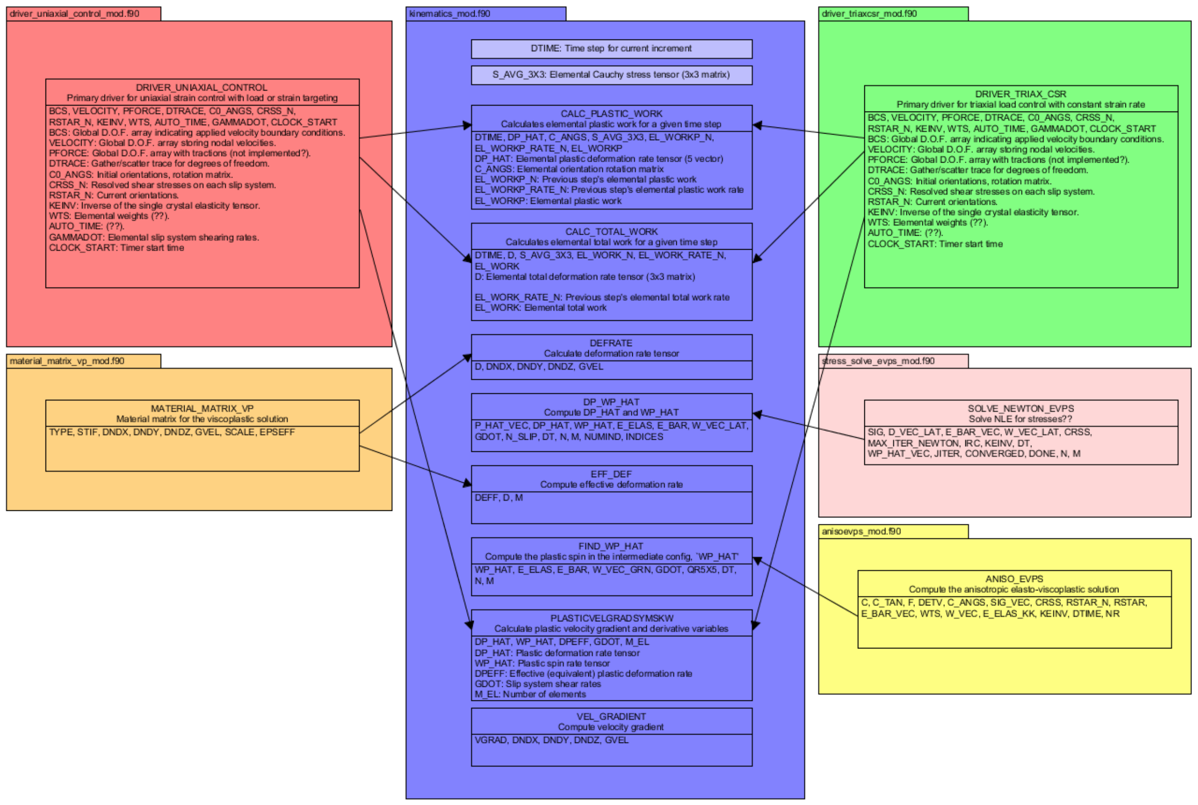

driver_uniaxial_control_mod.f90
DRIVER_UNIAXIAL_CONTROL
Primary driver for uniaxial strain control with load or strain targeting
BCS, VELOCITY, PFORCE, DTRACE, C0_ANGS, CRSS_N,
RSTAR_N, KEINV, WTS, AUTO_TIME, GAMMADOT, CLOCK_START
BCS: Global D.O.F. array indicating applied velocity boundary conditions.
VELOCITY: Global D.O.F. array storing nodal velocities.
PFORCE: Global D.O.F. array with tractions (not implemented?).
DTRACE: Gather/scatter trace for degrees of freedom.
C0_ANGS: Initial orientations, rotation matrix.
CRSS_N: Resolved shear stresses on each slip system.
RSTAR_N: Current orientations.
KEINV: Inverse of the single crystal elasticity tensor.
WTS: Elemental weights (??).
AUTO_TIME: (??).
GAMMADOT: Elemental slip system shearing rates.
CLOCK_START: Timer start time
material_matrix_vp_mod.f90
MATERIAL_MATRIX_VP
Material matrix for the viscoplastic solution
TYPE, STIF, DNDX, DNDY, DNDZ, GVEL, SCALE, EPSEFF
kinematics_mod.f90
DTIME: Time step for current increment
S_AVG_3X3: Elemental Cauchy stress tensor (3x3 matrix)
CALC_PLASTIC_WORK
Calculates elemental plastic work for a given time step
DTIME, DP_HAT, C_ANGS, S_AVG_3X3, EL_WORKP_N,
EL_WORKP_RATE_N, EL_WORKP
DP_HAT: Elemental plastic deformation rate tensor (5 vector)
C_ANGS: Elemental orientation rotation matrix
EL_WORKP_N: Previous step's elemental plastic work
EL_WORKP_RATE_N: Previous step's elemental plastic work rate
EL_WORKP: Elemental plastic work
CALC_TOTAL_WORK
Calculates elemental total work for a given time step
DTIME, D, S_AVG_3X3, EL_WORK_N, EL_WORK_RATE_N,
EL_WORK
D: Elemental total deformation rate tensor (3x3 matrix)
EL_WORK_RATE_N: Previous step's elemental total work rate
EL_WORK: Elemental total work
DEFRATE
Calculate deformation rate tensor
D, DNDX, DNDY, DNDZ, GVEL
DP_WP_HAT
Compute DP_HAT and WP_HAT
P_HAT_VEC, DP_HAT, WP_HAT, E_ELAS, E_BAR, W_VEC_LAT,
GDOT, N_SLIP, DT, N, M, NUMIND, INDICES
EFF_DEF
Compute effective deformation rate
DEFF, D, M
FIND_WP_HAT
Compute the plastic spin in the intermediate config, `WP_HAT'
WP_HAT, E_ELAS, E_BAR, W_VEC_GRN, GDOT, QR5X5, DT,
N, M
PLASTICVELGRADSYMSKW
Calculate plastic velocity gradient and derivative variables
DP_HAT, WP_HAT, DPEFF, GDOT, M_EL
DP_HAT: Plastic deformation rate tensor
WP_HAT: Plastic spin rate tensor
DPEFF: Effective (equivalent) plastic deformation rate
GDOT: Slip system shear rates
M_EL: Number of elements
VEL_GRADIENT
Compute velocity gradient
VGRAD, DNDX, DNDY, DNDZ, GVEL
driver_triaxcsr_mod.f90
DRIVER_TRIAX_CSR
Primary driver for triaxial load control with constant strain rate
BCS, VELOCITY, PFORCE, DTRACE, C0_ANGS, CRSS_N,
RSTAR_N, KEINV, WTS, AUTO_TIME, GAMMADOT, CLOCK_START
BCS: Global D.O.F. array indicating applied velocity boundary conditions.
VELOCITY: Global D.O.F. array storing nodal velocities.
PFORCE: Global D.O.F. array with tractions (not implemented?).
DTRACE: Gather/scatter trace for degrees of freedom.
C0_ANGS: Initial orientations, rotation matrix.
CRSS_N: Resolved shear stresses on each slip system.
RSTAR_N: Current orientations.
KEINV: Inverse of the single crystal elasticity tensor.
WTS: Elemental weights (??).
AUTO_TIME: (??).
CLOCK_START: Timer start time
stress_solve_evps_mod.f90
SOLVE_NEWTON_EVPS
Solve NLE for stresses??
SIG, D_VEC_LAT, E_BAR_VEC, W_VEC_LAT, CRSS,
MAX_ITER_NEWTON, IRC, KEINV, DT,
WP_HAT_VEC, JITER, CONVERGED, DONE, N, M
anisoevps_mod.f90
ANISO_EVPS
Compute the anisotropic elasto-viscoplastic solution
C, C_TAN, F, DETV, C_ANGS, SIG_VEC, CRSS, RSTAR_N, RSTAR,
E_BAR_VEC, WTS, W_VEC, E_ELAS_KK, KEINV, DTIME, NR

Figure 11: Overview of the kinematics module, its subroutines, and the other module subroutines calling them.

Lastly, Figure 11 shows the kinematics module *kinematics_mod.f90* and its different subroutines. A wide variety of modules makes use of its subroutines; both the driver modules and a more generic stress solving module require kinematics subroutines for calculations.

3.2.2 Running FEpX

To run a simulation with FEpX, simply write the following command in the command window:

```
    fepx
```

If more than one core is wanted, with *OpenMPI* installed the following command can be used instead to run FEpX with several cores:

```
    mpirun -np <NUMBER_OF_CORES> fepx
```

3.2.3 Input Files

The behavior of the simulation, and the model to be simulated should be stated in input files existing in the same directory as where the command is run. At least two files are required to run the simulation, but other optional files may be included. The first necessary file is a mesh file named *simulation.msh* created by Neper, that describes the polycrystalline structure that is to be simulated. At the moment only meshes made by 2nd order 10-node tetrahedral elements are supported [11]. The other file needed is a configuration file named *simulation.config* that describes the material being simulated, and how it should be simulated. The file is divided into four areas:

- **Optional Input** May relate to specific deformation modes or control standard simulation behavior.
- **Material Parameters** List of parameters describing the material. Typically includes stiffness tensor, rate sensitivity exponent, fixed-rate strain rate scaling coefficient, fixed-state hardening rate scaling coefficient, initial slip system saturation strength, initial slip system strength, and non-linear Voce hardening exponent.
- **Deformation History** Describes the deformation mode capable of reproducing various mechanical loading configurations. Can contain any number of steps, where each step defines the strain or load target that is to be reached.
- **Boundary Condition** Either defines a simple standard boundary condition available, or points towards another externally defined boundary conditions file *simulation.bcs*. It aslo states the strain rate value in units of [1/s].
- **Printing Results** List of all output data from the simulation that should be saved as nodal output in output-files.

Other optional files that can be included are:

- Orientation assignment file *simulation.ori* that describes the orientation of the nodes, overriding the values usually found in the mesh-file
- Crystallographic phase assignment *simulation.phase* which contains phase assignments overriding those in the mesh-file
- Defined boundary conditions *simulation.bcs* containing a list of node id's, the coordination index (X, Y, or Z), and the node's condition velocity in that direction. Such a file is used in the rolling simulation performed in this thesis. A simple C# script for creating such a file with rolling boundary conditions can be found in Appendix A.1.

3.2.4 Output Files

When a simulation is finished, FEpX creates a small *post.report* file with the necessary information for post-processing the raw data. The raw data is divided into one file per data type per core. Only data specifically asked for in the *Printing Results* section of the simulation.config file is printed. The output data can be divided into four categories:

- Nodal output that calculates and writes variables for each node in the mesh
- Elemental output that calculates and writes variables for each element in the mesh
- Restart output containing all variables needed to restart the simulation after a specific load step
- Miscellaneous output describing macroscopic forces and simulation statistics

For post-processing the raw data, the Simulation module of Neper is used. The processed data becomes more readable, and also available for the Visualization module of Neper. This module can then be used to create images and figures based on that data.

Chapter 4

Setup

4.1 Simulation Goals

4.2 Linux Machine Setup

Most of the programs that were used in this thesis require a Linux-like operating system to be run on. For testing and getting to know the software, a PC running *Ubuntu 22.04.1 LTS* was chosen. Ubuntu is a free, modern, open-source operating system on Linux, and capable of running both Neper and FEpX. The setup of these programs is written in more detail in the following sections.

4.2.1 Neper Setup

Neper needs to be run on a Linux-like system, so the operating system chosen for testing and getting to know the software was *Ubuntu 22.04.1 LTS*. As many other programs, Neper is reliant on other software to work properly. Some of these are needed before installation, so when doing the setup, it is recommended to install Neper last.

Both *OpenMP*, and *pthread library* are required, but can't be installed. They are features of the compiler, so make sure the operating system contains them.

Since Neper is compiled in C and C++, it needs a C-Compiler. For this *GCC (GNU C Compiler)* was used. It can be installed by using the following commands:

```
sudo apt update
sudo apt install build-essential
sudo apt install gfortran
```

It is not difficult to download and install 3rd-party packages on Linux-systems, including many packages needed by Neper. Simply open a terminal and write the following command:

```
sudo apt-get install {PACKAGE_NAME}
```

The packages that can be installed this way are:

- libgsl-dev
- libnlopt-dev
- libomp-dev
- libscotch-dev
- libpthread-stubs0-dev

Before installing the programs Neper uses, *CMake* should be installed, as it is needed in the installation process of some of the other programs. *CMake* can be installed into your current directory by using the following commands, which download it from GitHub and runs the installation process:

```
git clone https://github.com/Kitware/CMake.git
cd cmake
./bootstrap
sudo make
sudo make install
```

Gmsh is a program used by Neper for meshing. It has its own packages it needs to run, which are given here. Their installation process is the same as for the Neper packages listed above:

- libgmp3-dev
- libhdf5-serial-dev
- libfltk1.3-dev
- xorg openbox
- libpng-dev
- povray
- libblas-dev libblapack-dev

Gmsh requires *OpenSSL*, which can be installed using these commands:

```
git clone https://github.com/openssl/openssl.git
cd openssl
./configure
sudo make
sudo make install
```

Freetype is also needed, and can be acquired by writing these commands in the terminal at the chosen designated directory:

```
git clone https://github.com/freetype/freetype.git
cd freetype
./configure
sudo make
sudo make install
```

Gmsh itself requires *CMake* to be installed. It should be done using these commands:

```
git clone https://gitlab.onelab.info/gmsh/gmsh.git
cd gmsh
mkdir build
```

```
cd build
cmake ..
sudo make
sudo make install
```

Asymptote is a program needed by Neper for pole figure plots in the Visualization (-V) module. To get it, the following commands may be used:

```
git clone https://github.com/vectorgraphics/asymptote.git
cd asymptote
sudo apt-get build-dep asymptote
./autogen.sh
./configure
sudo make all
sudo make install
```

When all dependencies have been successfully downloaded and installed, Neper itself can be set up like this:

```
git clone https://github.com/neperfepx/neper.git
cd neper/src
mkdir build
cd build
cmake ..
sudo make
sudo make install
```

When this is done, Neper should be finished installed on the operating system. To check if it's working properly, write the following command:

```
make test
```

or simply:

```
ctest
```

4.2.2 FEpX Setup

Compared to the Neper setup, setting up FEpX is fairly simple. As Gmsh and Neper, it requires CMake for its installation process. The installation of CMake is noted in the Neper section. Type in the given commands in the terminal at the directory where FEpX is wanted:

```
git clone https://github.com/neperfepx/FEpX.git
cd FEpX/src
mkdir build
cd build
cmake ..
sudo make
sudo make install
```

After this, FEpX should be finished installing and ready for use. To test if it works as intended, type in this command in the terminal, in any directory:

```
make test
```

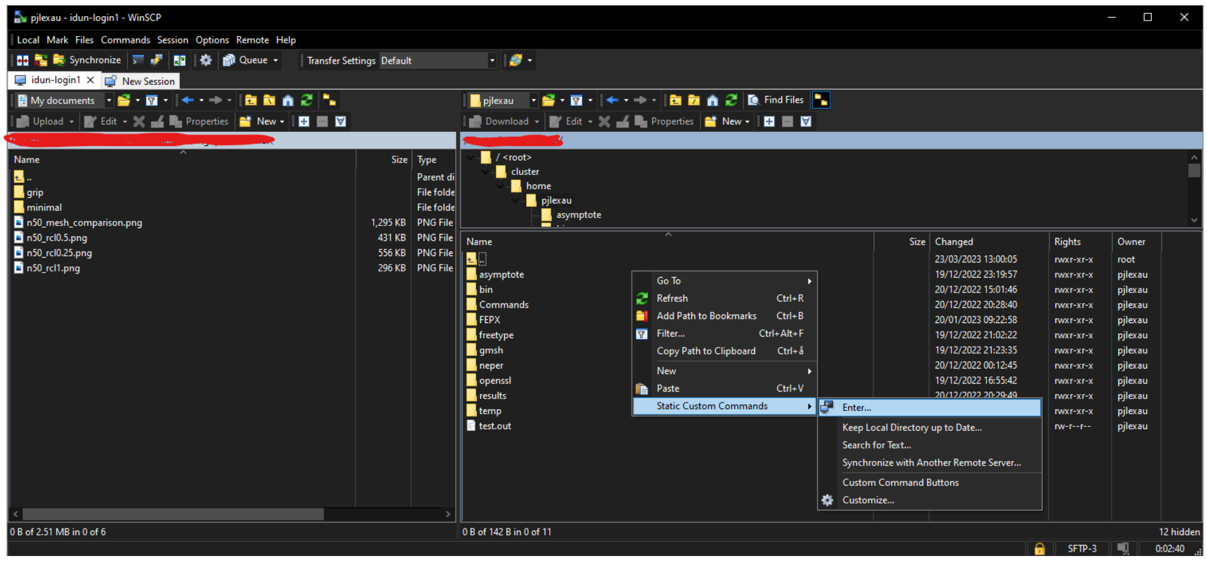
alternatively:

```
ctest
```

4.3 IDUN Setup

All simulations were performed on the NTNU supercomputer IDUN [14]. To access IDUN with a SSH connection outside the NTNU network, the VPN program *Cisco AnyConnect Secure Mobility Client v4.x* can be used. The SSH connection itself can be handled by *WinSCP*, which allows files to be handled, uploading and downloading files to a personal computer, and writing commands in the correct directories. Figure 12 gives a good overview of how to write a command with *WinSCP*.

The setup of Neper and FEpX on IDUN followed some of the same steps as the Linux machine, but a few can be skipped, while other steps need to be added. In short;


pjlexau - idun-login1 - WinSCP
Local Mark Files Commands Session Options Remote Help
Synchronize
Queue
Transfer Settings Default
idun-login1
New Session
My documents
Upload
Edit
Properties
New
pjlexau
Download
Find Files
Go To
Refresh Ctrl+R
Add Path to Bookmarks Ctrl+B
Filter... Ctrl+Alt+F
Copy Path to Clipboard Ctrl+å
New
Paste Ctrl+V
Static Custom Commands
Enter...
Keep Local Directory up to Date...
Search for Text...
Synchronize with Another Remote Server...
Custom Command Buttons
Customize...

[redacted]

```
Name                          Size      Type
..                                      Parent di
grip                                    File folde
minimal                                 File folde
n50_mesh_comparison.png       1,295 KB  PNG File
n50_rcl0.5.png                431 KB    PNG File
n50_rcl0.25.png               556 KB    PNG File
n50_rcl1.png                  296 KB    PNG File

0 B of 2.51 MB in 0 of 6
```

[redacted]

```
/ <root>
  cluster
    home
      pjlexau
        asymptote

Name        Size  Changed              Rights      Owner
..                23/03/2023 13:00:05  rwxr-xr-x   root
asymptote         19/12/2022 23:19:57  rwxr-xr-x   pjlexau
bin               20/12/2022 15:01:46  rwxr-xr-x   pjlexau
Commands          20/12/2022 20:28:40  rwxr-xr-x   pjlexau
FEPX              20/01/2023 09:22:58  rwxr-xr-x   pjlexau
freetype          19/12/2022 21:02:22  rwxr-xr-x   pjlexau
gmsh              19/12/2022 21:23:35  rwxr-xr-x   pjlexau
neper             20/12/2022 00:12:45  rwxr-xr-x   pjlexau
openssl           19/12/2022 16:55:42  rwxr-xr-x   pjlexau
results           20/12/2022 20:29:49  rwxr-xr-x   pjlexau
temp                                   rwxr-xr-x   pjlexau
test.out                               rw-r--r--   pjlexau

0 B of 142 B in 0 of 11                            12 hidden
SFTP-3  0:02:40
```

Figure 12: A command can be performed in the correct directory with WinSCP by righ-clicking inside the folder space, then click *Static Custom Commands* and *Enter...*

installing 3rd-party packages is not needed, but some modules need to be loaded before the installation process can start. Since the *sudo* command is not allowed, the installation paths have to be set manually.

4.3.1 Neper Setup

Before any installations can be done, a CMake module needs to be loaded. This is done by writing

```
module load {CMAKE_MODULE_NAME}
```

In this thesis, *CMake/3.24.3-GCCcore-12.2.0* was used. With CMake available, the installation of the dependencies can start. The installation process for *OpenSSL* is almost the same as for a normal Linux machine. As the *sudo* command can't be used, and the */usr/local* folder isn't available, the path must be set manually. Start by creating a *bin* folder in the users main directory. Then write the following commands in the terminal in the designated directory:

```
git clone https://github.com/openssl/openssl.git
cd openssl
```

```
./configure --prefix=/cluster/home/{USERNAME}/bin
make
make install
```

Freetype is installed much in the same way as *OpenSSL* is. Go to the directory where the folder should be downloaded and type:

```
git clone https://github.com/freetype/freetype.git
cd freetype
./configure --prefix=/cluster/home/{USERNAME}/bin
make
make install
```

Gmsh works much the same way as well:

```
git clone https://gitlab.onelab.info/gmsh/gmsh.git
cd gmsh
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/cluster/home/{USERNAME}/bin ..
make
make install
```

Neper again uses the same method as *Gmsh*:

```
git clone https://github.com/neperfepx/neper.git
cd neper/src
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/cluster/home/{USERNAME}/bin ..
make
make install
```

With this done, Neper should be able to run normally by using its commands in the IDUN terminal.

4.3.2 FEpX Setup

FEpX, as Neper, is installed much the same way as on a normal Linux machine. It as well needs a CMake module to be loaded, and that the installation path is set manually during the installation process. Go in the terminal to the directory where FEpX should be installed, and write the following commands:

```
git clone https://github.com/neperfepx/FEpX.git
cd FEpX/src
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=/cluster/home/{USERNAME}/bin ..
make
make install
```

4.4 Simulations setup

This section explains how the different simulations with FEpX can be done. Descriptions of parameters, mesh creation and steps to be followed can be found here.

Since IDUN is a computer cluster shared between several teams, a way to queue intensive operations is needed. IDUN solves this by using the cluster resource management system called Simple Linux Utility Resource Management *Slurm* [16]. To queue a job with Slurm, a slurm-file is needed. Following is a typical example of such a file:

```
#!/bin/sh
#SBATCH --partition=GPUQ
#SBATCH --account=SHARE-nv-ima
#SBATCH --time={hhh:mm:ss}
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=16
#SBATCH --mem=60G
#SBATCH --job-name="{JOB_NAME}"
#SBATCH --output=test.out
```

```
#SBATCH --mail-user={USER_EMAIL}
#SBATCH --mail-type=ALL

WORKDIR=${SLURM_SUBMIT_DIR}
cd ${WORKDIR}

module load OpenMPI/4.1.4-GCC-11.3.0

mpirun --oversubscribe -np 16 fepx
neper -S -step 20 -o results .
```

Memory, number of cores needed, time until timeout, e-mail for updates, and a filename for any output data are among the information included. Mark that before any commands run, its important to load any necessary modules, in this case *OpenMPI*. The commands in the file will then be performed in the order they're listed.

4.4.1 Uniaxial Grip and Uniaxial Minimal Simulations

To run a simulation a mesh is necessary. By using Neper's generation module, a tessellation of a representative volume element can be generated. The meshing module can then create meshes out of the tessellation which the simulations will run with. For both the Uniaxial Grip simulation and the Uniaxial Minimal simulation the same meshes can be used. The given Neper commands will generate the tessellation:

```
neper -T -n 50 -reg 1 -morpho gg -o {TESSELLATION_NAME}
```

This creates a cube made up of fifty grains. The grains are generated with grain growth morphology, and since regularization is active, tiny edges and faces that can complicate the meshes will be absorbed. The meshes can then be created using this Neper command:

```
neper -M {TESSELATION_NAME}.tess -order 2 -part 16 -rcl {RCL_VALUE} -o
   {MESH_NAME}
```

The *rcl* value stands for *relative characteristic lengths* of the elements, and controls how fine the finished mesh will become. A smaller value gives a finer mesh, while a larger value gives a rougher mesh. The rcl values of the three meshes generated for these simulations

Table 1: The different meshes and their rcl values

Mesh type	rcl
Rough mesh	1.00
Average mesh	0.50
Fine mesh	0.25

can be seen in Table 1. The generated tessellation and the three meshes can be seen in Figure 13.

The simulation.config file tells FEpX what kind of material that is tested, and how it is to be deformed. The material parameters for both the grip and minimal simulations can be found in Table 2 and Table 3. In the deformation history, the deformation is controlled by uniaxial strain target, which means it is exposed to a constant strain rate until a specific load of uniaxial strain state is met. All simulations were performed with 18 strain steps, with each increment increasing with 0.045, and 0.005 % target strain every other increment:

Table 2: Single crystal elastic constants

Phase	Type	C_{11}[MPa]	C_{12}[MPa]	C_{44}[MPa]
α	FCC	245000	155000	62500

Table 3: Initial slip system strengths and other plasticity parameters

Phase	m[-]	$\dot{\gamma}_0$[1/s]	h_0[MPa]	g_0[MPa]	g_{s0}[MPa]	n[-]
α	0.05	1	200	210	330	1

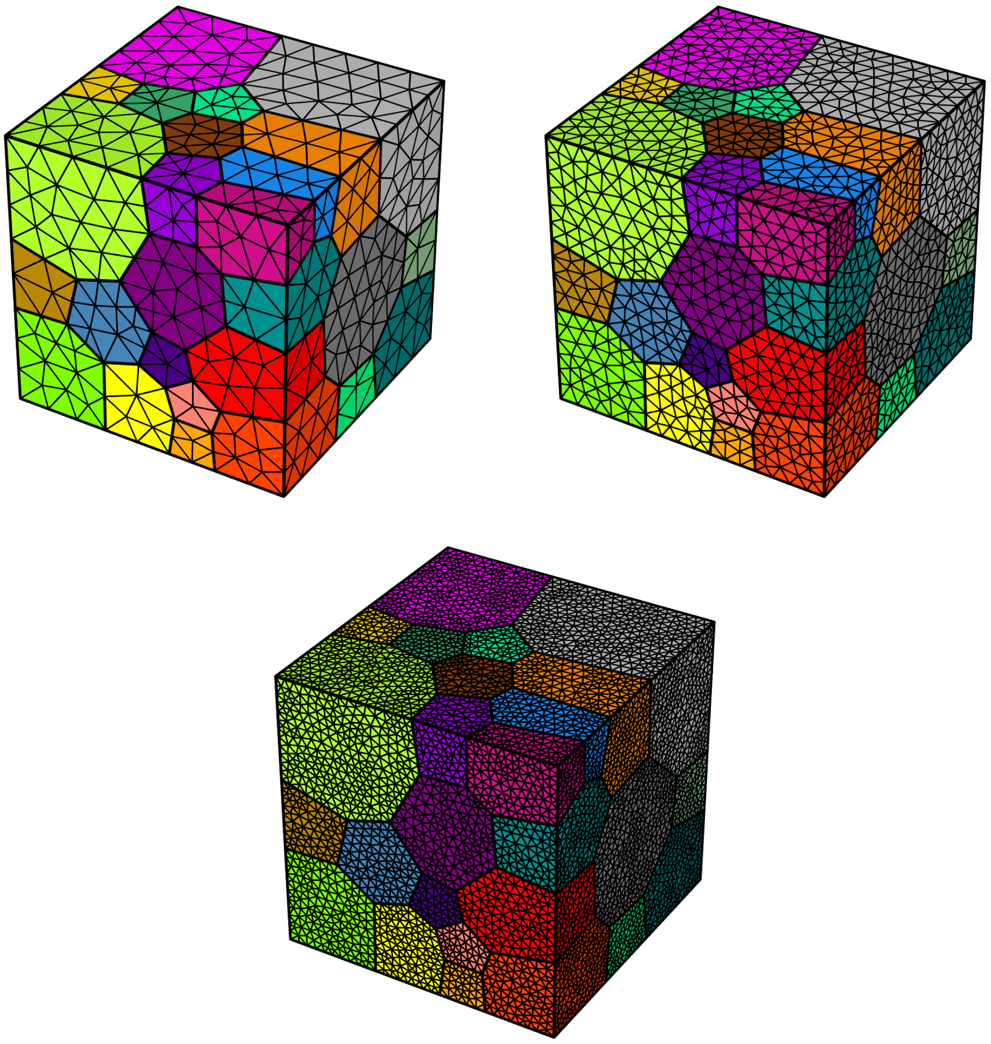


Figure 13: In the top left corner the rough mesh with rcl=1.00 can be seen. In the top right corner is the average mesh with rcl=0.50, and at the bottom is the fine mesh with rcl=0.25.

```
## Deformation History

    def_control_by uniaxial_strain_target

    number_of_strain_steps 18

    target_strain 0.045 9 suppress_data
    target_strain 0.05 1 print_data
    target_strain 0.095 9 suppress_data
    target_strain 0.1 1 print_data
    target_strain 0.145 9 suppress_data
    target_strain 0.15 1 print_data
    target_strain 0.195 9 suppress_data
    target_strain 0.2 1 print_data
    target_strain 0.245 9 suppress_data
    target_strain 0.25 1 print_data
    target_strain 0.295 9 suppress_data
    target_strain 0.3 1 print_data
    target_strain 0.345 9 suppress_data
    target_strain 0.35 1 print_data
    target_strain 0.395 9 suppress_data
    target_strain 0.4 1 print_data
    target_strain 0.445 9 suppress_data
    target_strain 0.45 1 print_data
```

To prevent too much unnecessary data from being printed, only the last increment to every 5% strain target will be printed. The simulations are done when a strain of 45% is reached.

The boundary conditions are the only difference between the uniaxial grip and uniaxial minimal simulations. Both will have the same strain rate, 0.01, and the same loading direction and loading face:

```
## Boundary Conditions

    boundary_conditions {uniaxial_grip/uniaxial_minimal}

    loading_direction Z
    loading_face Z_MAX

    strain_rate 1e-2
```

The data that should be output and is needed for the results are:

- Coordinates
- Equivalent plastic strain
- Stress tensor

4.4.2 FEpX and Abaqus Simulations

The simulations to compare FEpX and Abaqus are similar to the uniaxial grip simulations in the last section. The mesh is the same as the average mesh quality, which can be seen in the top right corner of Figure 13. The material parameters are mostly the same, with one small difference. Three simulations are done with both FEpX and Abaqus with different rate sensitivity exponents (m). The different simulations and exponents are listed in Table 4. The deformation history is simpler in these FEpX simulations compared to those in the last section. There are less steps, and they stop at 40% strain load. Instead more increments are needed to lower the time steps:

Table 4: The different rate sensitivity exponents used in the different simulations

Simulation	m[-]
1	0.05
2	0.01
3	0.001

```
## Deformation History

   def_control_by uniaxial_strain_target

   number_of_strain_steps 8

   target_strain 0.05 200 print_data
   target_strain 0.10 200 print_data
   target_strain 0.15 200 print_data
   target_strain 0.20 200 print_data
   target_strain 0.25 200 print_data
   target_strain 0.30 200 print_data
   target_strain 0.35 200 print_data
   target_strain 0.40 200 print_data
```

4.4.3 Rolling Simulations

The uniaxial rolling simulations work a little bit different than the other FEpX simulations. As their boundary conditions are slightly more complicated a simulation.bcs file is required to describe the boundaries. Their material properties are also different, to better simulate a cold rolling of a AA1050 aluminium alloy. Two different tessellations creating two different meshes will be simulated. A depiction of the simulation cubes can be seen in Figure 14. The first cube and mesh will be created as the cube and mesh that can be seen top left on Figure 13, but with 100 grains instead of 50. The other cube will have 1000 grains, generated with centrodial morphology instead of grain growth. When creating the mesh, the parameter absolute characteristic lengths cl is used instead of rcl, and its value is set to 10. This is to make sure the mesh will be as rough as Neper allows, to reduce the total number of nodes and elements. The Neper commands to create the new mesh are:

```
   neper -T -n 1000 -reg 1 -morpho centroidal -o {TESSELLATION_NAME}
   neper -M {TESSELLATION_NAME}.tess -order 2 -part 16 -cl 10 -o {MESH_NAME}
```

The boundary conditions for a rolling simulation must be defined in an external file

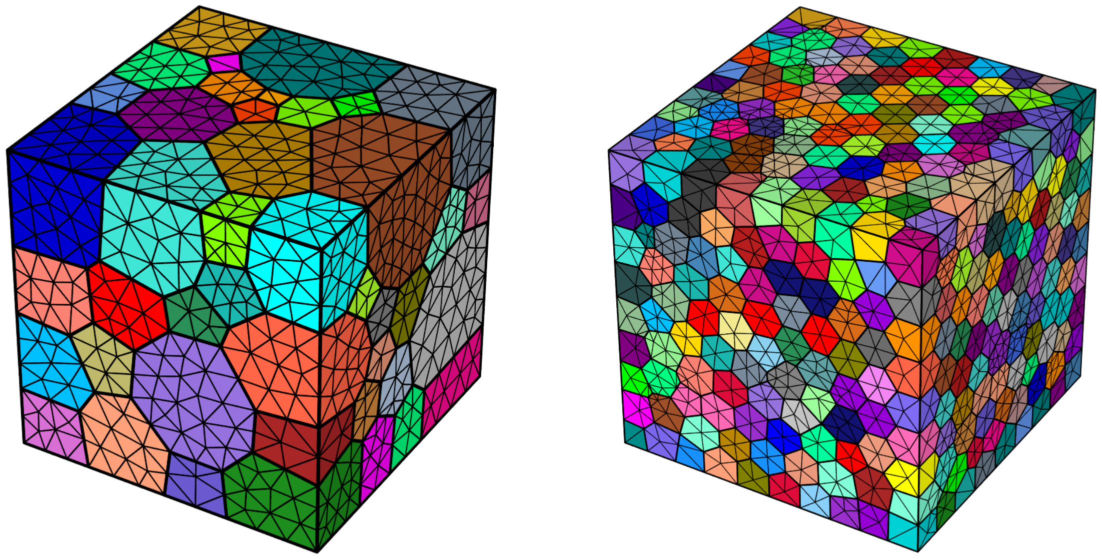


Figure 14: The two different tessellations and meshes for the rolling simulation. The left cube has 100 grains with grain growth morphology and a rcl of 1.00. The right cube has 1000 grains with centroidal morphology and a cl of 10.

for FEpX. In such a simulation, two faces must be kept static in he direction they are facing. One other face must be kept static in its direction, while the opposite face is given a velocity into the cube. One corner node must be kept completely static to prevent rotations of the cube. To accomplish this condition for the relevant nodes in the two meshes, a script was written to automate the process. The script can be found in Appendix A.1. In it the nodes with a y-value lower than 0.001, and higher than 0.999 has their y velocity set to 0. The nodes with z-values higher than 0.999 is given a z-velocity of 0, while the nodes with a z lower than 0.001 needs to have a positive z-velocity of 0.01, to make the compression happen. The node with an x, y, and z value lower than 0.001 has its velocity in all directions set to 0. Figure 15 shows a cube with these constraints. The material properties for the simulation can be seen in Table 5 and Table 6.

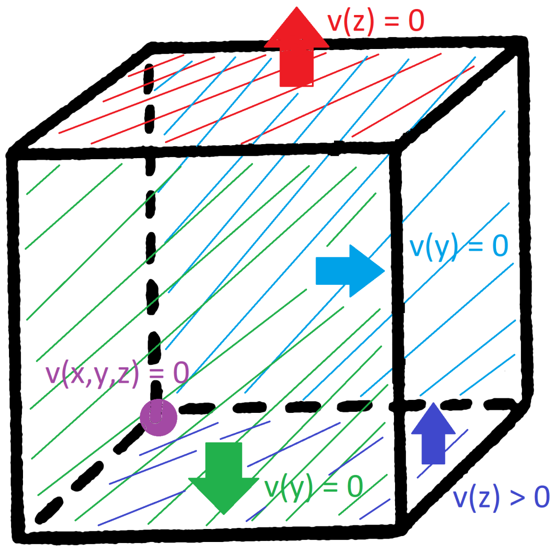


Figure 15: Boundary conditions for a rolling simulation. The y-velocity for the two y-faces is set to 0. The z-velocity of the top face is also set to 0. The 0,0,0 corner node is set to 0 in all directions. The bottom face is pushed up to create compression, while the two x-faces are unconstrained.

Table 5: Single crystal elastic constants

Phase	Type	C_{11}[MPa]	C_{12}[MPa]	C_{44}[MPa]
α	FCC	106750	60410	28340

Table 6: Initial slip system strengths and other plasticity parameters

Phase	m[-]	$\dot{\gamma}_0$[1/s]	h_0[MPa]	g_0[MPa]	g_{s0}[MPa]	n[-]
α	0.02	0.001	485	12.9	30	1

Chapter 5

Results and Discussion

5.1 Results

5.1.1 Uniaxial Grip and Uniaxial Minimal Simulations

When running simulations in FEpX, it was discovered that a too high strain target would distort the mesh too much, and the simulation would fail. This is more prevalent the finer the mesh the simulation is using, but at higher strain targets all simulations would fail. It was then decided to stop the simulations when a target of 45% was reached, as that's a load that even the simulation with the finest mesh can reach.

When a simulation is completed the simulation module of Neper is used to post-process the data into a form that is more readable for both humans and the visualization module of Neper. To get the images from the resulting post-processed data, a shell-script with Neper and imagemagic commands was used. The shell-script can be viewed in Appendix A.2.

Table 7: Overview of the uniaxial grip and uniaxial minimum simulation times for the different mesh qualities.

Type	rcl[-]	strain rate[1/s]	dt[s]	t_{max}[s]	Simulation time[s]
Uniaxial grip	1	0.01	0.5	45	213
	0.5	0.01	0.5	45	2135
	0.25	0.01	0.5	45	80133
Uniaxial minimum	1	0.01	0.5	45	484
	0.5	0.01	0.5	45	10795
	0.25	0.01	0.5	45	182273

Figure 16 shows the equivalent plastic strain on the different quality meshes after the uniaxial grip simulation has completed at 45% target strain. Figure 17 shows the stress33 values for the same meshes. In Figure 18 is the results of equivalent plastic strain after a uniaxial minimal simulation has completed at the 45% strain target. The stress33 effect after uniaxial minimal simulation has completed at the 45% strain target is visible in Figure 19. Table 7 shows the simulation times.

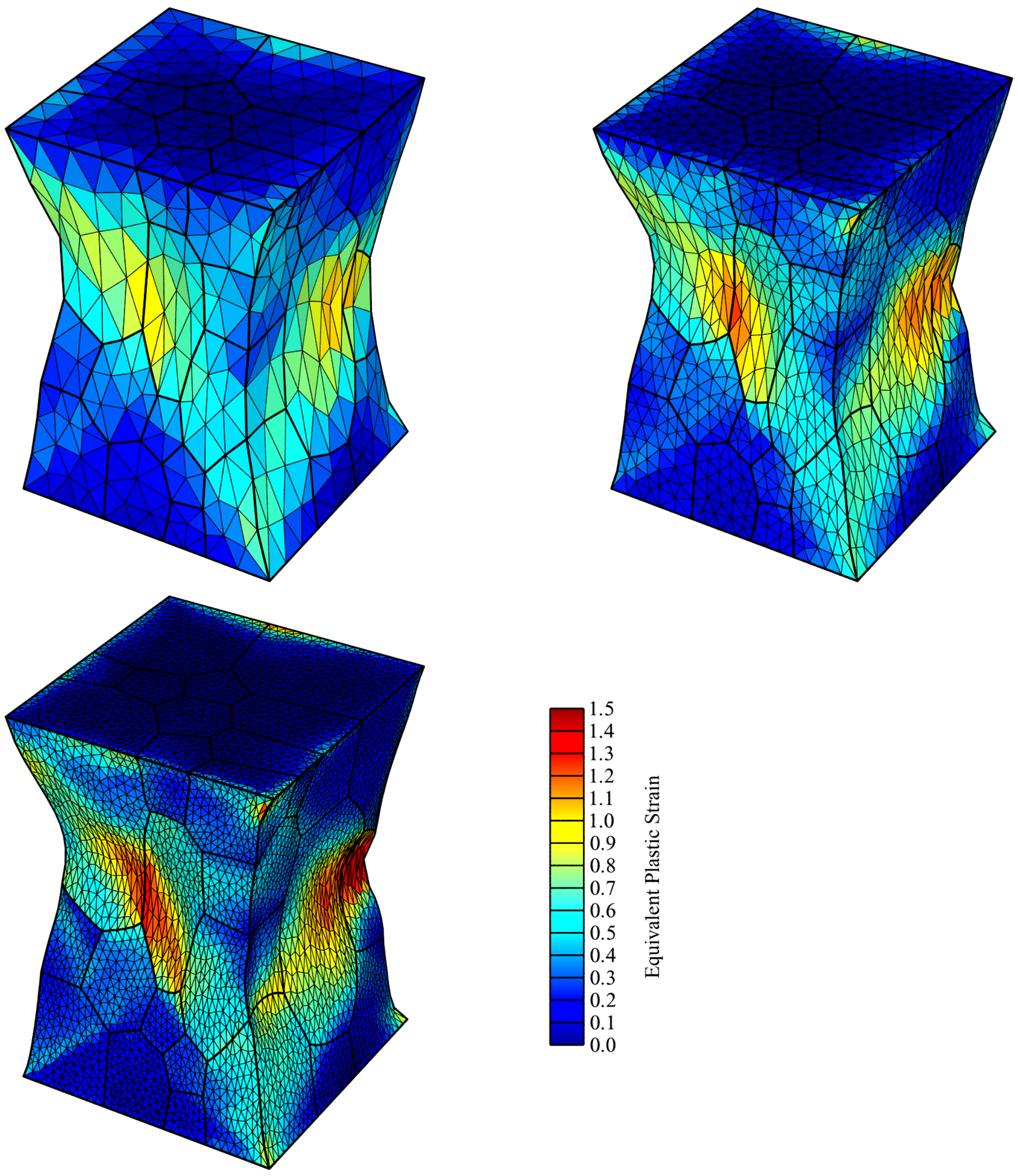


Figure 16: The equivalent plastic strain on three cubes with different mesh qualities after a uniaxial grip simulation has completed at 45% strain target.

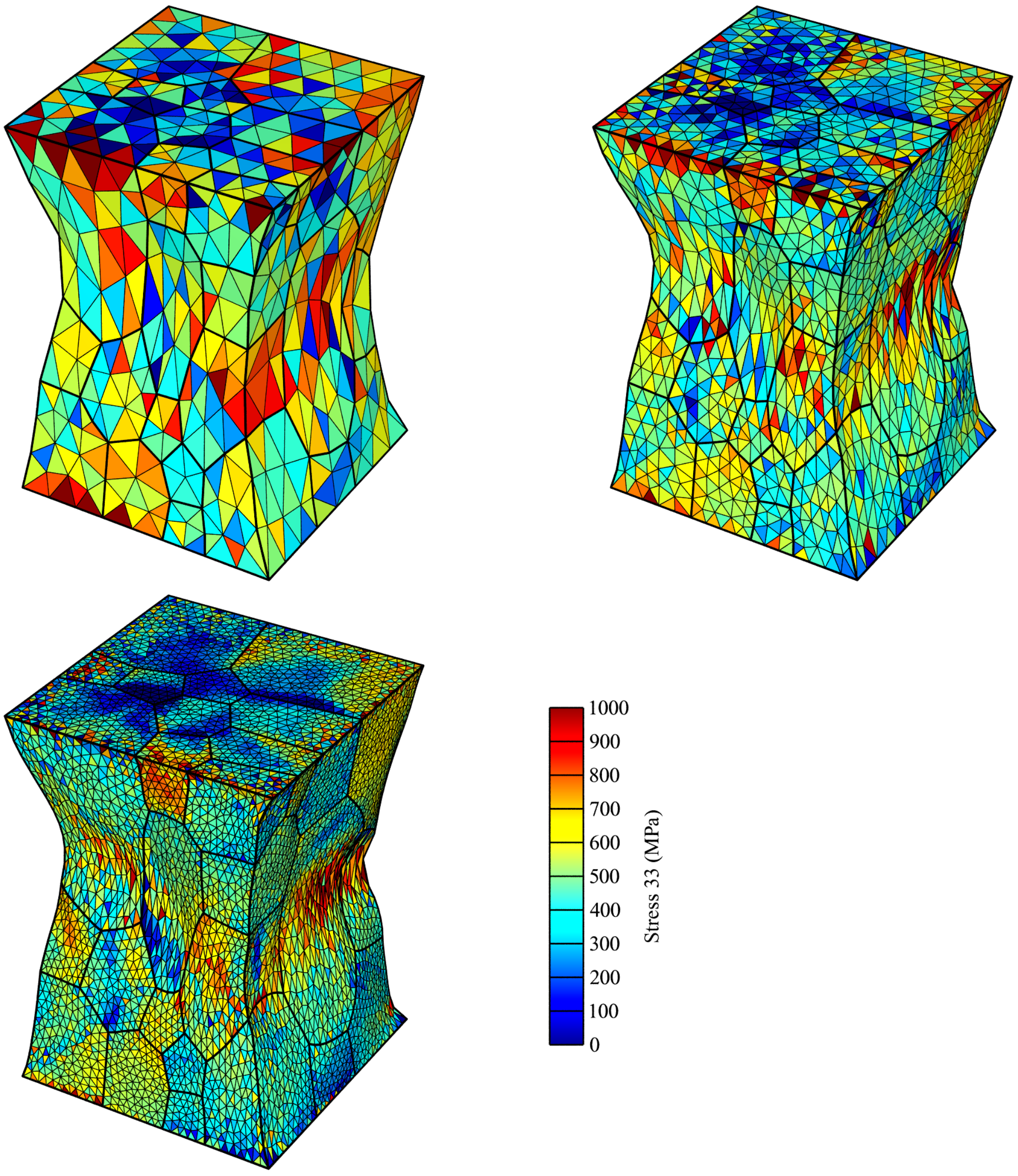


Figure 17: The stress33 on three cubes with different mesh qualities after a uniaxial grip simulation has completed at 45% strain target.

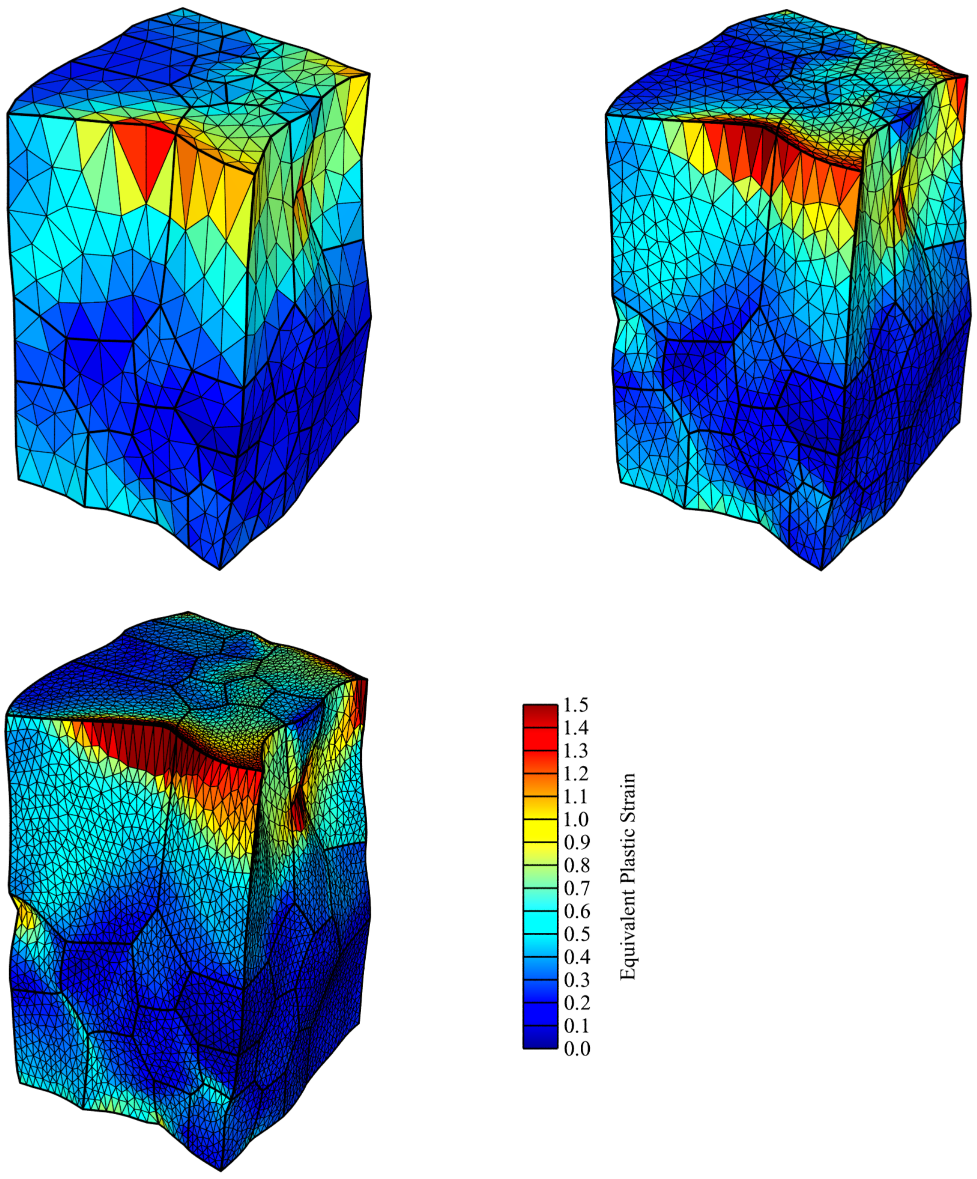


Figure 18: The equivalent plastic strain on three cubes with different mesh qualities after a uniaxial minimal simulation has completed at 45% strain target.

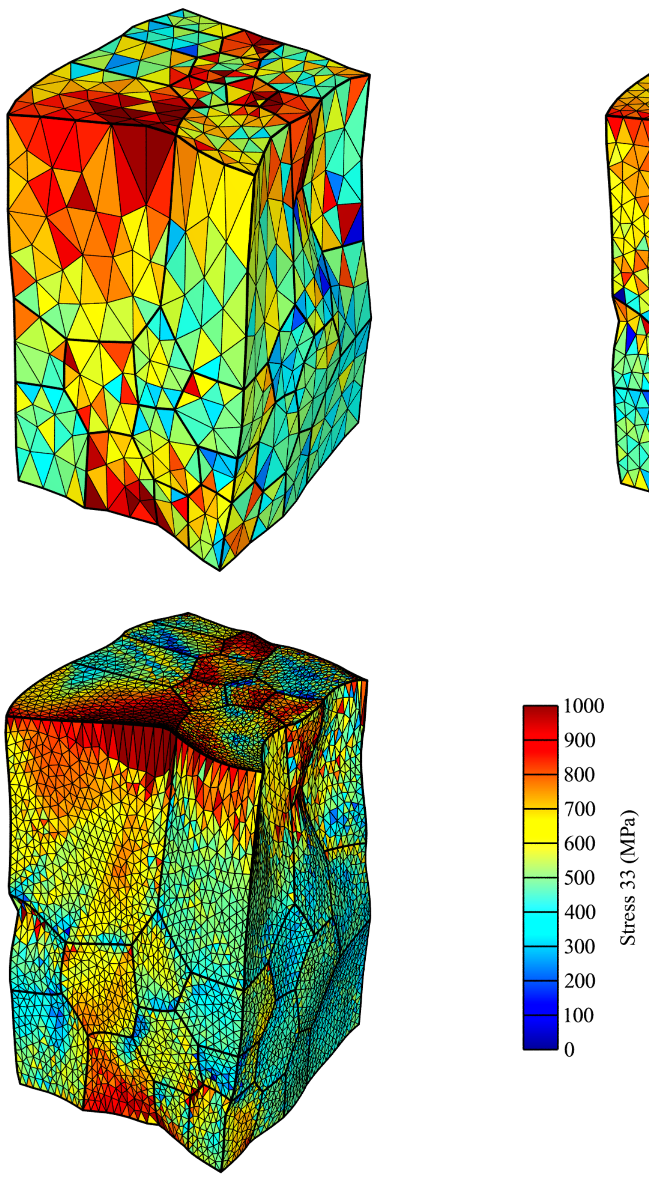


Figure 19: The stress33 on three cubes with different mesh qualities after a uniaxial minimal simulation has completed at 45% strain target.

5.1.2 FEpX and Abaqus Simulations

Figure 20 depicts the equivalent plastic strain on the mesh with different rate sensitivity exponents in the FEpX uniaxial grip simulations. In Figure 21 the results of the equivalent stress from the same simulations can be seen. The grain structure of the cubes that went through the Abaqus uniaxial grip simulations is in Figure 22. Figure 23 contains the Abaqus equivalent plastic strain results, while Figure 24 has the von Mises stress. The simulation times for the FEpX simulations are in Table 9.

The computational times performed on IDUN, on the same number of CPUS (16) as the FEpX simulations are shown in the Table below.

Table 8: Overview of the FEpX uniaxial grip simulation times for the different rate sensitivity exponents.

Type	rcl[-]	m[-]	strain rate[1/s]	dt[s]	t$_{max}$[s]	Simulation time[s]
Uniaxial grip	0.5	0.05	0.01	0.5	40	2226
	0.5	0.01	0.01	0.25	40	4868
	0.5	0.001	0.01	0.025	40	15782

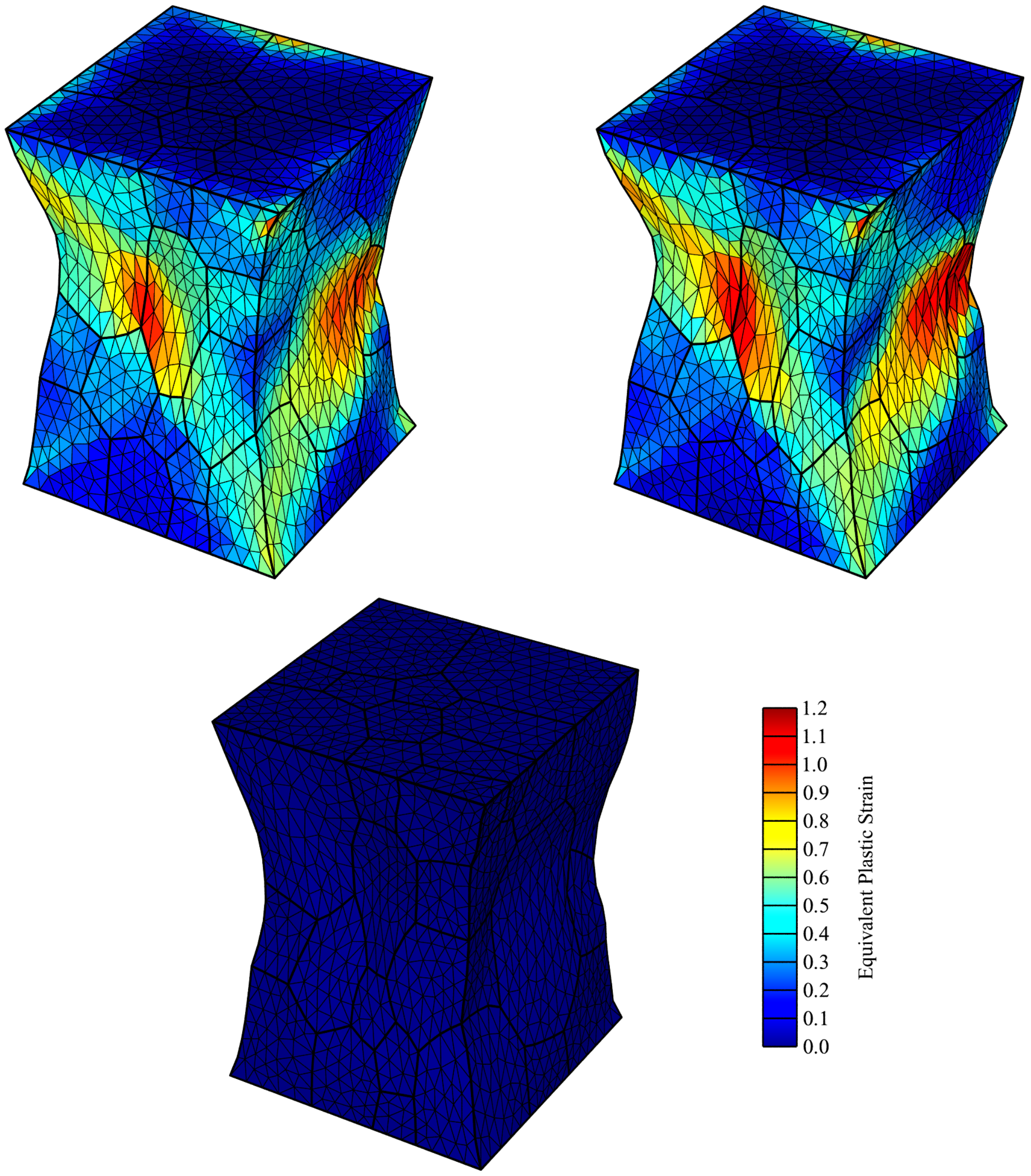


Figure 20: The equivalent plastic strain on three cubes with different rate sensitivity exponents after a FEpX uniaxial grip simulation has completed at 40% strain target. The top left has m=0.05, top right has m=0.01, and bottom has m=0.001.

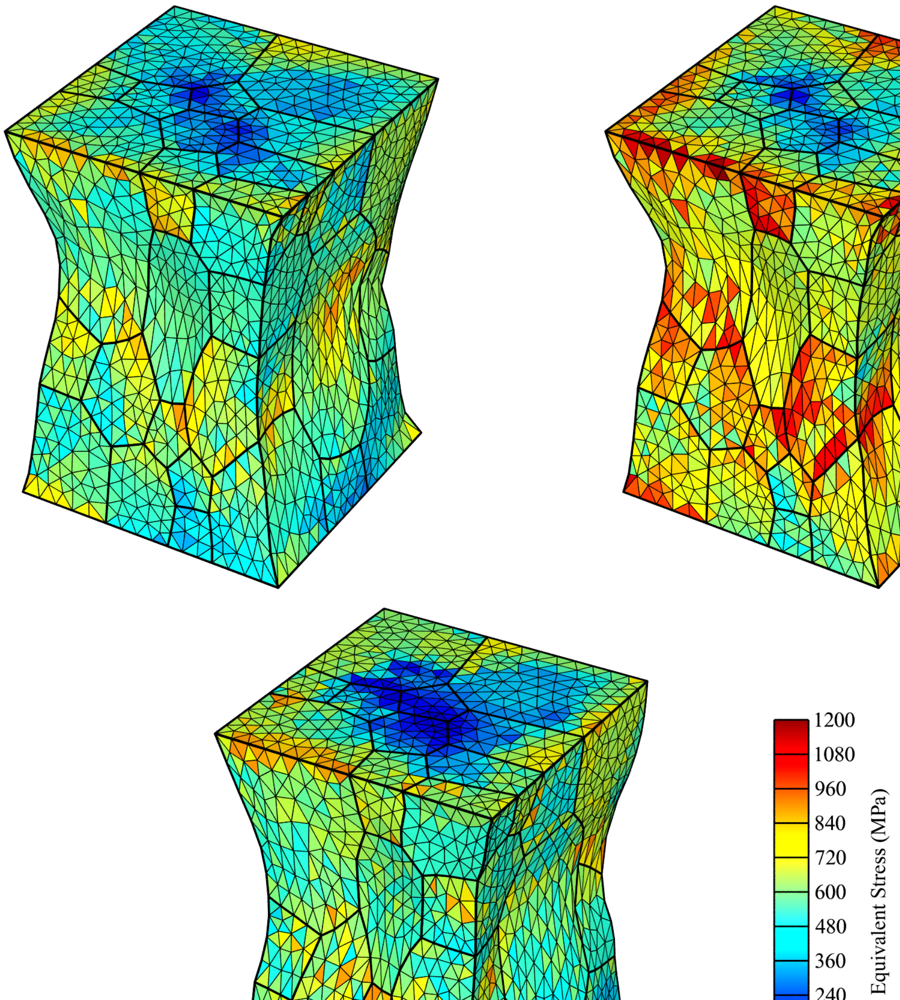


Figure 21: The equivalent on three cubes with different rate sensitivity exponents after a FEpX uniaxial grip simulation has completed at 40% strain target. The top left has m=0.05, top right has m=0.01, and bottom has m=0.001.

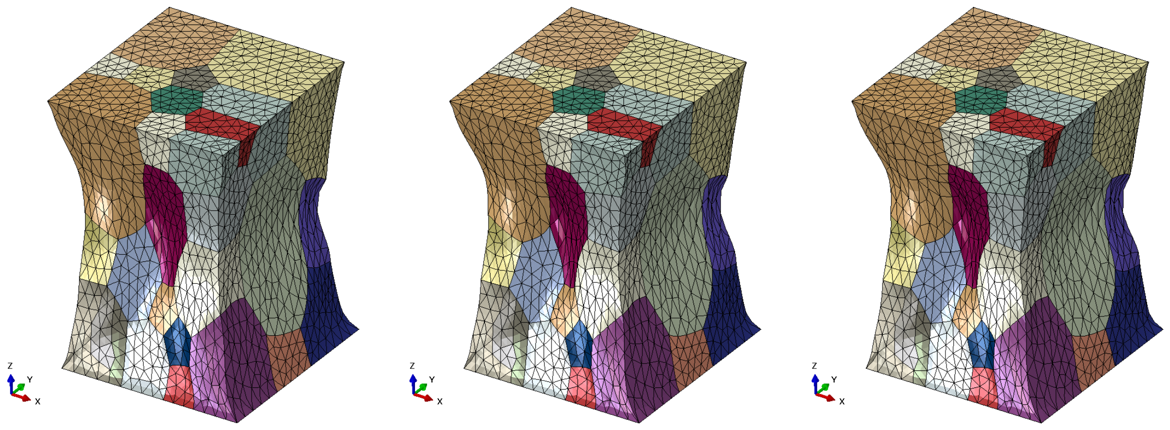


Figure 22: The grain structure of three cubes with different rate sensitivity exponents after an Abaqus uniaxial grip simulation has completed at 45% strain target. The left has m=0.05, middle has m=0.01, and right has m=0.001.

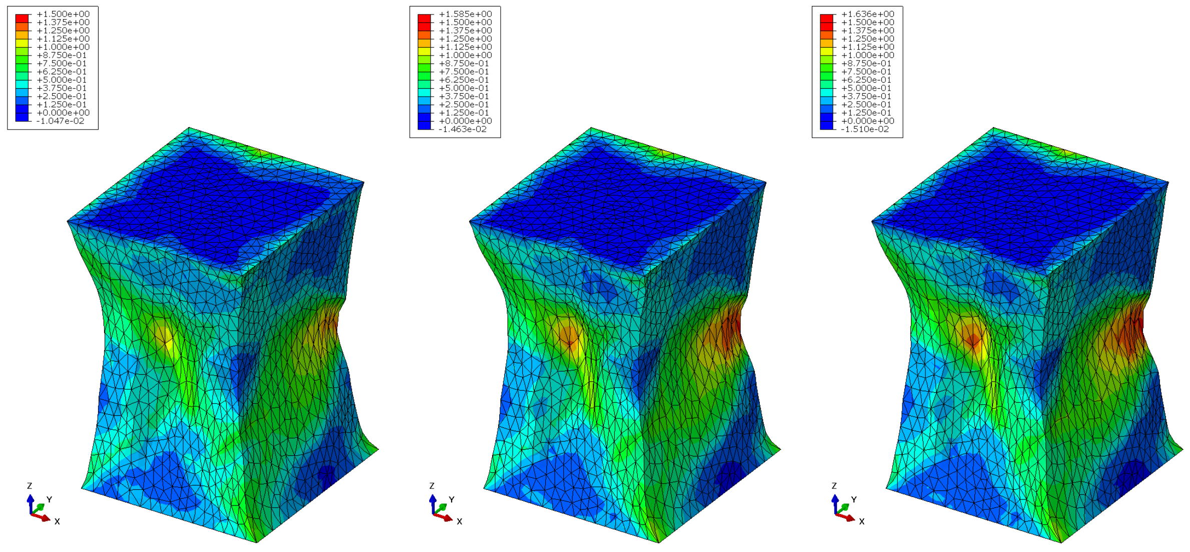


Figure 23: The equivalent plastic strain on three cubes with different rate sensitivity exponents after an Abaqus uniaxial grip simulation has completed at 45% strain target. The left has m=0.05, middle has m=0.01, and right has m=0.001.

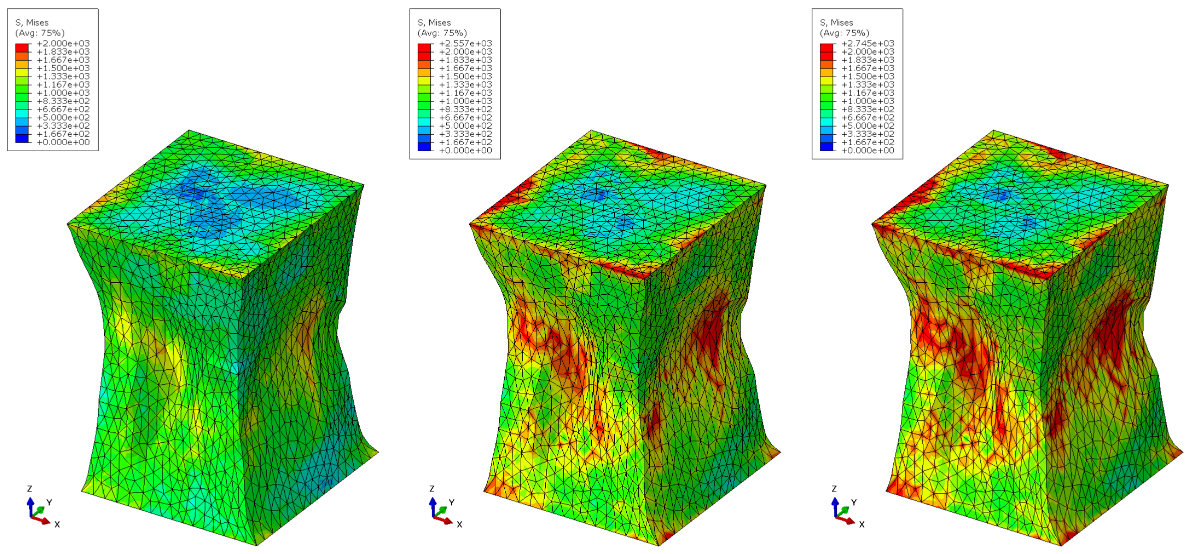


Figure 24: The von Mises stress on three cubes with different rate sensitivity exponents after an Abaqus uniaxial grip simulation has completed at 45% strain target. The left has m=0.05, middle has m=0.01, and right has m=0.001.

Table 9: Overview of the Abaqus uniaxial grip simulation times for the different rate sensitivity exponents.

Type	rcl[-]	m[-]	strain rate[1/s]	dt[s]	t_{max}[s]	Simulation time[s]
	0.5	0.05	0.01	0.5	45	1800
Uniaxial grip	0.5	0.01	0.01	0.25	45	3300
	0.5	0.001	0.01	0.025	45	25620

5.1.3 Rolling Simulations

Figure 25 contains the equivalent plastic strain results after the uniaxial rolling simulation on the 100 grain mesh. The results from the 1000 grain mesh can be seen in Figure 26. Both simulations stopped after reaching a strain target of 25% because of mesh distortions. In Figure 27 is the density pole figure for the 100 grain tessellation, while Figure 28 contains the density pole figure for the 1000 grain tessellation. The simulation times for both simulations are in Table 10. The Figure 29 shows the corresponding ODF of the texture after the deformation of the 1000 grain mesh at the 25% rolling reduction.

Table 10: Overview of the FEpX uniaxial rolling simulation times for the tessellation and meshes.

Type	grain number[-]	strain rate[1/s]	dt[s]	t_{max}[s]	Simulation time[s]
Uniaxial rolling	100	0.01	0.025	25	4174
	1000	0.01	0.125	25	55645

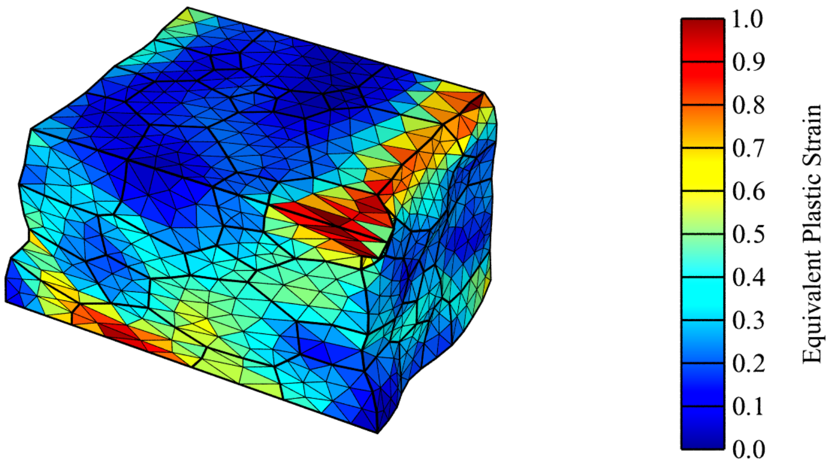


Figure 25: The equivalent plastic strain on a 100 grain mesh after a 25% strain uniaxial rolling simulation has completed.

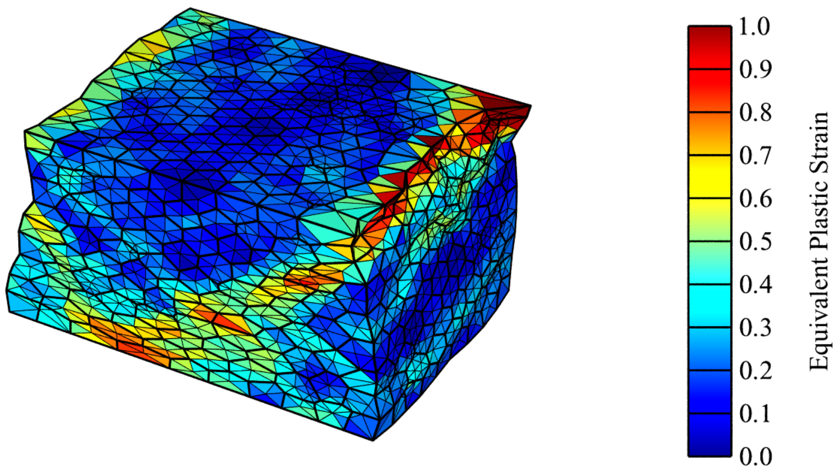


Figure 26: The equivalent plastic strain on a 1000 grain mesh after a 25% strain uniaxial rolling simulation has completed.

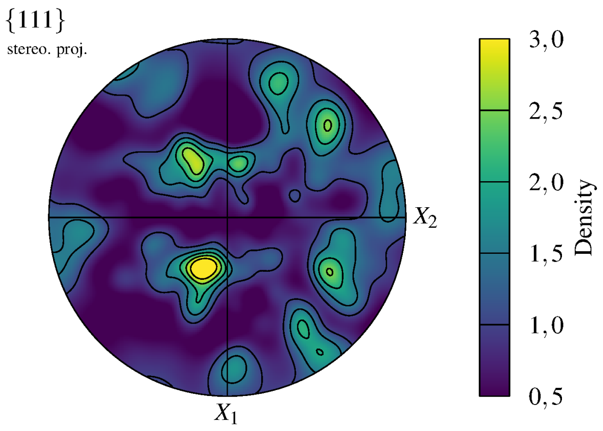


Figure 27: The orientation density pole figure of the 100 grain mesh after the simulation has completed.

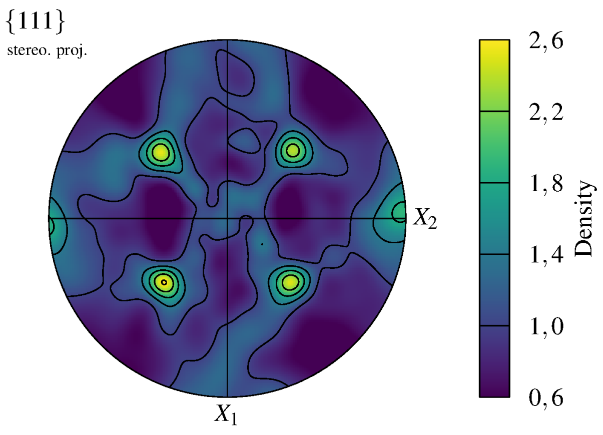


Figure 28: The orientation density pole figure of the 1000 grain mesh after the simulation has completed.

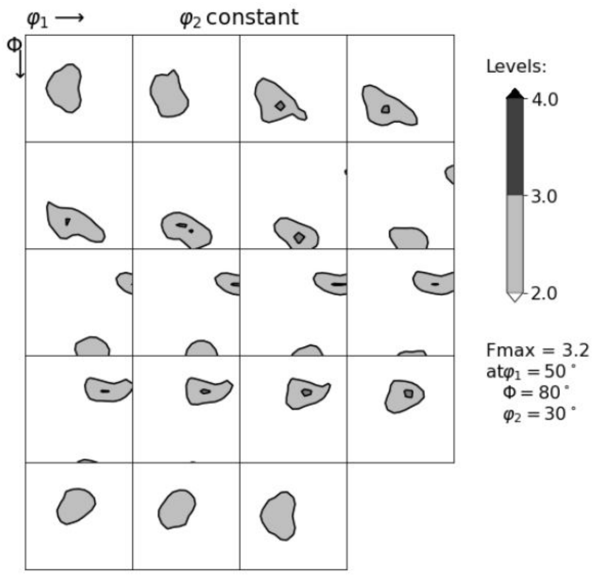


Figure 29: ODF of the 1000 grain mesh after deformation at the 25% rolling reduction.

5.2 Discussion

5.2.1 Uniaxial Grip and Uniaxial Minimal Simulations

Abaqus performed simulation with the uniaxial grid boundary conditions by the factor of 0.65 faster for the strain rate sensitivity $m = 0.05$, and by the factor of 0.8 faster for $m = 0.01$. The results for $m = 0.001$ were timed faster by FEpX but since they are incomplete, one cannot trust the total time measured.

From Figure 16, Figure 17, Figure 18, and Figure 19 it is visible that a finer mesh with more elements create a more accurate and detailed result. As the simulation times in Table 7 shows however, those simulations are much more computationally intensive, and require much more time than simulations with meshes that have less elements. It is therefore a trade-off between how good simulations that are wanted, and how much time and resources that should be spent on them.

From table 7 it is also clear that uniaxial minimal simulations are more computationally intensive than the uniaxial grip simulations. This could be because the uniaxial minimal have less boundary conditions which gives the elements more room to move. Figure 18 and Figure 19 shows that the cube has become more distorted than it is in the uniaxial grip simulations. More node-movement seems to make the simulations more intensive.

FEpX is capable of running both uniaxial grip and uniaxial minimal boundary conditions simulations, and with varying mesh quality. There is a clear correlation between mesh element amount, and node movement on the total simulation time.

5.2.2 FEpX and Abaqus Simulations

Both FEpX and Abaqus seem capable of handling varying rate sensitivity exponents, but FEpX's results cannot be relied on if the exponent is too low. As is shown in Figure 20, when the exponent is set to 0.001, it fails to compute any equivalent plastic strain. In Figure 21 FEpX manages to calculate the equivalent stress, but it doesn't follow the trend of increasing stress with lower rate sensitivity exponent that can be seen with Abaqus in Figure 24. The trend of increasing stress is visible for the FEpX simulations at m=0.05

and m=0.01, but falls at m=0.001. Another sign the results at this rate sensitivity exponent is unreliable.

Abaqus also works better at higher strain targets. While Abaqus manages to simulate a strain up to 45%, FEpX at lower rate sensitivity exponents fails when the strain reaches higher than 40%., but with remeshing it can reach higher strains [1]. If the strain forces the nodes too far from each other it can reach a point where the calculations cannot converg on a solution. With remeshing the nodes are redistributed at fitting distances, and the simulation can continue to higher strains.

From these simulations it is clear that the FEpX results become unreliable at rate sensitivity exponents lower than 0.01, at least at a value of 0.001. Abaqus in contrast shows no issues of handling rate sensitivity exponents of that value. FEpX cannot handle too high strain targets and fails sooner than Abaqus. With remeshing it may be able to continue simulating higher strain targets.

Even though 25% rolling reduction is too little to develop pronounced β-fiber, that is the signature of the cold rolled textures. Never the less weak β-fiber with a maximum ntensity of 3.2 can be clearly observed from the picture in Figure 29.

5.2.3 Rolling Simulations

The rolling simulations seen in Figure 25 and Figure 26 had to stop at 25% strain. The reason for the failure before a 30% strain could be reached is uncertain, but is most likely because of the shear bands visible in the figures. At a top corner of both simulations the mesh is clearly distorted to the point that the Finite-Element method is close to breaking. Such shear bands are common when rolling, so the simulations are capable of recreating natural phenomena seen in experimental data.

The density pole-figures in Figure 27 and Figure 28 show signs of a symmetry found in the orientations of more complete rolling simulations. The fact that the rolling simulations stopped at 25% strain is probably the reason that the pole-figures doesn't show more of the orientation density symmetry found in rolling simulations. With remeshing, a higher strain target could be attempted which should give a density pole-figure that looks more like the theoretical figure.

As in the uniaxial grip and uniaxial minimal simulations, the total simulation times for rolling simulations in Table 10 conveys the trade-off between more elements in the mesh, and computational time. Figure 26 shows localization and shear bands more clearly than Figure 25, but also requires over 10x more total simulation time.

FEpX is capable of running rolling simulations, but without remeshing they stop at a low strain target of 25%. Pole-figures created from the resulting orientations show signs of the density symmetry found in more complete rolling simulations. With remeshing and a higher strain target a pole-figure with a density symmetry more like the theoretical figure may be achieved.

Chapter 6

Conclusions and Future Work

This thesis has created an overview of the FEpX code-base, and its algorithms. It has created visual outline of what fortran modules and their subroutines that are being used, and what modules call the different subroutines. Unfortunately FEpX has no easily found access point where other algorithms can be plugged in. It was therefore found to be unfeasible to switch the core of the FEpX algorithms with the umat crystal plasticity code. It is technically possible, but will require extensive rewriting of both code-bases, and probably several subroutine calls between them instead of just one contact point. Such a configuration with the FEpX frontend and the umatCP is unforunately outside the scope of this thesis.

Neper has been used for many aspects of this thesis. All the different modules of Neper has been needed for parts of the simulations. The generation and meshing modules have been used and explained for creating the meshes that have been simulated. The simulation module is critical to post-process the raw data after the FePX simulations, and the visualization module has created almost all the figures in the Section 5.

FePX have been used to simulate two different simple boundary condition cases, and one more complicated custom boundary condition. With these simulations FEpX is better understood, and its advantages and limitations are found.

When comparing simulations of Abaqus and FepX, Abaqus was found to be faster, handling cases better, and more user-friendly. This is to be expected as Abaqus is a paid for product.

FEpX can handle plane-strain conditions, though only to 25% strain before it requires remeshing. The results are promising though, and if another simulation can be attempted in the future with remeshing, it would be interesting to see how far it could go.

What is the code-structure of FEpX, and how easy is it to substitute it with other algorithms?

FEpX is coded in Fortran, but has a fairly complicated structure where new code cannot easily be inserted. Different fortran modules have subroutines that call subroutines in other modules in in a bit of a criss-cross pattern that makes it hard to substitute the algorithms.

What are the advantages of using FEpX compared to Abaqus?

FEpX is free, and open-source, but in other areas fall short compared to Abaqus. Abaqus is more user-friendly, and handles higher strains, boundary conditions, and lower strain sensitivity exponents better. It is farily complicated to insert own algorithms into FEpX, but it is possible.
Does FEpX have any failings or limitations?

When testing FEpX some limitations where found. It is only capable of handling 2nd order 10-node tetrahedral elements in the mesh-file. This may cause some complications when comparing the simulations to other Finite-Element Polycrystal Plasticity programs, or when testing other algorithms with the FEpX code-base.

Neper that provides the tessellations and meshes for FEpX can create periodicity, but these will be structures of whole grains, not a cubic volume that FEpX needs for its simulations. Without using another software to create a mesh with periodicity, and convert it to a mesh-file FePX can read, periodicity is hard to simulate.

FEpX has a few built in boundary conditions, namely grip, symmetry, and minimal [1], but any other conditions must be set by the user in a simulation.bcs file. This file needs to contain all boundary nodes with conditions, which is not feasible to write by hand. As FepX doesn't contain a software to do this, a user create a program to do this themselves. This is not very user-friendly, at least compared to Abaqus that can create custom boundary conditions with a few clicks.

Another issue may appear if data from some steps in the deformation history are suppressed. When post-processing out data from the simulation with Neper, any suppressed steps are skipped, meaning the list of processed out-files are not in a clear order. For example, if every other step is suppressed, the out-files will jump over steps in their names. FePX is very intertwined with Neper, but Neper is incapable of handling out-data with names that are not increments of 1. If there are 10 out-files with step values from 0 to 20, and the visualization module is told there are 10 steps, it will stop at .step10, even if there are more steps afterwards. If there is no file with the name .step10, though there are 10 out-files, it will stop the process. This issue can be worked around by manually changing the name of all the out-files so they increment correctly, but it is not user-friendly.

Bibliography

[1] Fepx documentation webpage.

[2] *Continuum Scale Simulation of Engineering Materials*. Wiley VCH, 2004.

[3] Paul R. Dawson and Donald E. Boyce. Fepx – finite element polycrystals: Theory, finite element formulation, numerical implementation and illustrative examples, 2015.

[4] F. Di Gioacchino and J. Quinta da Fonseca. Plastic strain mapping with sub-micron resolution using digital image correlation, 2013.

[5] Olaf Engler and Valerie Randle. *Introduction to Texture Analysis: Macrotexture, Microtexture, and Orientation Mapping, Second Edition (2nd ed.)*. CRC Press, 2010.

[6] J. R. Hirsch. Correlation of deformation texture and microstructure. *Materials Science and Technology*, 6(11):1048–1057, 1990.

[7] Eugene Loh. The ideal hpc programming language. `https://queue.acm.org/detail.cfm?id=1820518`, 2010. Accessed: (March 2023).

[8] T. Mánik, H.M. Asadkandi, and B. Holmedal. A robust algorithm for rate-independent crystal plasticity. *Computer Methods in Applied Mechanics and Engineering*, 393:114831, 2022.

[9] Tomáš Mánik and Bjørn Holmedal. Review of the taylor ambiguity and the relationship between rate-independent and rate-dependent full-constraints taylor models. *International Journal of Plasticity*, 55:152–181, 2014.

[10] R. Quey, P.R. Dawson, and F. Barbe. Large-scale 3d random polycrystals for the finite element method: Generation, meshing and remeshing. *Computer Methods in Applied Mechanics and Engineering*, 200(17):1729–1745, 2011.

[11] Romain Quey. Fepx - github discussions: periodic boundary conditions #35. `https://github.com/neperfepx/FEPX/discussions/35#discussioncomment-4931919`, 2023. Accessed: (February 2023).

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

[13] F. Roters, P. Eisenlohr, L. Hantcherli, D.D. Tjahjanto, T.R. Bieler, and D. Raabe. Overview of constitutive laws, kinematics, homogenization and multiscale methods in crystal plasticity finite-element modeling: Theory, experiments, applications. *Acta Materialia*, 58(4):1152–1211, 2010.

[14] Magnus Själander, Magnus Jahre, Gunnar Tufte, and Nico Reissmann. EPIC: An energy-efficient, high-performance GPGPU computing research infrastructure, 2019.

[15] Michael Smith. *ABAQUS/Standard User's Manual, Version 6.9.* Dassault Systèmes Simulia Corp, United States, 2009.

[16] Andy B. Yoo, Morris A. Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, pages 44–60, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

Appendix A

A.1 Boundary Conditions File Script

```
using System;

namespace boundary_conditions_writer
{
    class Program
    {
        static void Main(string[] args)
        {
            IBoundaryConditionsHandler boundaryConditionsHandler = new
                RollingBoundaryConditionsHandler();
            string meshName = args.Length > 0 ? args[0] : "simulation";
            MeshReader.ReadNodes(boundaryConditionsHandler, meshName);
            BoundaryConditionsWriter.WriteBoundaryConditionsFile(boundaryConditionsHandler.Bo
            Console.Write("Finished!");
        }
    }
}
```

```
namespace boundary_conditions_writer
{
```

```
public static class MeshReader
{
    public static void ReadNodes(IBoundaryConditionsHandler
        boundaryConditionsWriter, string meshName)
    {
        string path = Environment.CurrentDirectory + $"\\{meshName}.msh";

        bool hasReachedNodesNumber = false;
        bool hasReachedNodes = false;
        int totalNodesNumber = 0;
        int i = 0;
        foreach (string line in File.ReadLines(path))
        {
            if (line.Contains("Nodes"))
            {
                if (!hasReachedNodes)
                {
                    hasReachedNodesNumber = true;
                    continue;
                }
                else
                    break;
            }
            if (hasReachedNodesNumber)
            {
                totalNodesNumber = Int32.Parse(line);
                hasReachedNodesNumber = false;
                hasReachedNodes = true;
                continue;
            }
            if (!hasReachedNodes)
                continue;

            string[] args = line.Split(' ');
            if (args.Length != 4)
            {
                Console.Write("Error: Line does not have expected number of
                    arguments!");
                continue;
            }
            boundaryConditionsWriter.CheckNode(ReadNode(args[0], args[1],
                args[2], args[3]));
            i++;
        }

        if (i != totalNodesNumber)
            Console.Write($"Error: Number of read nodes: {i} not equal
                total number of nodes: {totalNodesNumber}!");
    }

    public static Node ReadNode(string idString, string xString, string
        yString, string zString)
    {
        int id = Int32.Parse(idString);
        float x = float.Parse(xString);
        float y = float.Parse(yString);
        float z = float.Parse(zString);

        return new Node(id, x, y ,z);
    }
  }
}
```

```
namespace boundary_conditions_writer
{
    public class Node
    {
        public int ID;
        public float X;
        public float Y;
        public float Z;

        public Node(int id, float x, float y, float z)
        {
            ID = id;
            X = x;
            Y = y;
            Z = z;
        }

        public bool IsMinX()
            => X < 0.001f;

        public bool IsMaxX()
            => X > 0.999f;

        public bool IsMinY()
            => Y < 0.001f;

        public bool IsMaxY()
            => Y > 0.999f;

        public bool IsMinZ()
            => Z < 0.001f;

        public bool IsMaxZ()
            => Z > 0.999f;
    }
}
```

```
namespace boundary_conditions_writer
{
    public interface IBoundaryConditionsHandler
    {
        List<string> BoundaryConditions { get; }
        void CheckNode(Node node);
    }
}
```

```
namespace boundary_conditions_writer
{
    public class RollingBoundaryConditionsHandler : IBoundaryConditionsHandler
    {
        public List<string> BoundaryConditions { get; private set; }

        public RollingBoundaryConditionsHandler()
        {
            BoundaryConditions = new List<string>();
        }

        public void CheckNode(Node node)
        {
            int i = 0;
            if (node.IsMinX())
                i++;
            else if (node.IsMaxX())
                i++;
            if (node.IsMinY())
            {
                BoundaryConditions.Add($"{node.ID} y 0");
                i++;
            }
            else if (node.IsMaxY())
            {
                BoundaryConditions.Add($"{node.ID} y 0");
                i++;
            }
            if (node.IsMinZ())
            {
                BoundaryConditions.Add($"{node.ID} z 1e-2");
                i++;
            }
            else if (node.IsMaxZ())
            {
                BoundaryConditions.Add($"{node.ID} z 0");
                i++;
            }

            if (i == 3)
            {
                Console.Write($"Corner node found! ID: {node.ID}\n");
                if (node.IsMinX() && node.IsMinY() && node.IsMinZ())
                    BoundaryConditions.Add($"{node.ID} x 0");
            }
        }
    }
}
```

```
namespace boundary_conditions_writer
{
    public static class BoundaryConditionsWriter
    {
        public static void WriteBoundaryConditionsFile(List<string>
            boundaryConditions)
        {
            string path = Directory.GetCurrentDirectory() +
                $"\\simulation.bcs";
            using (StreamWriter writer = new StreamWriter(path))
            {
                foreach (string line in boundaryConditions)
                    writer.WriteLine(line);
            }
        }
    }
}
```

A.2 Neper and imagemagic Shell Script

The following shell-script was used to get images from the result folder after a completed FEpX simulation. The Neper commands make images and scales into separate images, and the imagemagic commands glue them together. The script takes two inputs; the name of the images, and how many steps the simulation used. These are the same amount of steps as are set in the FEpX simulation.config file under deformation history.

```
neper -V results -step $2 -datanodecoo coo -dataelt1drad 0.004
   -dataelt3dedgerad 0.0015 -dataelt3dcol strain-pl-eq -dataelt3dcolscheme
   jet -showelt1d all -dataeltscale
   0.0:0.1:0.2:0.3:0.4:0.5:0.6:0.7:0.8:0.9:1.0 -dataeltscaletitle "Equivalent
   Plastic Strain" -cameraangle 25 -imagesize 1600:1000 -print
   $1_eqplstrain_front
convert $1_eqplstrain_front.png $1_eqplstrain_front-scale3d.png -gravity East
   -composite $1_eqplstrain_front_wscale.png
neper -V results -step $2 -datanodecoo coo -dataelt1drad 0.004
   -dataelt3dedgerad 0.0015 -dataelt3dcol strain-pl-eq -dataelt3dcolscheme
   jet -showelt1d all -dataeltscale
   0.0:0.1:0.2:0.3:0.4:0.5:0.6:0.7:0.8:0.9:1.0 -dataeltscaletitle "Equivalent
   Plastic Strain" -cameraangle 25 -cameracoo
   x-length*vx:y-length*vy:z+length*vz -imagesize 1600:1000 -print
   $1_eqplstrain_back
convert $1_eqplstrain_back.png $1_eqplstrain_back-scale3d.png -gravity East
   -composite $1_eqplstrain_back_wscale.png
neper -V results -step $2 -datanodecoo coo -dataelt1drad 0.004
   -dataelt3dedgerad 0.0015 -dataelt3dcol stress33 -dataelt3dcolscheme jet
   -dataeltscaletitle "Stress 33 (MPa)" -dataeltscale 0:1000 -showelt1d all
   -cameraangle 25 -imagesize 1600:1000 -print $1_s33deform_front
convert $1_s33deform_front.png $1_s33deform_front-scale3d.png -gravity East
   -composite $1_s33deform_front_wscale.png
neper -V results -step $2 -datanodecoo coo -dataelt1drad 0.004
   -dataelt3dedgerad 0.0015 -dataelt3dcol stress33 -dataelt3dcolscheme jet
```

```
    -dataeltscaletitle "Stress 33 (MPa)" -dataeltscale 0:1000 -showelt1d all
    -cameraangle 25 -cameracoo x-length*vx:y-length*vy:z+length*vz -imagesize
    1600:1000 -print $1_s33deform_back
convert $1_s33deform_back.png $1_s33deform_back-scale3d.png -gravity East
    -composite $1_s33deform_back_wscale.png
```