Technische Universität München
Fakultät für Maschinenwesen
Lehrstuhl für Werkstoffkunde und Werkstoffmechanik
Prof. Dr. mont. habil. E. Werner
Christian Doppler Laboratorium für Werkstoffmechanik von Hochleistungslegierungen
Dr.-Ing. C. Krempaszky

Max-Planck-Institut für Eisenforschung GmbH
Abteilung Mikrostrukturphysik und Umformtechnik
Prof. Dr.-Ing. habil. D. Raabe

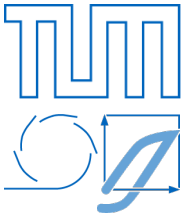
A spectral method using fast Fourier transform to solve elastoviscoplastic mechanical boundary value problems

Martin Diehl

Diploma thesis

Written by:	Martin Diehl
Matriculation No.:	2819862
Supervisor:	Prof. Dr. mont. habil. Ewald Werner
	Dr.-Ing. Cornelia Schwarz
	Dr.-Ing. Christian Krempaszky
Start date:	15.05.2010
Submission date:	15.11.2010

Involved organizations



Technische Universität München
Fakultät für Maschinenwesen
Lehrstuhl für Werkstoffkunde und Werkstoffmechanik
Boltzmannstraße 15
D-85748 Garching

Christian Doppler Laboratorium für Werkstoffmechanik von Hochleistungslegierungen
Boltzmannstraße 15
D-85748 Garching

Max-Planck-Institut für Eisenforschung GmbH
Abteilung Mikrostrukturphysik und Umformtechnik
Max-Planck-Straße 1
D-40237 Düsseldorf

Erklärung

Hiermit erkläre ich an Eides statt, dass ich diese Diplomarbeit zum Thema

A spectral method using fast Fourier transform to solve elastoviscoplastic mechanical boundary value problems

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Garching, den 15.11.2010

Martin Diehl

Martin Diehl
Apfelkammerstraße 13-15
81241 München
e-Mail: martin.diehl@mytum.de

Acknowledgements

This diploma thesis was written during my stay at the department for *Mikrostrukturphysik und Umformtechnik* of the *Max-Planck-Institut für Eisenforschung* GmbH (*MPIE*). It would not have been possible to carry out my work without the academic and technical support of the *MPIE* staff.

I would like to express my special appreciation to Philip Eisenlohr. Many hours of discussion and his hints in programming were an invaluable help regarding all aspects of the work. I am also grateful for his support in writing the thesis and his excellent suggestions for formatting and expression.

I would like to thank my principal supervisor Cornelia Schwarz from the *Lehrstuhl für Werkstoffkunde und Werkstoffmechanik*, *TU München* for her good advice. Her comments on the intermediate versions of my work definitely helped me to improve this thesis.

I am most grateful to Ricardo Lebensohn for providing me with his implementation of the spectral method. His support in integrating the spectral method into the existing *MPIE* routines was extremely useful.

I would like to thank Franz Roters for answering my various questions concerning all aspects of the material subroutines and their underlying physics.

The interesting talks with Christian Krempaszky (*Christian Doppler Laboratorium für Werkstoffmechanik von Hochleistungslegierungen*, *TU München*) greatly helped to understand the mathematics of the implemented spectral method.

I am grateful for Lucy Duggan's comments and suggestions regarding the phrasing. She also picked up a large amount of spelling errors and made helpful suggestions on how to improve the readability of the thesis.

For any errors or inadequacies that may remain in this work, of course, the responsibility is entirely my own.

Nomenclature

Latin Letters

Symbol	*Description*	*Unit*
a_{tol}	abort criterion	-
$\mathcal{B}$	body	
$\boldsymbol{b}$	BURGER's vector	m
$\boldsymbol{B}$	left CAUCHY–GREEN deformation tensor	-
$\boldsymbol{C}$	right CAUCHY–GREEN deformation tensor	-
$\mathbb{C}$	stiffness tensor	Pa
E	YOUNG's modulus	Pa
$\boldsymbol{E}_0$	GREEN–LAGRANGE strain tensor	-
$\boldsymbol{E}_{\mathrm{t}}$	EULER–ALMANSI strain tensor	-
$\mathcal{F}$	FOURIER transform	
$\boldsymbol{F}$	deformation gradient	-
G	GREEN's function	
$\mathcal{H}$	HEAVISIDE function	
$\boldsymbol{H}_0$	displacement gradient	-
$\boldsymbol{H}_{\mathrm{t}}$	inverse displacement gradient	-
$\boldsymbol{I}$	identity, unit matrix	
J	JACOBIan determinant of the deformation gradient	-
J_2	2nd invariant of the deviatoric part of the CAUCHY stress	Pa^2
k	angular frequency	1/s
l	side length	m
$\boldsymbol{L}$	velocity gradient	m/s
M	TAYLOR factor	
N	number of sampling points	
$\boldsymbol{P}$	1st PIOLA–KIRCHHOFF stress tensor	Pa
$\boldsymbol{r}$	microstructure parametrization (slip resistance r for the used models)	
$\boldsymbol{R}$	rotation tensor	-
$\mathbb{S}$	compliance tensor	Pa^{-1}
$\boldsymbol{S}$	2nd PIOLA–KIRCHHOFF stress tensor	Pa
$\boldsymbol{s}$	line direction	m
$\boldsymbol{u}$	displacement	m

$\boldsymbol{U}$	right stretch tensor	-
$\boldsymbol{V}$	left stretch tensor	-
$\boldsymbol{x}$	coordinates in reference configuration	m
$\boldsymbol{y}$	coordinates in current configuration	m

Greek Letters

Symbol	*Description*	*Unit*
δ	unit impulse function	
δ_{im}	KRONECKER delta	
Δ	deviation	
$\boldsymbol{\varepsilon}$	CAUCHY strain tensor	-
γ	shear strain	-
$\mathbb{\Gamma}$	Γ-operator for GREEN's function	
κ	frequency	Hz
ν	POISSON ratio	-
$\boldsymbol{\sigma}$	CAUCHY stress tensor, infinitesimal stress tensor	Pa
$\boldsymbol{\tau}$	polarization field	Pa
τ	shear stress	Pa
$\boldsymbol{\omega}$	rotation	-

Superscripts

Symbol	*Description*
$'$	deviatoric part of a tensor
$\cdot$	derivative with respect to time
~	fluctuating part of a quantity
–	average quantity, negative quantity for MILLER indices
^	quantity in FOURIER space
α	slip system
β	twin system
m	iteration counter

Subscripts

Symbol	*Description*
0	quantity in reference configuration
e	elastic part
p	plastic part
ref	reference value
t	quantity in current configuration
vM	VON MISES equivalent of a tensorial quantity

Contents

1 Introduction

Higher requirements on the mechanical properties of construction materials are needed in applications in all kinds of engineering. The growing demands led to an enormous variety of alloying concepts to produce metallic materials with the desired specifications. The outstanding performance of recently developed alloys is based on interactions on the scale of the microstructure. Most of the effects are related to the interaction of different phases, phase changes, dislocation movement and twinning.

The mechanical properties of materials are not only derived from experiments, but also examined using computer assisted simulations. The simulation of mechanical behavior is often done on a volume element (VE) with a representative structure for the material. That means, a representative volume element (RVE) predicts the behavior of a body made out of the material. Usually, periodic boundary conditions (BCs) are enforced on each side of the RVE. They expand the volume under consideration to an infinite body by repeating it infinitely [10]. An exemplary VE consisting of 50 grains is shown in fig. 1.1.

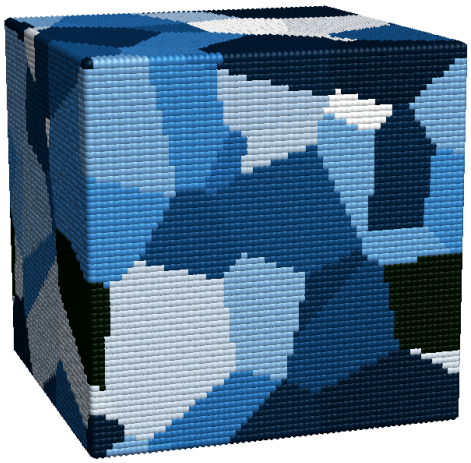


Figure 1.1: Volume element consisting of 50 periodically repeating grains

Since interactions in the microstructure are responsible for the performance of the material, the knowledge of the underlying physics is becomming more and more important for the simulation of the mechanical behavior. For a single-phase alloy, even a simple constitutive law will quite accurately predict results. To simulate the behavior of a complex material, the underlying effects such as interaction of different phases, phase changes, dislocation mobility, twinning etc. have to be considered in order to produce applicable results [29].

One of the crucial points in using simulations is the time spent on calculations. For a complex microstructure a very detailed description is needed. Even fast computers take a long time to complete the calculation. The most commonly used technique for solving partial differential equations (PDEs) describing the mechanical behavior is the "Finite Element Method"(FEM) [2, 29, 31, 35]. For VEs with periodic BCs the so-called spectral methods are a fast alternative to the FEM [4, 32].

In this thesis, the implementation of a spectral method using the fast FOURIER transform (FFT) around an existing framework for crystal plasticity FEM (CPFEM) is described. The spectral method serves as an alternative for the FEM-based solvers to the calculation of VEs with periodic

BCs. The framework and the spectral method are written in *Fortran*.

The general idea of using spectral methods for mechanical boundary value problems was introduced by H. Moulinec and P. Suquet in 1998 [18, 19]. Further development was done by S. Neumann, K. P. Herrmann and W. H. Müller [6, 22] and R. A. Lebensohn [14]. Extensions capable of handling large strain formulations were presented by N. Lahellec, J. C. Michel, H. Moulinec, and P. Suquet in 2001 [17].

An implementation of the algorithm was used by R. A. Lebensohn and A. Prakash to evaluate the performance of the method. The promising results of their study show significantly decreased computation time compared to a standard FEM. While the results for almost homogeneous deformations were close to the results achieved by FEM, the shape update algorithm presented there is not suitable for locally extremely distorted morphologies [25].

This thesis presents an implementation is shown that overcomes the limits of the former method concerning the shape update. Moreover, it is integrated in a flexible framework to handle arbitrary material models. To speed up the calculations, particular care was taken to implement a fast algorithm. Due to the fast computations, the VE can have a size which is large enough to be considered as representative for the material. Thus, the simulation can serve as a "virtual laboratory" to derive the parameters describing the mechanical behavior of a material.

In this work, first a mathematical framework is introduced in chapter 2 to describe deformations of bodies under load. In chapter 3 the deformation mechanisms of crystal materials are explained as they are important to derive suitable constitutive laws. A selection of constitutive models are also presented in this chapter. In chapters 4 and 5 the mathematical background of the spectral method, Green's function method and Fourier transform are briefly recapitulated. The basic concept of spectral methods and their application to mechanical boundary value problems is outlined in chapter 6. In chapter 7 the details of the implementation are described. Results of some completed simulations are given in chapter 8. The thesis ends with a summary and the conclusions drawn from the work. The last chapter, chapter 9, gives an outlook on further work.

2 Continuum mechanics

The theory of continuum mechanics describes the global mechanical behavior of solids and fluids. In continuum mechanics, a hypothetic continuous medium is used to describe the macroscopic behavior of an object. According to the assumption of a continuous medium, the material of the object completely fills the space it occupies. It is not possible to model empty spaces, cracks or discontinuities inside the material. Therefore, the atomic structure of materials cannot be described. The concept of an continuous medium allows us to describe the behavior of the material with continuous mathematical functions. In this work, it is used to describe the behavior of solid materials under external forces and applied displacements.

In this chapter, at first the different configurations of a body under load are shown (section 2.1). From the configurations, strain (section 2.2) and stress (section 2.5) measures are derived. This chapter is mainly based on [29, 34].

2.1 Configurations

A continuous body $\mathcal{B}$ can be described as a composition of an infinite number of material points or particles x, with $x \in \mathcal{B}$. The body in the undeformed configuration occupies the region $\mathcal{B}_0$. This configuration is also called the reference configuration. The reference configuration does not depend on time. In the time-dependent deformed state, the body occupies the region $\mathcal{B}_t$. This configuration is called the deformed configuration or the current configuration. The location of the material points in the undeformed state is given by the vector $\boldsymbol{x}$ and in a deformed state by the vector $\boldsymbol{y}$. Two example configurations are shown in fig. 2.1. In each configuration a different base with corresponding unit vectors exists. In this work, the same coordinate system with the same unit vectors

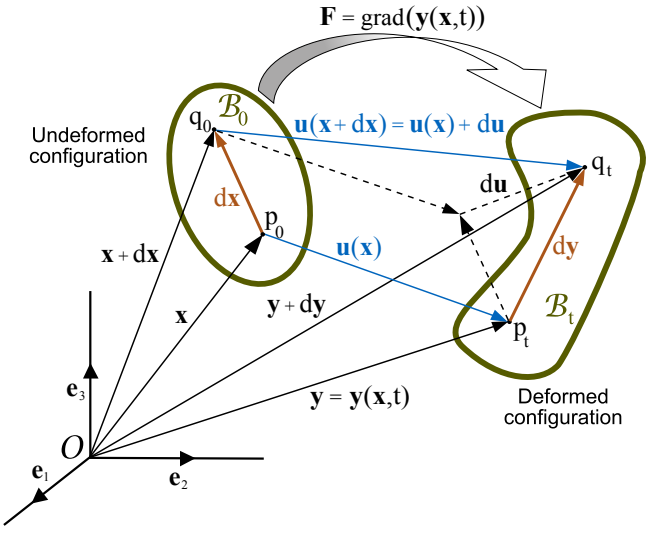


Figure 2.1: Continuum body shown in undeformed and deformed configurations[1]

[1] File taken from `http://commons.wikimedia.org/wiki/File:Continuum_body_deformation.svg`, accessed 14th November 2010. The copyright information can be found in appendix C.

is used in both configurations. This allows us to write down the tensors without explicitly notating the unit vectors.

A deformation can be described in both configurations. In the material description, each particle belongs to its current spatial location. This is also called LAGRANGian description. The spatial description is also called EULERian description. In EULERian description, the location belongs to the particle. Loosely speaking, the LAGRANGian description answers the question "At which location is the particle?", while the EULERian description answers the question "Which particle is at the location?". As usual in structural mechanics (and in contrast to fluid mechanics), in this work a LAGRANGian description is used.

2.2 Deformation and strain measures

The displacement $\boldsymbol{u}$ of a material point is the difference vector from a point in reference configuration to the deformed configuration:

$$\boldsymbol{u}(\boldsymbol{x}, t) = \boldsymbol{y}(\boldsymbol{x}, t) - \boldsymbol{x} \tag{2.1}$$

$\boldsymbol{u}$ must be a continuous function. For a given time—that is a given deformation state—the equation reads $\boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{y}(\boldsymbol{x}) - \boldsymbol{x}$.

A line segment $\mathrm{d}\boldsymbol{x}$ in an infinitesimal neighborhood of a material point $\boldsymbol{x}$ in the reference configuration is transformed into the current configuration by:

$$\boldsymbol{y}(\boldsymbol{x}) + \mathrm{d}\boldsymbol{y} = \boldsymbol{y}(\boldsymbol{x}) + \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \cdot \mathrm{d}\boldsymbol{x} + \mathcal{O}(\mathrm{d}\boldsymbol{x}^2) \tag{2.2}$$

By neglecting terms of higher order, $\mathrm{d}\boldsymbol{y}$ can be written as:

$$\mathrm{d}\boldsymbol{y} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} \cdot \mathrm{d}\boldsymbol{x} = \boldsymbol{F} \cdot \mathrm{d}\boldsymbol{x} \tag{2.3}$$

where $\boldsymbol{F} := \partial \boldsymbol{y} / \partial \boldsymbol{x}$ is a 2nd order tensor called deformation gradient. It is also denoted $\mathrm{grad}(\boldsymbol{y})$. The relations between vectors and tensors in the different configurations are graphically shown in fig. 2.1. The deformation gradient maps the vector $\mathrm{d}\boldsymbol{x}$ at $\boldsymbol{x}$ in the reference configuration to the vector $\mathrm{d}\boldsymbol{y}$ at $\boldsymbol{y}$ in the current configuration. The deformation tensor has one base in the reference configuration and one in the current configuration. It is therefore called a 2-point tensor.

The inverse of the deformation tensor $\boldsymbol{F}^{-1}$ maps an element from the current configuration to the reference configuration. It is sometimes called the spatial deformation gradient, while $\boldsymbol{F}$ is called the material deformation gradient. The spatial line segment $\mathrm{d}\boldsymbol{y}$ is called the "push forward" of the material line segment $\mathrm{d}\boldsymbol{x}$, which in turn can be called the "pull back" (performed by $\boldsymbol{F}^{-1}$) of $\mathrm{d}\boldsymbol{y}$.

Inserting eq. (2.1) into eq. (2.3) results in the deformation gradient written as:

$$\boldsymbol{F} = \frac{\partial (\boldsymbol{x} + \boldsymbol{u})}{\partial \boldsymbol{x}} \tag{2.4}$$

$$= \boldsymbol{I} + \frac{\partial \boldsymbol{u}}{\partial \boldsymbol{x}} \tag{2.5}$$

$\boldsymbol{H}_0 := \partial \boldsymbol{u}/\partial \boldsymbol{x}$ is called the displacement gradient. The tensor $\boldsymbol{I}$ is the identity or unit matrix.

Displacement gradient and deformation gradient are a means of describing the deformation of a body. In the same way as $\boldsymbol{F}$ is called the material deformation gradient, $\boldsymbol{H}_0$ is called the material displacement gradient. The spatial displacement gradient is defined as $\boldsymbol{I} - \boldsymbol{F}^{-1}$ and denoted as $\boldsymbol{H}_\mathrm{t}$.

Deformation gradient, displacement gradient and their respective inverse are 2-point tensors. It is also possible to describe the deformation in the reference configuration only by:

$$\mathrm{d}\boldsymbol{y} \cdot \mathrm{d}\boldsymbol{y} = (\boldsymbol{F} \cdot \mathrm{d}\boldsymbol{x}) \cdot (\boldsymbol{F} \cdot \mathrm{d}\boldsymbol{x}) \tag{2.6}$$

$$= \mathrm{d}\boldsymbol{x} \cdot (\boldsymbol{F}^\mathrm{T} \cdot \boldsymbol{F}) \cdot \mathrm{d}\boldsymbol{x} \tag{2.7}$$

$\boldsymbol{C} := \boldsymbol{F}^\mathrm{T} \cdot \boldsymbol{F}$ is called the right Cauchy–Green deformation tensor. It is a symmetric tensor completely in the material (reference) configuration.

The change of length (the strain) under a deformation can be expressed as:

$$\mathrm{d}\boldsymbol{y} \cdot \mathrm{d}\boldsymbol{y} - \mathrm{d}\boldsymbol{x} \cdot \mathrm{d}\boldsymbol{x} = \mathrm{d}\boldsymbol{x} \cdot \boldsymbol{C} \cdot \mathrm{d}\boldsymbol{x} - \mathrm{d}\boldsymbol{x} \cdot \mathrm{d}\boldsymbol{x} \tag{2.8}$$

$$= \mathrm{d}\boldsymbol{x} \cdot (\boldsymbol{C} - \boldsymbol{I}) \cdot \mathrm{d}\boldsymbol{x} \tag{2.9}$$

$$= \mathrm{d}\boldsymbol{x} \cdot (2\,\boldsymbol{E}_0) \cdot \mathrm{d}\boldsymbol{x} \tag{2.10}$$

$\boldsymbol{E}_0 := \frac{1}{2}(\boldsymbol{C} - \boldsymbol{I}) = \frac{1}{2}(\boldsymbol{F}^\mathrm{T} \cdot \boldsymbol{F} - \boldsymbol{I})$ is called the Green–Lagrange strain tensor. It depends only on the right Cauchy–Green deformation tensor, and is therefore also completely in the reference configuration.

A similar transform as in eq. (2.6) can express the deformation in the current configuration:

$$\mathrm{d}\boldsymbol{x} \cdot \mathrm{d}\boldsymbol{x} = (\boldsymbol{F}^{-1} \cdot \mathrm{d}\boldsymbol{y}) \cdot (\boldsymbol{F}^{-1} \cdot \mathrm{d}\boldsymbol{y}) \tag{2.11}$$

$$= \mathrm{d}\boldsymbol{y} \cdot (\boldsymbol{F}^{-\mathrm{T}} \cdot \boldsymbol{F}^{-1}) \cdot \mathrm{d}\boldsymbol{y} \tag{2.12}$$

$\boldsymbol{B}^{-1} := \boldsymbol{F}^{-\mathrm{T}} \cdot \boldsymbol{F}^{-1}$ leads to $\boldsymbol{B} = \boldsymbol{F} \cdot \boldsymbol{F}^\mathrm{T}$. The tensor $\boldsymbol{B}$ is called the left Cauchy–Green deformation tensor. It is a symmetric tensor completely in the spatial (current) configuration.

The change of length under a deformation can be expressed as:

$$\mathrm{d}\boldsymbol{y} \cdot \mathrm{d}\boldsymbol{y} - \mathrm{d}\boldsymbol{x} \cdot \mathrm{d}\boldsymbol{x} = \mathrm{d}\boldsymbol{y} \cdot \mathrm{d}\boldsymbol{y} - \mathrm{d}\boldsymbol{y} \cdot \boldsymbol{B}^{-1} \cdot \mathrm{d}\boldsymbol{y} \tag{2.13}$$

$$= \mathrm{d}\boldsymbol{y} \cdot (\boldsymbol{I} - \boldsymbol{B}^{-1}) \cdot \mathrm{d}\boldsymbol{y} \tag{2.14}$$

$$= \mathrm{d}\boldsymbol{y} \cdot (2\,\boldsymbol{E}_\mathrm{t}) \cdot \mathrm{d}\boldsymbol{y} \tag{2.15}$$

$\boldsymbol{E}_\mathrm{t} := \frac{1}{2}(\boldsymbol{I} - \boldsymbol{B}^{-1}) = \frac{1}{2}(\boldsymbol{I} - \boldsymbol{F}^{-\mathrm{T}} \cdot \boldsymbol{F}^{-1})$ is called the Euler–Almansi strain tensor. As a product of the left Cauchy–Green deformation tensor it is completely in the current configuration.

The push forward and pull back are also defined for the deformation measures. The push forward of the Green–Lagrange stretch tensor is the Euler–Almansi stretch tensor, while the pull

back performs the inverse operation:

$$\boldsymbol{E}_{\mathrm{t}} = \boldsymbol{F}^{-\mathrm{T}} \cdot \boldsymbol{E}_0 \cdot \boldsymbol{F}^{-1} \tag{2.16}$$

$$\boldsymbol{E}_0 = \boldsymbol{F}^{\mathrm{T}} \cdot \boldsymbol{E}_{\mathrm{t}} \cdot \boldsymbol{F} \tag{2.17}$$

For small strains, i.e. $\boldsymbol{y} - \boldsymbol{x} \approx \boldsymbol{0}$, the linearization of the Euler–Almansi strain tensor and the Green–Lagrange strain results in the same strain tensor. It is called the Cauchy strain tensor $\boldsymbol{\varepsilon}$. It reads as:

$$\varepsilon_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i}) \approx E_{0,ij} \approx E_{\mathrm{t},ij} \tag{2.18}$$

when using index notation and Einstein convention[2]. In vector notation it reads as:

$$\boldsymbol{\varepsilon} = \frac{1}{2}(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^{\mathrm{T}}) \tag{2.19}$$

With the nabla-operator ∇ and $\boldsymbol{\omega} := \frac{1}{2}(\nabla \boldsymbol{u} - (\nabla \boldsymbol{u})^{\mathrm{T}})$ denoted the rotation tensor, the displacement gradient can be written in the infinitesimal strain framework as:

$$\nabla \boldsymbol{u} = \boldsymbol{\varepsilon} + \boldsymbol{\omega}. \tag{2.20}$$

For the one-dimensional case without lateral contraction, the deformation can be described by two variables: The length in the current configuration l_{t} and the length in the reference configuration l_0. By defining the stretch ratio as $\lambda := l_{\mathrm{t}}/l_0$ the corresponding strain measures and the limits for infinite tension and infinite compression read as:

Table 2.1: Definition of strain measures and behavior for tension and compression

strain measure	definition	compression	tension
Green–Lagrange strain	$\boldsymbol{E}_{0,1\mathrm{dim}} = \frac{1}{2}(\lambda^2 - 1)$	$\lim_{\lambda \to 0} \boldsymbol{E}_{0,1\mathrm{dim}} = -\frac{1}{2}$	$\lim_{\lambda \to \infty} \boldsymbol{E}_{0,1\mathrm{dim}} = \infty$
Euler–Almansi strain	$\boldsymbol{E}_{\mathrm{t},1\mathrm{dim}} = \frac{1}{2}(1 - \frac{1}{\lambda^2})$	$\lim_{\lambda \to 0} \boldsymbol{E}_{\mathrm{t},1\mathrm{dim}} = -\infty$	$\lim_{\lambda \to \infty} \boldsymbol{E}_{\mathrm{t},1\mathrm{dim}} = \frac{1}{2}$
Cauchy strain	$\varepsilon_{1\mathrm{dim}} = \lambda - 1$	$\lim_{\lambda \to 0} \varepsilon_{1\mathrm{dim}} = -1$	$\lim_{\lambda \to \infty} \varepsilon_{1\mathrm{dim}} = \infty$

From tab. 2.1 it can be seen that the measures are not symmetrical and reach different limits for infinite tension and infinite compression. A measure that overcomes these inconsistencies is the logarithmic strain $\varepsilon_{\log} = \ln(\lambda)$. Tension or compression applied at the same rate (section 2.4) for a given time will result in a logarithmic strain which differs only in the algebraic sign. The different strain measures for the one-dimensional case are shown in fig. 2.2.

For the spatial and material case, different strain measures with power n of λ can be derived. Both measures are based on the formula $1/\alpha(1 - \lambda^{-\alpha})$. For the material measures, the exponent α has a negative sign, for the spatial measures a positive one. From tab. 2.1 it can be seen that λ contributes with its second power to the Green–Lagrange and the Euler–Almansi strain. The strain measure of order 0 is the logarithmic strain [34].

[2] According to Einstein notation or Einstein summation, it is implicitly summed over an index variable that appears twice in a product.

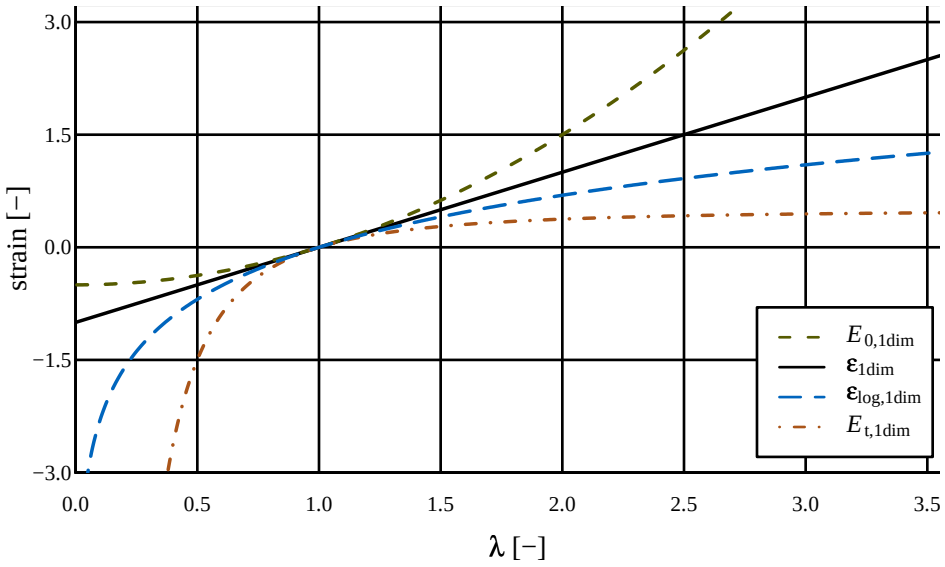


Figure 2.2: Behavior of different strain measures for tension and compression

In the multidimensional case the calculation of the different strain measures must be conducted in the principal coordinate system, i.e., in the basis of eigenvectors.

2.3 Polar decomposition

Each tensor can be decomposed into a component of pure stretches and a pure rotation. For an invertible tensor the decomposition is unique and can be expressed by two forms. For the deformation gradient, the decomposition reads as:

$$\boldsymbol{F} = \boldsymbol{V} \cdot \boldsymbol{R} = \boldsymbol{R} \cdot \boldsymbol{U} \tag{2.21}$$

where $\boldsymbol{R}$ is the rotation tensor, $\boldsymbol{v}$ is called the left stretch tensor and $\boldsymbol{U}$ the right stretch tensor.

It can be shown that $\boldsymbol{U}^2 = \boldsymbol{C}$ and $\boldsymbol{V}^2 = \boldsymbol{B}$ with $\boldsymbol{B}$ being the left and $\boldsymbol{C}$ the right Cauchy–Green deformation tensor. While $\boldsymbol{F}$ is a 2-point tensor, $\boldsymbol{U}$ is in material configuration and $\boldsymbol{V}$ in spatial configuration only. They have all the same determinant J, called the Jacobian:

$$J = \det(\boldsymbol{F}) = \det(\boldsymbol{U}) = \det(\boldsymbol{V}) \tag{2.22}$$

A pure rotation does not change the shape of the body and should therefore result in zero strain. The polar decomposition can be used to check whether a strain measure is valid. The framework for large deformations, also called "finite strain framework" presented here is able to describe large deformations properly. It can be shown that the small strain formulation (infinitesimal strain formulation) does not fulfill this requirement for large deformations and is therefore not suitable to describe them.

A summary of the polar decomposition of the deformation gradient and the derived deformation tensors is given in tab. 2.3.

2.4 Velocity gradient

For a moving body, the position of the material points vary with time. The material velocity field is defined as:

$$\boldsymbol{v} = \frac{\mathrm{d}\boldsymbol{u}(\boldsymbol{x})}{\mathrm{d}t} = \dot{\boldsymbol{u}} = \dot{\boldsymbol{y}} \tag{2.23}$$

$\dot{\boldsymbol{u}} = \dot{\boldsymbol{y}}$ holds because the points in the reference configuration do not change their position, i.e., $\mathrm{d}\boldsymbol{x}/\mathrm{d}t = \boldsymbol{0}$. The spatial gradient of the velocity field is:

$$\boldsymbol{L} = \frac{\partial \boldsymbol{v}}{\partial \boldsymbol{y}} = \dot{\boldsymbol{F}} \cdot \boldsymbol{F}^{-1} \tag{2.24}$$

$\boldsymbol{L}$ is called the velocity gradient. Loading a body with a constant velocity gradient will result in the same rate of deformation independently of the shape in the current configuration.

2.5 Stress measures

Stress is defined as force per area. As introduced in section 2.1, in nonlinear continuum mechanics a distinction has to be made between the reference and the current configuration. As a result, different stress measures exist, depending on the configuration in which force and area are defined. In the current configuration, a force $\Delta\boldsymbol{r}_\mathrm{t}$ on an area Δa_t with normal vector $\boldsymbol{n}_\mathrm{t}$ results in:

$$\boldsymbol{t}_\mathrm{t}(\boldsymbol{n}_\mathrm{t}) = \lim_{\Delta a_\mathrm{t} \to 0} \frac{\Delta \boldsymbol{r}_\mathrm{t}}{\Delta a_\mathrm{t}} \tag{2.25}$$

where $\boldsymbol{t}_\mathrm{t}$ is called the vector of surface traction or CAUCHY traction. The CAUCHY stress tensor or "true stress tensor" $\boldsymbol{\sigma}$ is defined as:

$$\boldsymbol{t}_\mathrm{t}(\boldsymbol{y}, t, \boldsymbol{n}_\mathrm{t}) =: \boldsymbol{\sigma}(\boldsymbol{y}, t) \cdot \boldsymbol{n}_\mathrm{t} \tag{2.26}$$

The CAUCHY stress is a 2nd order tensor in spatial coordinates.

The traction $\boldsymbol{t}_\mathrm{t}$ is defined on the the current area ($\Delta a_\mathrm{t} \to 0$). Scaling it to the area in reference configuration Δa_0 results in the pseudo traction vector $\boldsymbol{t}_{0,\mathrm{t}}$. It is also called nominal or 1st PIOLA–KIRCHHOFF traction vector. It can be determined by looking at an infinitesimal force $\mathrm{d}\boldsymbol{r}_\mathrm{t}$:

$$\mathrm{d}\boldsymbol{r}_\mathrm{t} = \boldsymbol{t}_\mathrm{t}\,\mathrm{d}a_\mathrm{t} = \boldsymbol{t}_{0,\mathrm{t}}\,\mathrm{d}a_0 \tag{2.27}$$

The vector notation of the areas in the reference configuration, or the current configuration is $\mathrm{d}\boldsymbol{a}_\mathrm{t} = \boldsymbol{n}_\mathrm{t}\,\mathrm{d}a_\mathrm{t}$ and $\mathrm{d}\boldsymbol{a}_0 = \boldsymbol{n}_0\,\mathrm{d}a_0$. According to [34] the deformation of an area can be expressed by $\mathrm{d}\boldsymbol{a}_\mathrm{t} = J\,\boldsymbol{F}^{-\mathrm{T}} \cdot \mathrm{d}\boldsymbol{a}_0$. This allows the transformation of eq. (2.27) and eq. (2.26) into:

$$\boldsymbol{t}_{0,\mathrm{t}} = \left(J\,\boldsymbol{\sigma} \cdot \boldsymbol{F}^{-\mathrm{T}}\right) \cdot \boldsymbol{n}_0 \tag{2.28}$$

The product of the two 2nd order tensors and the JACOBIan is called the 1st PIOLA–KIRCHHOFF stress $\boldsymbol{P}$:

$$\boldsymbol{P} = J\,\boldsymbol{\sigma} \cdot \boldsymbol{F}^{-\mathrm{T}} \tag{2.29}$$

Like $\boldsymbol{F}$ it is a 2-point tensor (one base in spatial base and one in material base). It is a non-symmetric tensor of 2nd order. It relates the force in the deformed configuration to an oriented area vector in the reference configuration.

To get a stress measure in the current configuration only, the resulting force $\mathrm{d}\boldsymbol{r}_\mathrm{t}$ in reference configuration can be written as:

$$\mathrm{d}\boldsymbol{r}_0 = \boldsymbol{F}^{-1} \cdot \mathrm{d}\boldsymbol{r}_\mathrm{t} = \boldsymbol{F}^{-1} \cdot \boldsymbol{t}_{\mathrm{t},0}\, \mathrm{d}a_0 \tag{2.30}$$

$\boldsymbol{t}_0 := \boldsymbol{F}^{-1} \cdot \boldsymbol{t}_{t,0}$ is called the 2nd PIOLA–KIRCHHOFF traction vector. It can be shown that $\boldsymbol{t}_0 = (J\,\boldsymbol{F}^{-1}{\cdot}\boldsymbol{\sigma}{\cdot}\boldsymbol{F}^{-\mathrm{T}}){\cdot}\boldsymbol{n}_0$. The tensor $\boldsymbol{S} := (J\,\boldsymbol{F}^{-1}{\cdot}\boldsymbol{\sigma}{\cdot}\boldsymbol{F}^{-\mathrm{T}})$ is called the 2nd PIOLA–KIRCHHOFF stress tensor. It is a pure material, symmetric tensor of 2nd order. The 2nd PIOLA–KIRCHHOFF stress tensor is the pull back of the CAUCHY stress tensor.
The different stress measures and the corresponding strain measures are summarized in tab. 2.2.

All stress measures are tensors of 2nd order. In the three-dimensional case they consist of nine components, of which all (for non-symmetric tensors) or six (for symmetric tensors) are independent. Different approaches exist to express the stress state in one variable that can be compared to uniaxial stress, coming from tensile test for example. The most common approach is the VON MISES equivalent stress σ_vM. It is based on the hypothesis that the material state is solely dependent on the change of shape. Or in other words, hydrostatic compression or expansion of the volume is not important for describing the stress state. Mathematically it can be expressed by $\sigma_\mathrm{vM} = \sqrt{3J_2}$, with J_2 being the 2nd invariant of the deviatoric part $\boldsymbol{\sigma}'$ of the stress tensor $\boldsymbol{\sigma}$. Experiments have shown that the assumption fits well for ductile materials [15].

2.6 Constitutive relation

A constitutive equation relates the response of a material to an external load. In continuum mechanics, it usually gives the connection between stress and the resulting deformation. Without a constitutive equation, the equations describing the mechanical behavior of a material cannot be solved.

A wide range of constitutive relations exist for describing the relation between stress and strain in materials. While some of them—called phenomenological laws—are based on measurements only, others try to describe the underlying physics.

Complex constitutive laws need several variables to characterize the material state and its response to the load. Before deriving constitutive laws that are suitable to describe the material response accurately enough to be used in crystal plasticity, a closer look at the mechanical behavior of crystalline structures is needed. In the following chapter some fundamentals about deformation mechanisms in crystalline structures are given. The constitutive models used in this work are derived from this underlying physics. They are also introduced in chapter 3.

Table 2.2: Stress and strain in different configurations

configuration	stress	⇔	deformation	⇔	strain			symmetry
current, spatial	$\boldsymbol{\sigma}$		$\boldsymbol{B}^{-1}$	$\frac{1}{2}(\boldsymbol{I}-\boldsymbol{B}^{-1})=\boldsymbol{E}_\mathrm{t}$	$\boldsymbol{E}_\mathrm{t}$			symmetric
⇕	$J\,\boldsymbol{\sigma}\cdot\boldsymbol{F}^{-T}=\boldsymbol{P}$	constitutive relation	$\boldsymbol{F}^{-T}\cdot\boldsymbol{F}^{-1}=\boldsymbol{B}^{-1}$			$\boldsymbol{E}_\mathrm{t}=\boldsymbol{F}^{-T}\cdot\boldsymbol{E}_0\cdot\boldsymbol{F}^{-1}$	$\boldsymbol{E}_0=\boldsymbol{F}^{T}\cdot\boldsymbol{E}_\mathrm{t}\cdot\boldsymbol{F}$	
2-point	$\boldsymbol{P}$		$\boldsymbol{F}^{-1}$	$\boldsymbol{I}-\boldsymbol{F}^{-1}=\boldsymbol{H}_\mathrm{t}$	$\boldsymbol{H}_\mathrm{t}$			non-symmetric
			$\boldsymbol{F}$	$\boldsymbol{F}-\boldsymbol{I}=\boldsymbol{H}_0$	$\boldsymbol{H}_0$			
⇕	$\boldsymbol{F}^{-1}\cdot\boldsymbol{P}=\boldsymbol{S}$		$\boldsymbol{F}^{T}\cdot\boldsymbol{F}=\boldsymbol{C}$					
reference, material	$\boldsymbol{S}$		$\boldsymbol{C}$	$\frac{1}{2}(\boldsymbol{C}-\boldsymbol{I})=\boldsymbol{E}_0$	$\boldsymbol{E}_0$			symmetric

$\boldsymbol{\sigma}$	Cauchy stress	$\boldsymbol{B}$	left Cauchy–Green deformation tensor	$\boldsymbol{E}_\mathrm{t}$	Euler–Almansi strain tensor
$\boldsymbol{P}$	1st Piola–Kirchhoff stress	$\boldsymbol{F}^{-1}$	inverse deformation gradient	$\boldsymbol{H}_\mathrm{t}$	inverse displacement gradient
$\boldsymbol{S}$	2nd Piola–Kirchhoff stress	$\boldsymbol{F}$	deformation gradient	$\boldsymbol{E}_0$	displacement gradient
		$\boldsymbol{C}$	right Cauchy–Green deformation tensor	$\boldsymbol{E}_0$	Green–Lagrange strain tensor

Table 2.3: Polar decomposition

determinant	deformation/stretch	configuration
$J=\det(\boldsymbol{V})$	$\boldsymbol{V}^2=\boldsymbol{B}$	current, spatial
$J=\det(\boldsymbol{F})$	$\boldsymbol{F}=\boldsymbol{V}\cdot\boldsymbol{R}$	2-point
	$\boldsymbol{F}=\boldsymbol{R}\cdot\boldsymbol{U}$	
$J=\det(\boldsymbol{U})$	$\boldsymbol{U}^2=\boldsymbol{C}$	reference, material

$\boldsymbol{B}$	left Cauchy–Green deformation tensor
$\boldsymbol{F}$	deformation gradient
$\boldsymbol{C}$	right Cauchy–Green deformation tensor
$\boldsymbol{V}$	left stretch tensor
$\boldsymbol{R}$	rotation tensor
$\boldsymbol{U}$	right stretch tensor
J	Jacobian

3 Mechanical behavior of crystalline structures

In chapter 2, a mathematical framework describing deformations of a body was introduced. The cause of the deformation, a force (or stress), is connected via a constitutive law to the deformation. The origin of the deformation is so far not included in that framework. In material mechanics, it is important to take a closer look on the origin of the deformation. In this chapter, the fundamentals needed to model the mechanical response of crystalline structures are briefly discussed and the derived constitutive models are introduced.

3.1 Crystalline structures

The binding forces in metallic bonding between metal atoms are undirected. This leads to atomic arrangements with the maximum filling in space, the tightest dimensional packing. For a pure metal without any foreign atoms, two closest packages are possible: the face-centered cubic (fcc) and the hexagonal lattice structure (hcp[1], hexagonally closed packed). Both crystal lattices can be interpreted as a combination of closest packed planes. They differ in the order in which the close-packed lattice are piled on one another. Coventionally, each plane with the same orientation is named with the same capital letter, starting from "A". In that notation, hcp has a ABAB... stacking order, while fcc is arranged as ABCABC... That means, the difference between hcp and fcc is the way the different planes are piled. While for hcp, first and third plane have the same orientation and the second one is translated in plane, an fcc lattice consist of three differently orientated planes [1]. The different stacking orders and the resulting unit cells of the hcp and the bcc lattice are shown in fig. 3.1.

A third important lattice structure exist for metals: the body-centered cubic (bcc). It is not a closest packed lattice, but its volume ratio is close to the highest possible ratio achieved by hcp and fcc. A bcc lattice is shown in fig. 3.2, an fcc lattice is shown for comparison in fig. 3.3

Directions and planes in crystal structures are usually described using the MILLER indices. In this notation, a lattice direction or plane is determined by three digits in the case of cubic structures (e.g. bcc and fcc). For hcp a similar notation exist that uses four digits[2]. All of the following examples are taken from [36]. The digits are written in square brackets $[h\,k\,l]$ for the direction given by the vector $\eta \cdot (h, k, l)$, with η being an arbitrary factor. The smallest possible integers are used, meaning notations like $[1/2\,0\,1]$ or $[2\,0\,4]$ are not valid. The correct representation would

[1] The hexagonally closest packed structure is only a model. Real crystallites have structures with a stacking order close to hcp. Thus, to be exact, their structure should be named "hex" instead of "hcp".

[2] This is only for reasons of convenience. As in three-dimensional space only 3 parameters are independent, the 4 digits are linear depending on each other and can equally be expressed by a set of 3 digits only.

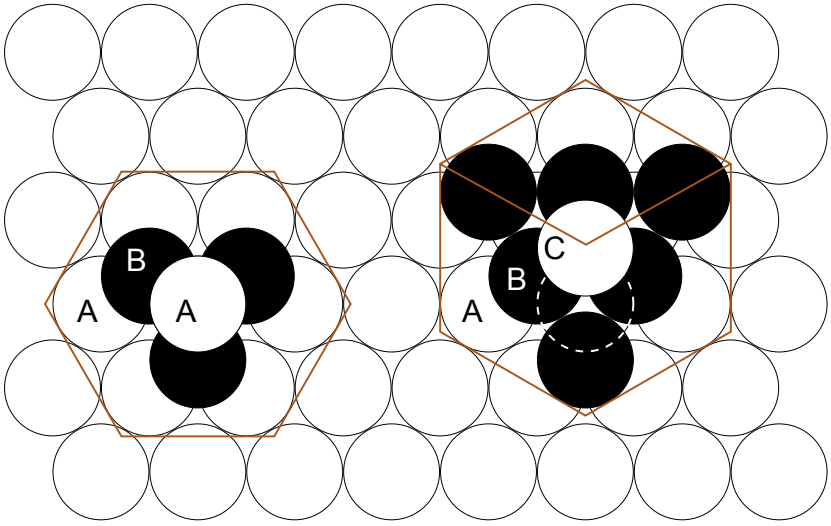


Figure 3.1: Stacking order of the hcp lattice (left) and the fcc lattice (right)[3]

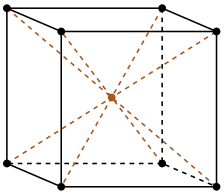


Figure 3.2: Bcc lattice[4]

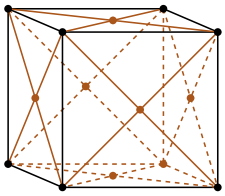


Figure 3.3: Fcc lattice[5]

be $[1\,0\,2]$. This notation ensures that each directions is described by a unique set of integers. A negative direction is denoted by a bar as in $[\bar{1}\,1\,0]$ for direction $(-1, 0, 0)$. The family of crystal directions that are equivalent to the direction $[h\,k\,l]$ is notated as $\langle h\,k\,l\rangle$ (in angle brackets). A similar scheme exists for planes. The plane orthogonal to the direction $\eta \cdot (u, v, w)$ is written in normal brackets: $(u\,v\,w)$. The notation $\{l\,m\,n\}$ (in curly brackets) is used for all planes that are equivalent to $(u\,v\,w)$ by the symmetry of the lattice.

The structure of real crystals or crystallites is different from the idealized lattice. The defects are characterized by their spatial dimension. The following defects exist [1, 36]:

- 0-dimensional: Vacancies, interstitials, antisite defects, substitutional defects, FRENKEL pairs
- 1-dimensional: Dislocations
- 2-dimensional: Grain boundaries, small angle grain boundaries, anti-phase boundaries, stacking faults, twins
- 3-dimensional: Precipitates, inclusions, cavities

0-dimensional defects are not directly simulated in crystal plasticity. The contribution of dislocations and twinning is used as a parameter in the constitutive laws. Grain boundaries and larger precipitates can be directly created by specifying the corresponding geometry. Because of the special importance of dislocations and twins for the derivation of constitutive models, they are briefly described in section 3.3.

[3] File taken from `http://commons.wikimedia.org/wiki/File:Close_packing.svg`, accessed 14th November 2010. The copyright information can be found in appendix C.

[4] File taken from `http://en.wikipedia.org/wiki/File:Lattice_body_centered_cubic.svg`, accessed 14th November 2010. The copyright information can be found in appendix C.

[5] File taken from `http://en.wikipedia.org/wiki/File:Lattice_body_centered_cubic.svg`, accessed 14th November 2010. The copyright information can be found in appendix C.

3.2 Elastic response

Elastic deformation occurs when the atoms in the regular lattice are displaced forcefully, but without changing their neighboring atoms. The bonding between the atoms causes them to fall back to the initial position in a stress-free configuration. Elastic deformation can be described by HOOKE's law as a linear relation between stress and strain.

The simplest stress–strain constitutive relation is HOOKE's law. It describes the mechanical response of a linear elastic material. HOOKE's law reads in the one-dimensional case as $\sigma = E \cdot \varepsilon$ for small deformations. E is YOUNG's modulus and connects linear stress with strain. For the three-dimensional case, the connection between the 2nd order tensors of stress and strain is a 4th order tensor, the stiffness tensor $\mathbb{C}$. The equation reads as:

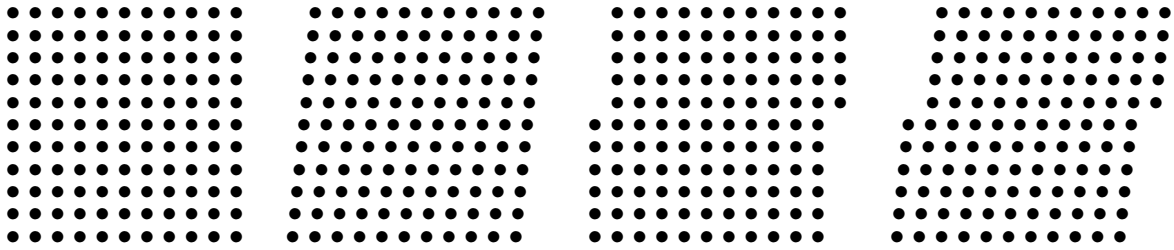
$$\boldsymbol{\sigma} = \mathbb{C} : \boldsymbol{\varepsilon} \tag{3.1}$$

For an isotropic material, the tensor $\mathbb{C}$ depends only on YOUNG's modulus E and the POISSON ratio ν.

3.3 Plastic response

While elastic deformation is the reversible part of deformation that is recovered after the force is removed, plastic deformation remains even when the material is not under loading any more. Metals usually have a combined elastic–plastic response. For small strains (and short loading times), the behavior is usually purely elastic and only after reaching the yield stress (or long holding times), the material starts to deform plastically [31].

The response of a material is not only dependent on the strain, but also on the strain rate. For faster deformation (i.e. higher rate), the plastic deformation requires higher stress compared to a lower rate. This time-dependent behavior is described as viscous. The viscous response of the material must therefore also be modeled to produce applicable results [31].



(a) unstrained reference | **(b)** elastic shear only | **(c)** plastic shear only | **(d)** combined elasto-plastic deformation

Figure 3.4: Elastic and plastic deformation

The difference between plastic and elastic deformation can be explained by the atomic structure of the material. The atoms in crystals are arranged in a regular three-dimensional order, the crystal lattice. Metal bonding is based on the interaction between the positively charged metal ions and free valence electrons. If the change of position exceeds a threshold depending on the radius of

the atom, the atoms get a new closest neighbor and the deformation is permanent. The lowest stress required to do so is called the elastic limit. Fig. 3.4 schematically presents these different types of deformation.

The stress state depends on the elastic deformation. Plastic deformation might relax stresses by changing the shape of the stress-free configuration.

The shear stresses needed to deform a crystalline structure plastically are theoretically (calculated from the atom bonding force) much higher than the measured forces. This can be explained by the existence of dislocations and twinning. Both mechanisms are briefly outlined in the following.

3.3.1 Dislocations

Dislocations are one-dimensional (line) defects in the lattice (e.g. a line of misorder) that can move under shear stress. Thereby atoms break their bonds and rebond with other atoms repeatedly. This leads to a plastic deformation of the material. The energy required to break a single bond is far less than that required to break all the bonds on an entire plane of atoms at once. Dislocations contribute significantly to the deformation of crystalline materials, it is said that they are the carriers of plastic deformation [29, 36].

Dislocations are described by the tangential vector to the line of the dislocation $\boldsymbol{s}$, and the BURGER's vector $\boldsymbol{b}$, measuring length and direction of the dislocation. Depending on the relation between $\boldsymbol{s}$ $\boldsymbol{b}$, two types of dislocation exist. They are called screw dislocation and edge dislocation. A typical representation is shown in fig. 3.5. In the following, the different types of dislocations are briefly characterized:

- Edge dislocation: An edge dislocation is a defect whereby an additional layer of atoms is inserted into the crystal. In an edge dislocation, the BURGER's vector is perpendicular to the line direction. In fig. 3.5b, the extra layer of atoms is inserted at the plane $(1, 2, 3, 4)$. From the fig. it can be seen that the BURGER's vector $\boldsymbol{b}$ is perpendicular to the resulting line defect $\boldsymbol{s}$ at line $(3, 4)$. In the representation, $\boldsymbol{b}$ has the length of the extra layer.

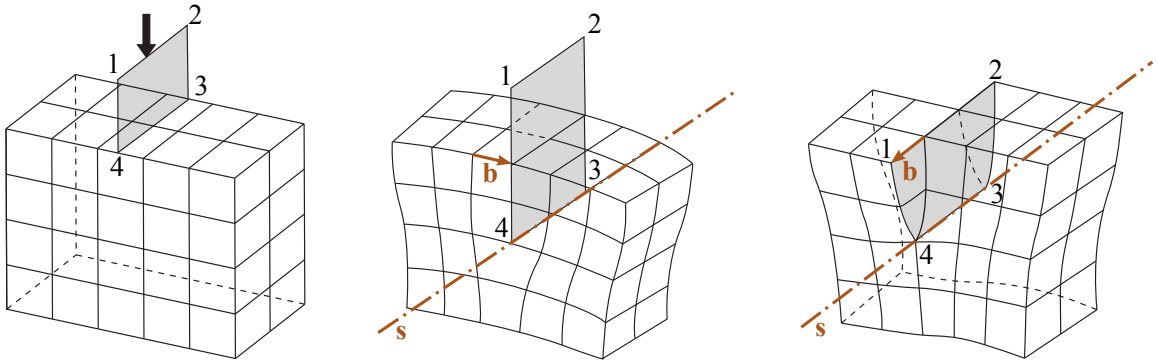
  The stress field of an edge dislocation is complex due to its asymmetry [29].

- Screw dislocation: A screw dislocation is a dislocation in which the BURGER's vector is parallel to the line direction. It can be constructed by cutting along a plane through a crystal and slipping one side by a lattice vector. If the cut only goes part way through the crystal, the result is a screw dislocation. In fig. 3.5c, the crystal is cut at the plane $(1, 2, 3, 4)$. From the fig. it can be seen that the BURGER's vector $\boldsymbol{b}$ is parallel to the resulting line defect $\boldsymbol{s}$ at line $(3, 4)$. As in fig. 3.5b, $\boldsymbol{b}$ has the length of one atom layer.

  Due to its symmetry, the stresses caused by a screw dislocation are less complex than those of an edge dislocation [29].

- Mixed dislocation: In many materials, dislocations are found where the line direction and BURGER's vector are neither perpendicular nor parallel and these dislocations are called mixed dislocations, combining the characteristics of both screw and edge dislocation.

  In real materials, most dislocations are of the mixed type.



(a) undistorted lattice **(b)** edge dislocation **(c)** screw dislocation

Figure 3.5: Undistorted lattice compared to a lattice with an edge and a screw dislocation

Dislocations can move within the crystallite only in certain ways. Dislocation glide is only possible on the so-called slip systems. Each slip system contains of a slip plane and a slip direction. A slip plane is usually a plane with a closest package and a slip direction is a densely packed direction within the slip plane. In the slip system, the deformation caused by a dislocation is the smallest possible in the lattice. Thus, minimum energy is needed for the deformation. While edge dislocations can slip only in the single plane where dislocation and BURGER's vector $\boldsymbol{b}$ are perpendicular, screw dislocations may slip in the direction of any lattice plane containing the dislocations line vector $\boldsymbol{s}$.

Depending on the crystal structure, different planes are densely packed and therefore the preferred slip planes. For fcc it is usually the $\{1\,1\,0\}$ for bcc the $\{1\,1\,2\}$ or $\{1\,1\,2\}$ and for hcp the $\{0\,0\,0\,1\}$. Typical densely packed directions are $\langle 1\,1\,0\rangle$ for fcc and $\langle 1\,1\,1\rangle$ for bcc. For hcp usually $\langle 1\,1\,2\,0\rangle$ is the preferred slip direction.

Edge dislocations—in contrast to screw dislocations—have a second way of moving, called "dislocation climb". This is an effect driven by the movement or diffusion of vacancies through a crystal lattice. As a diffusion dependent effect it is temperature dependent and occurs much more rapidly at high temperatures than low temperatures. In contrast, slip has only a small dependence on temperature [1].

Plastic deformation starts in a slip system, where the maximum shear stress is resolved. The shear stress depends on the tension, the angle between tension and the slip plane normal, and the angle between tension and slip direction. The factor calculated of the cosines of the angles, connecting shear stress and tension is known as the SCHMID factor [36].

The deformation of crystals leads to an increasing dislocation density, as new dislocations are generated during deformation. The interaction of the dislocations hampers the further dislocation motion. If a certain dislocation density is reached or grain rotates due to deformation, another slip system is in a favorable state to deform. The glide system which is activated later deforms at higher stress. This causes a hardening of the metal as with increasing deformation. This effect is known as strain hardening or work hardening. A heat treatment (annealing) causes the defects to heal and can therefore remove the effect of strain hardening [36].

3.3.2 Twinning

A crystal twin consists of two crystals that are separated by a twin boundary. A twin boundary is a special form of a grain boundary, in which some lattice planes and directions are not misordered. The twin boundary can be seen as a lattice plane at which the crystals are mirrored. The crystal planes that are in plane with the twin boundary are not distorted. A twinned structure with two twin boundaries is schematically shown in fig. 3.6. The middle part of the shown structure sheared due to twinning. As can be seen from the fig., the twin boundary is the plane at which two crystals are mirrored. Moreover, it can be seen that the fraction of the sheared part in the crystal is a suitable measure for the deformation of the crystal if the shear angle is known. This information can be used for the implementation of twinning into constitutive models.

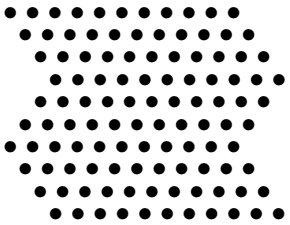


Figure 3.6: Twinned crystal

Twinning is the result of three different mechanisms. Depending on the origin mechanism, the twins are called:

- growth twins
- annealing (or transformation) twins
- deformation (or gliding) twins

Deformation twins are of special importance as they are the result of stress on the crystal after the crystal has formed. Deformation induced twinning allows a mode of plastic deformation in crystalline structures. Deformation twinning occurs if one layer of crystals changes its orientation under shear stress. Similar to the way in which dislocations move in slip systems, twinning is only possible in twin systems where a certain shear stress is resolved.

Depending on the crystal structure, temperature and dislocation density, twinning might require less energy than other deformation modes. Of the three crystal structures, the hcp structure is most likely to twin. Fcc structures usually will not twin because slip is energetically more favorable.

3.4 Constitutive models

The implemented version of the spectral method is closely integrated into the existing routines and can handle all material models available for the FEM-based solvers. The various material laws differ in their complexity and in the effects they take into account.

In general, each constitutive model consists of three parts:

- microstructure parameterization
- structural evolution rates (hardening)
- deformation kinetics (deformation rates)

The microstructure parameterization reflects the degree of simplification, e.g. isotropic versus non-isotropic behavior or phenomenological slip resistance versus dislocation densities. The structure

evolution describes the change of the microstructure parameters during deformation, resulting, for instance, in hardening. It is a function of the current microstucture and the stress state at each point. With the microstructure parameters denoted as vector $\boldsymbol{r}$, the structure evolution is a function of the current microstructure parameters and the CAUCHY stress $\boldsymbol{\sigma}$: $\dot{\boldsymbol{r}} = f(\boldsymbol{r}, \boldsymbol{\sigma})$. The deformation kinetics describe the shear rate (or rates per slip system) at each point. They can also be described as a function of $\boldsymbol{r}$ and $\boldsymbol{\sigma}$: $\dot{\gamma} = f(\boldsymbol{r}, \boldsymbol{\sigma})$. The composition of these three parts results in a coupled system of ordinary differential equations (ODEs) at each point.

To determine the shear rate, the deformation gradient $\boldsymbol{F}$ is decomposed in an elastic and a plastic part. For small deformations, the elasto–plastic decomposition can be calculated additively [31], while for large deformations a multiplicative decomposition is suitable [29]. For the decomposition, a virtual intermediate (or relaxed) configuration is introduced. In this configuration, every material point is elastically unloaded, i.e. only plastically deformed. The transformation from the reference to the intermediate configuration is characterized by the plastic deformation gradient $\boldsymbol{F}_\mathrm{p}$. The subsequent transformation from the intermediate to the current configuration is characterized by the elastic deformation gradient $\boldsymbol{F}_\mathrm{e}$. Therefore, the overall deformation gradient relating the reference to the current configuration reads for a large strain formulation as:

$$\boldsymbol{F} = \boldsymbol{F}_\mathrm{e} \cdot \boldsymbol{F}_\mathrm{p} \tag{3.2}$$

Eq. (3.2) enables the elasto–plastic decomposition of the velocity gradient. This decomposition is additively and reads as:

$$\boldsymbol{L} = \boldsymbol{L}_\mathrm{e} + \boldsymbol{F}_\mathrm{e} \cdot \boldsymbol{L}_\mathrm{p} \cdot \boldsymbol{F}_\mathrm{e}^{-1} \tag{3.3}$$

with $\boldsymbol{L}_\mathrm{p}$ being the plastic and $\boldsymbol{L}_\mathrm{e}$ the elastic velocity gradient. The plastic deformation rate depends, typically rather strongly, on the resolved shear stress and the orientations of the slip systems or twin systems. The elastic velocity gradient $\boldsymbol{L}_\mathrm{e}$ depends on the elastic constants of the material and the orientation of the lattice and is usually much smaller than $\boldsymbol{L}_\mathrm{p}$.

The models used in the examples presented in chapter 8 are briefly explained in the following section. A detailed description of the models and their underlying physics can be found in [28, 29].

3.4.1 J_2-plasticity

The *J_2-plasticity* model is an isotropic constitutive law. It is based on the VON MISES yield criterion described in chapter 2.5. Isotropy results, since the stress state is only determined by the 2nd invariant J_2 of the deviatoric part of the stress tensor $\boldsymbol{\sigma}'$ [15]. For this reason, the orientation of the grains is not considered. The microstructure is characterized by only one state variable, the "flow stress" r.

The deformation rate is given by:

$$\dot{\gamma} = \dot{\gamma}_\mathrm{ref} \left| \frac{\tau}{r} \right|^n \mathrm{sign}(\tau) \tag{3.4}$$

where $\tau = \sigma_\mathrm{vM}/M$ is the resolved shear stress. The factor M is called the TAYLOR factor. It is the inverse of the SCHMID factor (section 3.3.1), depends on the lattice type, and gives the average of the resolved shear stress in all slip systems for the given VON MISES stress σ_vM. The

other variables in eq. (3.4) are a reference shear strain rate $\dot{\gamma}_{\mathrm{ref}}$ and a stress exponent n.

The structure evolution reads as:

$$\dot{r} = |\dot{\gamma}| \; h_{\mathrm{ref}} \left(1 - \frac{r}{r_\infty}\right)^{\omega_{\mathrm{ref}}} \tag{3.5}$$

with the saturation value r_∞ and fitting parameters ω_{ref} and h_{ref}.

The plastic velocity gradient is $\boldsymbol{L}_{\mathrm{p}}$ determined by the following equation:

$$\boldsymbol{L}_{\mathrm{p}} = \frac{\dot{\gamma}}{M} \frac{\boldsymbol{\sigma}'}{||\boldsymbol{\sigma}'||} \tag{3.6}$$

where $\dot{\gamma}/M$ determines the velocity and $\boldsymbol{\sigma}'/||\boldsymbol{\sigma}'||$ the tensorial direction of the deformation rate.

3.4.2 Phenomenological powerlaw

The *phenomenological powerlaw* extends the isotropic *J_2-plasticity* model by considering the orientation of slip systems in the crystal. Twinning is introduced as a second deformation mechanism. The model is able to predict the response of the crystallite under consideration for various lattices type and orientations. Depending on the lattice structure, different slip and twin systems are available. The state variables describing the material condition are "slip resistance" r^α, "twin resistance" r^β, "cumulative shear strain" γ^α, and "twin volume fraction" v^β. Superscripts α and β denote slip respectively twinning.

Shear strain rate due to slip is described in a similar way as for the *J_2-plasticity* model given in eq. (3.4). Instead of using the value of the VON MISES stress to calculate an average resolved shear stress, the resolved shear stress on each slip system τ^α is considered. It depends on the stress $\boldsymbol{\sigma}$ and the so-called SCHMID matrix. The SCHMID matrix is the product of normalized BURGER's vector $\boldsymbol{b}^\alpha$ and the normalized normal vector $\boldsymbol{n}^\alpha$ of the slip system:

$$\tau^\alpha = \sigma_{ij} \frac{b_i^\alpha}{||\boldsymbol{b}^\alpha||} \frac{n_j^\alpha}{||\boldsymbol{n}^\alpha||} \tag{3.7}$$

The shear strain rate $\dot{\gamma}^\alpha$ in each slip system is given by

$$\dot{\gamma}^\alpha = \dot{\gamma}_{\mathrm{ref}} \left|\frac{\tau^\alpha}{r^\alpha}\right|^n \mathrm{sign}\left(\tau^\alpha\right) \tag{3.8}$$

Following the same phenomenology, the twin volume fraction rate is described by:

$$\dot{v}^\beta = \frac{\dot{\gamma}_{\mathrm{ref}}}{\gamma^\beta} \left|\frac{\tau^\beta}{r^\beta}\right|^n \mathcal{H}\left(\tau^\beta\right) \tag{3.9}$$

where $\mathcal{H}$ is the HEAVISIDE function, τ^β the resolved shear stress on each twin system, and γ^β the specific shear due to mechanical twinning . The value of γ^β depends on the lattice type. In fcc lattices it is rather large at $\gamma^\beta = \sqrt{2}/2$ while in hexagonal crystals it depends on the packing ratio and the exact twin type [7].

The relationship between the evolution of state and kinetic variables is given by a vector equation, comparable to the scalar eq. (3.10) of the *J_2-plasticity* model. It connects changes in slip and twin

resistance of the various slip and twin system with the shear rates on all slip and twin systems:

$$\begin{bmatrix} \dot{\boldsymbol{r}}^{\alpha} \\ \dot{\boldsymbol{r}}^{\beta} \end{bmatrix} = \begin{bmatrix} \boldsymbol{M}_{\mathrm{slip-slip}} & \boldsymbol{M}_{\mathrm{slip-twin}} \\ \boldsymbol{M}_{\mathrm{twin-slip}} & \boldsymbol{M}_{\mathrm{twin-twin}} \end{bmatrix} \begin{bmatrix} \dot{\boldsymbol{\gamma}}^{\alpha} \\ \boldsymbol{\gamma}^{\beta} \cdot \dot{\boldsymbol{v}}^{\beta} \end{bmatrix} \tag{3.10}$$

with the four distinct interaction matrices $\boldsymbol{M}_{\mathrm{slip-slip}}$, $\boldsymbol{M}_{\mathrm{slip-twin}}$, $\boldsymbol{M}_{\mathrm{twin-slip}}$, and $\boldsymbol{M}_{\mathrm{twin-twin}}$. The matrices depend in detail on the number of slip or twin systems in the crystal structure and the interactions between these systems.

For more information on the *phenomenological powerlaw* see [30, 37].

4 Green's function method

For the derivation of the spectral method for elastoviscoplastic boundary value problems, the mathematical fundamentals, GREEN's function method and FOURIER transform are presented in this and the following chapter. GREEN's function method is derived and explained in detail in [8, 13].

A GREEN's function $G(x, x')$ is any solution of the equation[1]

$$LG(x, x') = \delta(x - x') \tag{4.1}$$

with δ being the DIRAC delta function (unit impulse function) and $L = L(x)$ a linear differential operator [13, 21].

GREEN's function method can be used to solve inhomogeneous linear differential equations like

$$Lu(x) = f(x) \tag{4.2}$$

For a translation invariant operator, i.e. when L has constant coefficients with respect to x, a convolution operator $G(x - x')$ can be used for $G(x, x')$. Multiplying eq. (4.1) with $f(x')$ and integrating over x' results in:

$$\int LG(x - x')f(x')\mathrm{d}x' = \int \delta(x - x')f(x')\mathrm{d}x' \tag{4.3}$$

where the right side equals $f(x)$ by virtue of the properties of the delta function. Inserting into eq. (4.2) results in

$$Lu(x) = \int LG(x - x')f(x')\mathrm{d}x' \tag{4.4}$$

and, because of $L = L(x)$ does not depend on x' and acts on both sides,

$$u(x) = \int G(x - x')f(x')\mathrm{d}x' \tag{4.5}$$

for a translation invariant operator $L(x)$.

The initial eq. (4.2) is solved by finding $G(x - x')$ and carrying out the integration [13]. As the GREEN's function is not known a priori, the use of this method is limited to cases where a technique is applicable to find the corresponding GREEN's operator. The method for finding GREEN's operator presented in this thesis uses the FOURIER transform. The FOURIER transform and ways to compute it are presented in chapter 5.

[1] Example taken from `http://en.wikipedia.org/wiki/Green%27s_function`, accessed 14th November 2010.

5 Fourier transform

The FOURIER transform (FT) $\mathcal{F}$ is an operation that transforms a function from one domain ($f(x)$) into another domain ($\hat{f}(k)$). The FT is widely used in image and digital signal processing. It is a useful tool in solving differential equations. In the spectral method presented here, the FT allows the equations describing the equilibrium state of the VE to be solved quickly.

When the FT is used on a function in time domain, the domain of the new function is frequency. The FT is therefore also called the frequency domain representation of the original function. The formula to calculate the FT in angular frequency k and frequency $\kappa = k/2\pi$ and in is given by [9]:

$$\mathcal{F}(f(x)) = \hat{f}(k) = \int_{-\infty}^{\infty} f(x) e^{i\,k\,x} \mathrm{d}x \tag{5.1}$$

$$\mathcal{F}(f(x)) = \hat{f}(\kappa) = \int_{-\infty}^{\infty} f(x) e^{i\,2\pi\kappa\,x} \mathrm{d}x \tag{5.2}$$

where $i^2 = -1$ or $i = \sqrt{-1}$ is the imaginary unit.

The inverse transforms are performed via the following two equations [9]:

$$\mathcal{F}^{-1}(\hat{f}(k)) = f(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \hat{f}(k) e^{-i\,k\,x} \mathrm{d}k \tag{5.3}$$

$$\mathcal{F}^{-1}(\hat{f}(\kappa)) = f(x) = \int_{-\infty}^{\infty} \hat{f}(\kappa) e^{-i\,2\pi\kappa\,x} \mathrm{d}\kappa \tag{5.4}$$

One advantage of the FT is the simple way of differentiating and integrating in the frequency domain. The derivative is simply the original function multiplied by $2\pi i\kappa$ or ik:

$$\mathcal{F}\left(\frac{d}{\mathrm{d}x} f(x)\right) = (i\,k) \cdot \hat{f}(k) = (i\,2\pi\,\kappa) \cdot \hat{f}(\kappa) \tag{5.5}$$

The FT of the delta function is 1:

$$\mathcal{F}\left(\delta(x)\right) = \int_{-\infty}^{\infty} \delta(x) \cdot e^{i\,k\,x} \mathrm{d}x = e^0 = 1 \tag{5.6}$$

The convolution theorem states that for $h(x) = (f * g)(x) = \int_{-\infty}^{\infty} (f(x) g(x-y)) \mathrm{d}y$ the FT

$\mathcal{F}(h(x)) = \hat{h}(k)$ is the product of the convolved functions:

$$\hat{h}(k) = \hat{f}(k) \cdot \hat{g}(k) \tag{5.7}$$

For further properties of the FT that are not needed in this work, standard literature such as [3, 9] is available.

5.1 Discrete Fourier transform

It is also possible to apply the FT to discrete data. The discrete FOURIER transform (DFT) is a FT on discrete input functions. It can be used as an approximation of the continuous FT if the data is properly discretized. The DFT works only if the analyzed segment represents one period of an infinitely extended periodic function.

For discrete data, the frequency domain is called the wavenumber domain or wavenumber space. The DFT is shown for frequency κ rather than for angular frequency k to avoid factors of 2π. Each discrete κ stands for one wavenumber, where the number of waves equals the number of points in the input data. With $\kappa = k/2\pi$ the formulas can easily rewritten for angular frequency.

To fulfill the requirement for using the DFT, the space has to be discretized. That is done by defining discrete points in it (FOURIER points, FP) and setting out periodic boundary conditions (BCs) to the volume element under consideration. The periodic BCs expand the space into an infinite space, with the space under consideration being exactly one period of the longest wave [3].

The DFT of a sequence with N complex numbers $f(x_n)$ with $n = 0, \ldots, N-1$ is the sequence $\hat{f}(\kappa_j)$, with $j = 0, \ldots, N-1$ of N wavenumbers $\kappa_0, \ldots, \kappa_{N-1}$ according to:

$$\hat{f}(\kappa_j) = \sum_{n=0}^{N-1} f(x_n) \cdot e^{\frac{i\,2\pi}{N}\,jn}, \quad j = 0, \ldots, N-1 \tag{5.8}$$

The wavenumbers are chosen such that [26]:

$$\kappa_j \equiv \frac{j}{\Delta N}, \quad j = -\frac{N}{2}, \ldots, 0, \ldots, \frac{N}{2} \tag{5.9}$$

with Δ being the sampling interval. Note that κ_j in eq. (5.8) is defined for $N+1$ wavenumbers. As the extreme values $-N/2$ and $N/2$ give the same result, it does not collide with the definition given in eq. (5.8).

The inverse DFT is done by [3]:

$$f(x_n) = \frac{1}{N} \sum_{j=0}^{N-1} \hat{f}(\kappa_j)\, e^{-\frac{i\,2\pi}{N}\,jn}, \quad n = 0, \ldots, N-1 \tag{5.10}$$

It gives the values at each discrete FP that results from the operations conducted in FOURIER space.

For an input of pure real data, i.e., $\mathfrak{I}m(f(x)) = 0$, the transformed data $\hat{f}(\kappa)$ in wavenumber domain is the conjugated complex of $\hat{f}(-\kappa)$: $\hat{f}(\kappa) = \mathfrak{R}e(\hat{f}(-\kappa)) - \mathfrak{I}m(\hat{f}(-\kappa))$. It is symmetric

with respect to the origin on the real part and anti-symmetric on the imaginary part. Therefore, only half of the outputs have to be computed using a DFT algorithm. The other half can be directly obtained from the transform data of the first half. In the same way, for the inverse transform for a data set with $\hat{f}(\kappa) = \Re e(\hat{f}(-\kappa)) - \Im m(\hat{f}(-\kappa))$ only half of it is needed to transform to a set of real data [3, 26].

The DFT was defined for a one-dimensional sequence x_n, where n counts the discrete values of the variable x. The DFT of a three-dimensional function depending on vector $\boldsymbol{x}$ with discrete values $n_\mathrm{x} = 0, \ldots, N_\mathrm{x} - 1; n_\mathrm{y} = 0, \ldots, N_\mathrm{y} - 1; n_\mathrm{z} = 0, \ldots, N_\mathrm{z} - 1$ for the components $x; y; z$ is a multidimensional DFT. It transforms a three-dimensional function of three discrete variables to the FOURIER space. The result is a discrete function depending on $\boldsymbol{\kappa} = (\kappa_1; \kappa_2; \kappa_3)$ with discrete values $j_1 = 0, \ldots, N_\mathrm{x} - 1; j_2 = 0, \ldots, N_\mathrm{y} - 1; j_3 = 0, \ldots, N_\mathrm{z} - 1$. The three-dimensional DFT is—according to [22]—defined by:

$$\hat{f}(\boldsymbol{\kappa}) = \sum_{n_\mathrm{x}=0}^{N_\mathrm{x}-1} \sum_{n_\mathrm{y}=0}^{N_\mathrm{y}-1} \sum_{n_\mathrm{z}=0}^{N_\mathrm{z}-1} f(\boldsymbol{x})\, e^{i\,2\pi\left(\frac{j_1\,n_\mathrm{x}}{N_\mathrm{x}} + \frac{j_2\,n_\mathrm{y}}{N_\mathrm{y}} + \frac{j_3\,n_\mathrm{z}}{N_\mathrm{z}}\right)} \tag{5.11}$$

The inverse of the multidimensional DFT is, analogous to the one-dimensional case, given by [22]:

$$f(\boldsymbol{x}) = \frac{1}{N_\mathrm{x} \times N_\mathrm{y} \times N_\mathrm{z}} \sum_{j_1=0}^{N_\mathrm{x}-1} \sum_{j_2=0}^{N_\mathrm{y}-1} \sum_{j_3=0}^{N_\mathrm{z}-1} \hat{f}(\boldsymbol{\kappa})\, e^{-i\,2\pi\left(\frac{j_1\,n_\mathrm{x}}{N_\mathrm{x}} + \frac{j_2\,n_\mathrm{y}}{N_\mathrm{y}} + \frac{j_3\,n_\mathrm{z}}{N_\mathrm{z}}\right)} \tag{5.12}$$

The multidimensional DFT can be computed by composing an algorithm for a one-dimensional DFT along each dimension. This approach is called a row-column algorithm.

5.2 Fast Fourier transform

The calculation of the DFT as introduced in eq. (5.8) needs $\mathcal{O}(N^2)$ operations. The computing time is increasing quadratically with the number of FPs under consideration. The fast-growing calculation time makes the direct DFT unattractive for use on large data sets. The fast FOURIER transform (FFT) is a group of algorithms that compute the DFT in only $\mathcal{O}(N \log N)$ operations. The FFT is widely used and enables the effective use of the DFT [3].

The most common type of FFT-algorithms is the COOLEY–TUKEY algorithm. It is a divide et impera method, meaning it will divide the whole transformation into smaller parts that are simpler (and faster) to compute. It is based on the idea of breaking down an FT with $N = N_1 \cdot N_2$ points into several FTs of N_1 and N_2. The most common implementation is dividing N repeatedly by 2, resulting in $N_1 = N_2 = N/2$. Therefore it has the requirement that the number of input data (FPs) hast to be a power of two. It is called the "radix-2" variant of the algorithm. Other divisions are also possible, mostly resulting in a loss of performance. Variants of the algorithm using different factors are called the "mixed-radix" variants. Especially poor performance is achieved for prime numbers [3, 16].

The speed of the calculations relies heavily on the speed of the employed FFT. Therefore it is important to implement a FFT with good performance. While proprietary libraries available from sources such as *Intel* are limited to certain processor architectures, freely-available implementations

are usually not as fast and flexible as their commercial counterparts. Most of them are limited to a unidimensional array with the input size being a power of two. A free DFT which has shown a performance comparable with algorithms available under commercial licenses is the *Fastest Fourier Transform in the West* (*FFTW*)[1].

5.3 FFTW

The *Fastest Fourier Transform in the West* (*FFTW*) is a library for computing the DFT. It is free software licensed under the GNU General Public License. The *FFTW* package is developed at the *Massachusetts Institute of Technology* (*MIT*) by M. Frigo and S. G. Johnson. It is a C subroutine library with interfaces to call it from C or Fortran codes. It can compute the DFT in one or more dimensions. Moreover, it can handle arrays of arbitrary input size and has interface to compute the DFT of real data. It also has multiprocessor support using the *LinuxThreads*[2] library or *Open Multi-Processing (OpenMP)*[3]. For larger problems, an interface called *p3dfft*[4] is available that has shown good performance on cluster computers with up to 32 768 cores.

To use *FFTW*, it first has to be compiled with options suitable for the computer architecture on which it should run. The resulting library files are linked to the main program to make the routines available. Three steps are needed to compute a DFT:

1. initialize *FFTW* for each call and create a "plan"
2. perform the actual FFT
3. deallocate the data, destroy the plans

The initialization has to be done once for each transform. It is necessary to declare the type of the DFT and the size of the arrays. Depending on these parameters, *FFTW* determines the fastest algorithm available for the specific DFT and stores the respective plan. For the transform of pure real data to Fourier space the "real to complex" (r2c) interface exists. In the same way, an inverse transform can be done by a "complex to real" (c2r) interface if the output does not contain any imaginary part.

The FFT can then be performed for each plan repeatedly, always using the same optimized plan by passing the variable containing the information about the plan. Depending on the type of transform, *FFTW* provides different interfaces to call the FFT. Depending on the plan created, the calls are slightly different.

When the program is finished and the transforms are no longer needed, the plan and all its associated data should be deallocated. This is done by calling the interface provided by *FFTW* for this task.
More information on *FFTW* can be found in [16].

[1] `http://www.fftw.org`, accessed 14th November 2010.
[2] `http://pauillac.inria.fr/~xleroy/linuxthreads`, accessed 14th November 2010.
[3] `http://openmp.org/wp`, accessed 14th November 2010.
[4] `http://code.google.com/p/p3dfft`, accessed 14th November 2010.

6 Spectral methods

A spectral method is a special algorithm to solve partial differential equations (PDEs) numerically. PDEs describe physical processes in, for instance, thermodynamics, acoustics, or mechanics. Different variants of the algorithm exist. According to [32] the variants include the GALERKIN approach, the τ method [24] and the pseudospectral or collocation approach. The GALERKIN approach is also the basis for the FEM. The main difference between the FEM and the spectral method is the way in which the solution is approximated.

The FEM takes a local approach. It takes its name from the elements on which local ansatz functions are defined. The ansatz functions are usually polynomials of low degree ($p < 3$) with compact support, meaning they are non-zero only in their domain (i.e. in one element). They equal zero in all other elements. The approximate solution is the result of the assembly of the single elements into a matrix. The matrix links the discrete input values with the discrete output values on the sampling points. Thus, the FEM converts PDEs into linear equations. The resulting matrix is sparse because only a few ansatz functions are non-zero on each point. The FEM is able to approximate the solution of partial differential equations on arbitrarily shaped domains. In the three-dimensional case, the elements are typically tetrahedra or hexahedra with edges of arbitrary lengths and thus can be easily fitted to irregularly-shaped bodies.

The FEM has low accuracy (for a given number of sampling points N) because each ansatz function is a polynomial of low degree. To achieve greater accuracy, three different modifications can be used for the FEM [2, 4]:

- h-refinement: Subdivide each element to improve resolution over the whole domain.
- r-refinement: Subdivide only in regions where high resolution is needed.
- p-refinement: Increase the degree of the polynomials in each subdomain.

The different spectral methods can be seen as a variant where p-refinement is applied while the number of elements is limited to one. Spectral methods use global ansatz functions $\phi_n(x)$ in the form of polynomials or trigonometric polynomials of high degree p. In contrast to the low-order shape functions used in the FEM, which are zero outside their respective element, $\phi_n(x)$ are non-zero over the entire domain (except at their roots). Because of this, the spectral methods take a global approach [4]. The high order of the ansatz functions gives high accuracy for a given number of sampling points N. Spectral methods have an "exponential convergence", meaning they have the fastest convergence possible.

If the approximation of the PDEs is done by trigonometric polynomials, it can equally be expressed as a finite FOURIER series. The resulting system of ordinary differential equations (ODEs)

can easily be solved in the FOURIER space. The outstanding performance of this approach is gained from the fact that the transform can be done using effective FFT algorithms (chapter 5.2). As the FOURIER series requires a periodic function (and so does the FFT), spectral methods using FFT can only be used for the solution of infinite bodies. Usually periodic BCs are applied to the domain of interest in order to expand it to an infinite body.

The spectral method for elastoviscoplastic boundary value problems presented here implicitly uses FOURIER series as ansatz functions. Thus, it can only be used for cubic domains with periodic boundary conditions, which fits well to problem of computing RVE responses. However, simulating the behavior of engineering parts of arbitrary shape is not possible. The global approach of spectral methods also has the disadvantage that the convergence is slow if the solution is not smooth. This is a problem for composite materials with high phase contrasts [4, 5, 17]. Because the presented method approximates the function exact at each sampling point (but only at the sampling points, not in between them) it falls into the category of collocation methods [32]. Other names for this type of spectral methods are "interpolating" or "pseudospectral" approach [4].

In the following section, the approximation of a function by a linear combination of ansatz functions is shown. In section 6.2 the spectral method for the small strain formulation for elastoviscoplastic boundary value problems is derived. Its extension to a large strain formulation that can be solved in reference configuration by using the 1st PIOLA–KIRCHHOFF stress and the deformation gradient is outlined in section 6.3.

6.1 Basic concept

Spectral methods are used for the solution of partial differential and integral equations. This is done by writing a function $u(x)$ as a linear combination of global ansatz functions $\phi_n(x)$. If the number of ansatz functions is limited to $N+1$, the approximation reads as (example taken from [4]):

$$u(x) \approx \sum_{n=0}^{N} a_n \phi_n(x) \tag{6.1}$$

This series is then used to find an approximate solution of an equation in the form:

$$L\,u(x) = f(x) \tag{6.2}$$

where L is the operator of the differential or integral equation and $u(x)$ the unknown function. Approximate and exact solution differ only by the "residual function" defined as:

$$R(x; a_0, a_1, \ldots, a_N) = L\left(\sum_{n=0}^{N} a_n \phi_n(x)\right) - f(x) \tag{6.3}$$

The residual function $R(x; a_n)$ is identically equal to zero for the exact solution. The different spectral methods have different approaches to minimizing the residual [4].

6.2 Small strain formulation

The presented small strain formulation was introduced in [19]. Variants of it are also presented in [14, 18, 22, 23]. Different improvements where developed, starting from this basic scheme. In [5, 17] two different formulations are shown that overcome problems associated with high phase contrasts. One possible extension for using the scheme to solve large strain problems according to [12] is given in section 6.3.

Starting point for the derivation presented in [19] is the relationship between stress and strain at the point $\boldsymbol{y}$. For small strain, it is given as:

$$\boldsymbol{\sigma}(\boldsymbol{y}) = \mathbb{C}(\boldsymbol{y}) : \boldsymbol{\varepsilon}(\boldsymbol{u}(\boldsymbol{y})) \tag{6.4}$$

or, when using index notation and EINSTEIN convention:

$$\sigma_{ij} = C_{ijkl}\,\varepsilon_{kl}, \quad i,j,k,l = 1,2,3 \tag{6.5}$$

where the CAUCHY stress σ_{ij} is a symmetric ($\sigma_{ij} = \sigma_{ji}$) tensor of 2nd order, ε_{kl} is the CAUCHY-strain tensor (small strain formulation) and C_{ijkl} is the symmetric fourth order stiffness tensor.

When neglecting body forces, the static equilibrium corresponds to a divergence-free stress field:

$$\operatorname{div} \boldsymbol{\sigma}(\boldsymbol{y}) = \boldsymbol{0} \tag{6.6}$$

The spatial average over the volume under consideration (usually a RVE) of the strain is denoted as $\langle \boldsymbol{\varepsilon} \rangle = \overline{\boldsymbol{\varepsilon}}$. Periodic BCs are applied to the VE, resulting in a periodic displacement field and a periodic strain field. Thus, introducing $\overline{\boldsymbol{\varepsilon}}$ allows the decomposition of the local strain field into its average and a periodic fluctuation $\tilde{\boldsymbol{\varepsilon}}$. By denoting the periodic displacement field as $\boldsymbol{u}^*$, the decomposition reads as:

$$\boldsymbol{\varepsilon}(\boldsymbol{\sigma}(\boldsymbol{y})) = \overline{\boldsymbol{\varepsilon}} + \tilde{\boldsymbol{\varepsilon}}(\boldsymbol{u}^*(\boldsymbol{y})) = \overline{\boldsymbol{\varepsilon}} + \tilde{\boldsymbol{\varepsilon}}(\boldsymbol{y}) \tag{6.7}$$

The average strain $\overline{\boldsymbol{\varepsilon}}$ depends on the current stress state and is spatially constant. The tractions on the opposite sides of the VE with periodic BCs must be anti-periodic to fulfill the static equilibrium.

A homogeneous reference material with stiffness tensor $\overline{\mathbb{C}}$ is introduced to write eq. (6.4) for an infinitely expanded and periodic strain field with eq. (6.7) as:

$$\boldsymbol{\sigma}(\boldsymbol{y}) = \overline{\mathbb{C}} : \tilde{\boldsymbol{\varepsilon}}(\boldsymbol{y}) + \overline{\mathbb{C}} : \overline{\boldsymbol{\varepsilon}} + \underbrace{\left[\mathbb{C}(\boldsymbol{y}) - \overline{\mathbb{C}}\right] : \left[\tilde{\boldsymbol{\varepsilon}}(\boldsymbol{y}) + \overline{\boldsymbol{\varepsilon}}\right]}_{:= \boldsymbol{\tau}(y)} \tag{6.8}$$

The last term in eq. (6.8) is termed fluctuation field and abbreviated as $\boldsymbol{\tau}(\boldsymbol{y})$. With $\operatorname{div} \boldsymbol{\sigma} = \boldsymbol{0}$ (eq. (6.6)) and, since it does not depend on $\boldsymbol{y}$, $\operatorname{div}\left(\overline{\mathbb{C}} : \overline{\boldsymbol{\varepsilon}}\right) = \boldsymbol{0}$, one can write:

$$\operatorname{div}\left(\overline{\mathbb{C}} : \tilde{\boldsymbol{\varepsilon}}(\boldsymbol{y})\right) = -\operatorname{div}\left(\boldsymbol{\tau}(\boldsymbol{y})\right) \tag{6.9}$$

This equation is called a periodic LIPPMANN–SCHWINGER equation.

For $\tilde{\varepsilon}_{kl} = \frac{1}{2}(\tilde{u}_{k,l} + \tilde{u}_{l,k})$ as in small strain theory the equation reads in index notation as:

$$\overline{C}_{ijkl}\, \tilde{u}_{k,lj}(\boldsymbol{y}) = -\tau_{ij,j}(\boldsymbol{y}) \tag{6.10}$$

This is a differential equation with a linear and translation invariant operator on the left side. According to [18, 19] it can be solved with given $\tau_{ij,j}$ by means of GREEN's function method (eq. (4.5)) for $\tilde{u}_k$:

$$\tilde{u}_k(\boldsymbol{y}) = \int\limits_{\mathbf{R}^3} G_{ki}(\boldsymbol{y} - \boldsymbol{y}')\, \tau_{ij,j}(\boldsymbol{y}')\, \mathrm{d}\boldsymbol{y}' \tag{6.11}$$

Integration by parts leads to

$$\tilde{u}_k(\boldsymbol{y}) = \int\limits_{\mathbf{R}^3} \frac{\partial}{\partial s_j} \left[G_{ki}(\boldsymbol{y} - \boldsymbol{y}')\, \tau_{ij}(\boldsymbol{y}') \right] \mathrm{d}\boldsymbol{y}' - \int\limits_{\mathbf{R}^3} \frac{\partial}{\partial s_j} G_{ki}(\boldsymbol{y} - \boldsymbol{y}')\, \tau_{ij}(\boldsymbol{y}') \mathrm{d}\boldsymbol{y}' \tag{6.12}$$

Because $\lim\limits_{s \to \pm\infty} G_{ki} = 0$ the first integral vanishes. With $\partial/\partial s_j G_{ki}(\boldsymbol{y} - \boldsymbol{y}') = -\partial/\partial y_j G_{ki}(\boldsymbol{y} - \boldsymbol{y}')$, the derivative with respect to y of the remaining equation is:

$$\tilde{u}_{k,l}(\boldsymbol{y}) = \int\limits_{\mathbf{R}^3} G_{ki,jl}(\boldsymbol{y} - \boldsymbol{y}')\, \tau_{ij}(\boldsymbol{y}')\, \mathrm{d}\boldsymbol{y}' \tag{6.13}$$

To solve eq. (6.13), GREEN's function is substituted by a Γ-Operator $\Gamma_{ijkl}(\boldsymbol{y} - \boldsymbol{y}')$:

$$-\Gamma_{ijkl}(\boldsymbol{y} - \boldsymbol{y}') = G_{ki,jl}(\boldsymbol{y} - \boldsymbol{y}') \tag{6.14}$$

with—according to [23]—major and minor symmetries. The right hand side of eq. (6.13) is a convolution. With eq. (6.14), it can be written as:

$$\tilde{u}_{k,l}(\boldsymbol{y}) = \tilde{\varepsilon}_{kl}(\boldsymbol{y}) = -\int\limits_{\mathbf{R}^3} \Gamma_{ijkl}(\boldsymbol{y} - \boldsymbol{y}')\, \tau_{ij}(\boldsymbol{y}') \mathrm{d}\boldsymbol{y}' = -\left(\mathbb{\Gamma} * \boldsymbol{\tau}\right)(\boldsymbol{y}) \tag{6.15}$$

As in the convolution each variable interacts with each other variable, the convolution operation can be seen as a stiffness matrix in FEM in which only single components equal zero. The effective calculation of the convolution leads to the fast convergence of the spectral method. According to eq. (5.7), a convolution in real space corresponds to a plain multiplication in FOURIER space. FT applied to the equation, therefore, leads to:

$$\hat{\tilde{\varepsilon}}_{kl}(\boldsymbol{k}) = -\hat{\Gamma}_{ijkl}(\boldsymbol{k})\, \hat{\tau}_{ij}(\boldsymbol{k}) \tag{6.16}$$

with the caret "$\hat{}$" denoting quantities in FOURIER space. The angular frequencies $\boldsymbol{k}$ correspond to $\boldsymbol{y}$ in real space.

To solve the problem iteratively, the given average strain $\overline{\varepsilon}_{kl}$ is applied on the VE and we must assume $\tilde{\varepsilon}^{m=0}_{kl}(\boldsymbol{y}) = 0$, i.e. $\sigma^{m=0}_{ij}(\boldsymbol{y}) = \overline{C}_{ijkl}\overline{\varepsilon}_{kl}$ for the first iteration. The superscript (m) denotes the current iteration step. Solve eq. (6.16) to get a new value for $\tilde{\varepsilon}^{m+1}_{ij}(\boldsymbol{y})$, assuming:

$$\hat{\tilde{\varepsilon}}^{m+1}_{kl}(\boldsymbol{k}) = -\hat{\Gamma}_{ijkl}(\boldsymbol{k})\, \hat{\tau}^{m}_{ij}(\boldsymbol{k}) \tag{6.17}$$

To solve the equation, $\hat{\Gamma}_{ijkl}$ must be known. According to [23], GREEN's function is the solution of the following differential equation:

$$\overline{C}_{ijkl}\, G_{km,lj}(\boldsymbol{y}-\boldsymbol{y}') + \delta_{im}\delta(\boldsymbol{y}-\boldsymbol{y}') = 0 \tag{6.18}$$

with the KRONECKER delta δ_{im} and the unit impulse function $\delta(\boldsymbol{y}-\boldsymbol{y}')$. Using FT on eq. (6.18) with eq. (5.5) and eq. (5.6) leads to:

$$\overline{C}_{ijkl}\, i^2\, k_l k_j\, \hat{G}_{km}(\boldsymbol{k}) + \delta_{im} = 0 \tag{6.19}$$

Using the substitution rule of the KRONECKER delta and with $i^2 = -1$, eq. (6.19) reads as:

$$\hat{G}_{ki}(\boldsymbol{k}) = \left[k_l k_j\, \overline{C}_{ijkl}\right]^{-1} \tag{6.20}$$

$\overline{K}_{ik}(\boldsymbol{k}) := k_l k_j\, \overline{C}_{ijkl}$ are the components of the acoustic tensor $\overline{\boldsymbol{K}}$ of the reference material. The inverse is defined as $\overline{N}_{ik} := \left[\overline{K}_{ik}\right]^{-1} \quad \forall\, \boldsymbol{k} \neq \boldsymbol{0}$.

Applying FOURIER transforms on eq. (6.14) results in:

$$\hat{\Gamma}_{ijkl}(\boldsymbol{k}) = k_l k_j\, \hat{G}_{ki}(\boldsymbol{k}) \tag{6.21}$$

which allows to solve for $\Gamma_{ijkl}(\boldsymbol{k})$ in FOURIER space:

$$\hat{\Gamma}_{ijkl}(\boldsymbol{k}) = k_j k_l\, \overline{N}_{ik}(\boldsymbol{k})|_{(ijkl)} \quad \forall\, \boldsymbol{k} \neq \boldsymbol{0} \tag{6.22}$$

where $|_{(ijkl)}$ denotes symmetrization with respect to all indices, resulting in major an minor symmetries for $\hat{\Gamma}_{ijkl}(\boldsymbol{k})$. The knowledge of $\Gamma_{ijkl}(\boldsymbol{k})$ enables the solution of eq. (6.17) for the strain. Due to the singularity of $\Gamma_{ijkl}(\boldsymbol{k})$, a solution for $\boldsymbol{k} = \boldsymbol{0}$ is not given. But as $\boldsymbol{k} = \boldsymbol{0}$ has an infinite wavelength, $\hat{\varepsilon}(\boldsymbol{0})$ is a priori known as the average strain: $\hat{\varepsilon}(\boldsymbol{0}) = \overline{\varepsilon}$.

A new value for the stress is achieved by:

$$\sigma_{kl}^{m+1}(\boldsymbol{y}) = C_{ijkl}(\boldsymbol{y})\left(\overline{\varepsilon}_{ij} + \tilde{\varepsilon}_{ij}^{m+1}\right) \tag{6.23}$$

where $\tilde{\varepsilon}_{ij}^{m+1}(\boldsymbol{y}) = \mathcal{F}^{-1}\left(\hat{\tilde{\varepsilon}}_{ij}^{m+1}(\boldsymbol{k})\right)$. Using eq. (6.17) and eq. (6.23) as a fix-point algorithm allows the solution of initial eq. (6.4).

The iterations are stopped if equilibrium is reached. This is done by calculating the divergence of the stress field in FOURIER space. The convergence criterion proposed in [14, 19] reads as:

$$\frac{\left\langle |\boldsymbol{k}\cdot\hat{\boldsymbol{\sigma}}^m(\boldsymbol{k})|^2\right\rangle^{\frac{1}{2}}}{|\hat{\boldsymbol{\sigma}}^m(\boldsymbol{0})|} \leq a_{\mathrm{tol}} \tag{6.24}$$

which is the average of the divergence in FOURIER space normalized by the value of the average stress (at wavenumber zero). Usually $a_{\mathrm{tol}} = 10^{-4}$.

The properties of $\mathbb{\Gamma}$ are well known and described for example in [17]. With:

$$\mathbb{\Gamma} * \left(\overline{\mathbb{C}} : \boldsymbol{\varepsilon}\right) = \tilde{\boldsymbol{\varepsilon}} \tag{6.25}$$

follows:

$$\tilde{\boldsymbol{\varepsilon}}^{m+1}(\boldsymbol{y}) = \tilde{\boldsymbol{\varepsilon}}^{m}(\boldsymbol{y}) - \mathbb{\Gamma}(\boldsymbol{y}) * \boldsymbol{\sigma}^{m}(\boldsymbol{y}) \tag{6.26}$$

For an equilibrated stress field $\boldsymbol{\sigma}$ no correction of the strain field is needed:

$$\mathbb{\Gamma} * \boldsymbol{\sigma} = \boldsymbol{0} \tag{6.27}$$

The simplified problem can be summarized as:

$$\boldsymbol{\varepsilon}^{m+1}(\boldsymbol{y}) = \boldsymbol{\varepsilon}^{m}(\boldsymbol{y}) - \mathcal{F}^{-1} \underbrace{\left(\hat{\mathbb{\Gamma}}(\boldsymbol{k}) : \mathcal{F}\left(\boldsymbol{\sigma}^{m}(\boldsymbol{y})\right)\right)}_{= \Delta\hat{\tilde{\boldsymbol{\varepsilon}}}^{m}(\boldsymbol{k}) \,=\, \Delta\hat{\boldsymbol{\varepsilon}}^{m}(\boldsymbol{k})} \tag{6.28}$$

The corresponding algorithm is especially efficient as there is no need to build the (intermediate) fluctuation field $\boldsymbol{\tau}(\boldsymbol{y})$. Furthermore, the equilibrium check (eq. (6.24)) can be performed at no additional cost, since the FOURIER transform of the stress field is required in eq. (6.28).

The presented scheme uses the small strain approximation and is therefore not suitable for the more general case of large strains involving a rotational part. A formulation valid in these circumstances is presented in the following section.

6.3 Large strain formulation

In [12], two possible methods are presented to extend the small strain formulation to problems with large deformations. In this thesis, the implementation of the variant using the LAGRANGian description is outlined. Instead of using the CAUCHY stress and strain, it is written in terms of the 1st PIOLA–KIRCHHOFF stress $\boldsymbol{P}(\boldsymbol{x})$ and the deformation gradient $\boldsymbol{F}(\boldsymbol{x})$. As those two quantities are 2-point tensors, it allows the problem to be solved in the reference configuration only. A shape update of the FP grid is not needed, as $\boldsymbol{F}(\boldsymbol{x})$ performs the push forward from the reference to the current configuration (chapter 2). The deformation gradient $\boldsymbol{F}(\boldsymbol{x})$ is incrementally calculated from the former one. The deformed configuration is not needed for the solution by means of the spectral method. For postprocessing the current configuration can be computed from the local deformation gradient at each point.

For an arbitrary constitutive law, the stress at each point is a function of the deformation gradient (or equivalently of the displacement gradient):

$$\boldsymbol{P}(\boldsymbol{x}) = f\left(\boldsymbol{F}(\boldsymbol{x})\right) \tag{6.29}$$

$$\boldsymbol{P}(\boldsymbol{x}) = f\left(\boldsymbol{H}_0(\boldsymbol{x})\right) \tag{6.30}$$

with equilibrium condition in reference configuration

$$\operatorname{div}\left(\boldsymbol{P}(\boldsymbol{x})\right) = \boldsymbol{0} \tag{6.31}$$

Eq. (6.29) has a similar form as eq. (6.4). Thus, the problem can be solved in a similar way,

resulting in:

$$\boldsymbol{F}^{\mathrm{m}+1}(\boldsymbol{x}) = \boldsymbol{F}^{\mathrm{m}}(\boldsymbol{x}) - \mathcal{F}^{-1}\left(\hat{\mathbb{\Gamma}} : \mathcal{F}(\boldsymbol{P}^{m}(\boldsymbol{x}))\right) \tag{6.32}$$

with

$$\hat{\Gamma}_{ijkl}(\boldsymbol{k}) = \overline{N}_{ik}(\boldsymbol{k}) k_j k_l|_{(ik)(jl)} \tag{6.33}$$

where $|_{(ik)(jl)}$ denotes symmetrization with respect to i, k and j, l only. Thus, in contrast to $\hat{\Gamma}_{ijkl}$ for the small strain formulation, the Γ-operator for the large strain formulation has only the minor symmetries.

A slightly different method is used to check for equilibrium for the large strain formulation. It gives comparable results to the criterion used for the small strain formulation, but is easier to calculate:

$$\max_{\boldsymbol{k}} \frac{|\boldsymbol{k} \cdot \hat{\boldsymbol{P}}^{m}(\boldsymbol{k})|}{|\hat{\boldsymbol{P}}^{m}(\boldsymbol{0})|} \leq a_{\mathrm{tol}}. \tag{6.34}$$

Usually where $a_{\mathrm{tol}} = 10^{-4}$.

6.3.1 Numerical aspects

The formulations for small and large strain presented here are solely analytical. As described earlier in this chapter, spectral methods approximate the solution by a sum of ansatz functions. There is no need to express the stress or strain field by ansatz functions in real space. A discretization can implicitly done by using the DFT to approximate the FT, thus approximate the infinite FOURIER series by finite ones. The chosen number of FPs determines the order of the FOURIER series, i.e. the accuracy.

7 Implementation

At the *Max-Planck-Institut für Eisenforschung* GmbH (*MPIE*), crystal plasticity finite element methods (CPFEM) are used to simulate the mechanical response of crystalline materials [29]. A RVE is used to predict the material answer to certain load cases. Its size depends on the phase contrasts and the orientation of the single grains under consideration [10].

At the *MPIE*, several routines have been developed for the calculation of crystal plasticity phenomena. The routines provide facilities to use several constitutive laws on the grain level—from simple phenomenological to physics-based models. The routines are included in a flexible framework with interfaces to commercial FEM solvers like *MSC.Marc*[1] or *Abaqus*[2].

In this chapter, it is shown how a spectral method using FFT is closely connected to the existing material routines. The spectral method is used as an alternative to FEM-based solvers. The extension to use the spectral method consists of routines to define the question to be solved, a method to calculate the deformation state fulfilling the given stress BCs, and the implementation of an FFT code. The use of the existing material routines is also clarified in this chapter. The resulting algorithm in short form is given at the end of the chapter. Application examples of the implemented solver are shown in chapter 8.

The CPFEM code and its extension using a spectral method is written in *Fortran*. *Fortran* as the first developed high-level programming language was introduced in 1954. Its name is an abbreviation of **for**mula **tran**slator. Although it is a general-purpose language, it is especially suited (and widely used) in the field of high-performance scientific computing. *Fortran* was successively enhanced, resulting in a number of versions. The new versions largely retain compatibility with previous versions. The newer versions are named after the year in which they were introduced [27]. The latest version is *Fortran* 2003, which is an extension of the widely-used *Fortran* 90/95. The version used for this implementation is *Fortran* 90/95.

The solver using a spectral method is wrapped around the material routines in a similar way to the FEM solvers. To use the interfaces developed for the FEM solvers, the functionality of the FEM tools has to be emulated. Each step from setting up the problem to the postprocessing has to be implemented to get a stand-alone executable. The information how to compile the source code and link the files in order to obtain an executable file is stored in a "makefile". The makefile is given in listing A.4.

[1] `http://www.mscsoftware.com/`, accessed 14th November 2010.
[2] `http://www.simulia.com/products/abaqus_fea.html`, accessed 14th November 2010.

7.1 Problem set-up

The problem to be solved is specified by three files:

1. the geometry specification file `*.geom`,
2. the material configuration file `material.config`,
3. the load case file `*.load`.

The use and the structure of the files is described in the following subsections. Example files are given in appendix B. The files are designed for the routines developed at the *MPIE*. They are modeled for easy readability for humans, rather than being optimized to give small file sizes. All files can be easily edited by standard text editors.

7.1.1 Geometry specification

To apply the spectral method to elastoviscoplastic boundary value problems, the volume under examination has to be discretized by a point grid. A VE consisting of 200 grains and discretized by 64 FOURIER points (FPs) in each direction is shown in fig. 7.1. Due to the simple geometry of the VE (hexahedra) with a regular FFT grid, no complex meshing as for the FEM has to be carried out. The information about the material at each FP and the discretization of the VE is stored in the geometry file (file extension `.geom`). An example geometry file consisting of five grains that is discretized by three FPs in each direction is shown in listing B.1. The geometry file is subdivided into a header and a main part.

In the header, the dimension of the geometry and its discretization is specified. The size of the cuboid VE is described by its three space dimensions. The keyword is `dimension`, followed by three pairs of letters and floating point numbers. Each of the letters `x`, `y`, and `z` stands for one dimension and is followed by the corresponding value. In a similar way, the number of FPs is specified. The keyword is `resolution` followed by `a`, `b`, and `c` and the number of FPs in each direction as an integer value. A "homogenization scheme" can also be used. This is done by using the keyword `homogenization` followed by a number referencing to a certain homogenization scheme[3]. The order of the keywords is arbitrary as long as they are at the beginning of the file.

The header is followed by the actual information about the geometry. Each row contains the information for one FP. Counting starts at one and goes up to the total number of FPs. The points are arranged in a linear list with the component x of the position vector $(x; y; z)$ changing fastest and z changing slowest. The single integer in each line stands for the material that is used at the specified position (referencing a particular microstructure defined in the `material.config` file).

To use the existing routines developed for the FEM-based solvers, a structure comparable to the one used by the FEM has to be emulated. Therefore, a linear element with reduced integration capacity (called *C3D8R* by *Abaqus*) is pretended. The element is displayed in fig. 7.2. This

[3] Homogenization techniques allows it to determine the average response of a cluster of grains. Homogenization is used in two-level approaches for the calculation of engineering parts. More information on homogenization techniques can be found in [29].

element is a hexahedra, its shape is specified by the position of the nodes on its eight corners. For FEM solving, it has one integration point (or GAUSS point) in the center of the element. For the spectral method, this point is the FP. Each face (roman numerals) and each node (arabic numerals) of the element has a unique number as shown in fig. 7.2. The counting scheme on the element is basically arbitrary, for compatibility reasons it follows the convention that is used in the standard FEM codes.

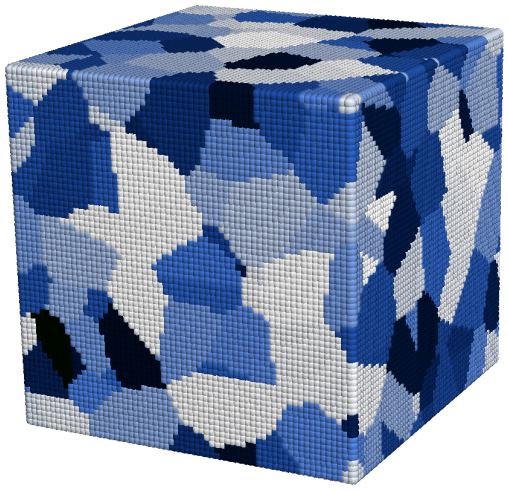


Figure 7.1: Volume element consisting of 200 grains discretized by 64^3 FPs

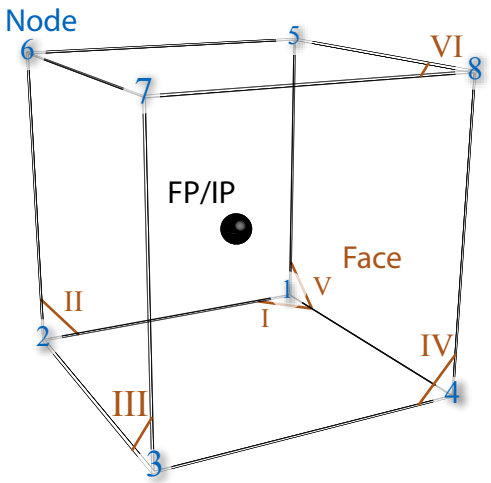


Figure 7.2: Hexahedral finite element with one integration point

Each point in a VE as shown in fig. 7.1 corresponds to one FP shown in fig. 7.2. The VE is built by generating a numbering scheme for a global grid of nodes and FPs in the VE and connecting the local numbering of each element with that global scheme. The complete VE consists of $N_{\mathrm{FP}} = N_{\mathrm{x}} \times N_{\mathrm{y}} \times N_{\mathrm{z}}$ FPs and $(N_{\mathrm{x}}+1) \times (N_{\mathrm{y}}+1) \times (N_{\mathrm{z}}+1)$ nodes, where $N_{\mathrm{x,y,z}}$ are the number of FPs as defined in the geometry file along each of the three dimensions.

An intermediate position for each FP and each node is calculated first for a hypothetic VE with the side lengths equal the number of FPs in the corresponding direction. Counting starts at one and goes op to the number of FPs for each component of the intermediate position vector. The positions are equispaced in each direction. By convention the position for the first dimension runs the fastest. Therefore the FPs with numbers from 1 to N_{x} are at intermediate positions $\boldsymbol{d}(1,\ldots,N_{\mathrm{x}}) = (1,\ldots,N_{\mathrm{x}};1;1)$. Element No. $N_{\mathrm{x}}+1$ is located at $\boldsymbol{d}(N_{\mathrm{x}}+1) = (1;2;1)$ and so on. The corresponding scheme for numbering the nodes works the same way, where $N_{\mathrm{x,y,z}}$ is replaced by $N_{\mathrm{x,y,z}}+1$ as there is one node more than FPs in each dimension.

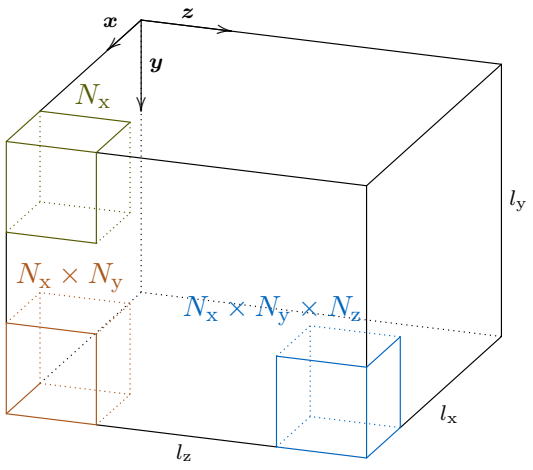


Figure 7.3: Assembly of the VE

The physical coordinates of nodes and FPs can be calculated from the intermediate position vector and the length specified for the three dimensions. The calculation is explained for the one-dimensional case in the following. The length of the side in the example is denoted as l_x and the number of FPs in that direction equals N_x. The first node is located at $(d_1(1)-1)/N_\mathrm{x} \times l_\mathrm{x} = 0$, the second at $(d_1(2)-1)/N_\mathrm{x} \times l_\mathrm{x}$ and so on. The last node $(N_\mathrm{x}+1)$ is located at l_x and—due to the periodic BCs—falls together with the first node. The FPs are located in the middle of their element, therefore the first FP is positioned exactly between the first two nodes, thus at position $(d_1(1)-1/2)/N_\mathrm{x} \times l_\mathrm{x}$. The second FP is in between node two and three and so on. The extension to the three-dimensional case is straightforward.

As each element has exactly one FP, each element gets the same global number as the associated FP. All elements have eight nodes that surround the FP. Due to the periodic BCs, each node is always shared by eight elements. The assembly of the VE is schematically shown in fig. 7.3, where three elements and their numbers are given exemplary.

The implemented numbering routine is able to handle arbitrarily shaped elements with various numbers of integration points (used only by FEM-based solvers). It connects each element with its neighboring elements by providing information as to which nodes and faces are shared by which elements. The result is a "virtual mesh" that allows the results achieved by the spectral method to be presented in a compatible way to interfaces developed for the FEM solvers. The routines for building the virtual mesh are given in listing A.3. The functions named `*spectral*` were implemented especially for the spectral method. All other functions in the source code file are generic. They are also used for the FEM-based solvers. The functions used solely for *MSC.Marc* or *Abaqus* are removed from the appended file in order to reduce the length of the appendix.

7.1.2 Material specification

The specification of the material behavior relies on an existing framework developed at the *MPIE*. Details on the CPFEM implementation can be found in [28]. The framework uses a very flexible material representation that allows the description of single crystals as well as multiphase materials. To achieve this, all material information is stored in a configuration file named `material.config`, which is read during the initialization of the material subroutine. An simple example file describing two grains is shown in listing 7.1.

The file is divided into several parts. Each part starts with a label of the form `<keyword>`, where valid keywords are: `homogenization`, `microstructure`, `crystallite`, `phase`, and `texture`. The parts can hold multiple sections, of which each starts with a label of the form `[entryname]`. `entryname` is an arbitrary string. In the model, a material is assigned to an FP by specifying its homogenization and microstructure. For this purpose, the sections are consecutively numbered so that they can be indexed by their position within the part. In this way it is possible to use different materials, even different constitutive laws, within one model. In the following, the keywords are briefly described:

- `<homogenization>` The homogenization scheme for the whole VE. Since RVEs do typically not contain more than one phase per material point, a homogenization scheme is not required. Therefore, a homogenization scheme where the keyword `Ngrains` is set to `1` and

type equals `isostrain` should be used.

- `<microstructure>` Each microstructure specified in this part corresponds to one or more grains in the VE. Each microstructure consists of one ore more constituents (specified by `(constituents)`). The constituents are characterized by their `phase`, their `texture`, and their `fraction` on the microstructure. The keyword `crystallite` is used to specify the desired output variables. The keywords `crystallite`, `phase`, and `texture` refer to the respective keywords in angle brackets given below.
- `<crystallite>` This part is used to specify which information of each FP should be recorded. Each output option is specified after a keyword `(output)`.
- `<phase>` After this keyword the information about the mechanical behavior of each phase is given. The most important entry for each phase is the key `constitution`. Its value defines which constitutive model is used for this phase with remaining entries in the section giving the necessary parameters and output options connected to the constitutive law. Possible keywords are: `j2`, `phenopowerlaw` (the constitutive models used for the examples presented in chapter 3.4), `nonlocal`, and `dislotwin`.
- `<texture>` The orientation (distribution) of the constituent.

Listing 7.1: Example material configuration file

```
<homogenization>
[SX]                                          # homogenization scheme 1
type    isostrain
Ngrains 1

<microstructure>
[Grain001]                                    # microstructure 1
crystallite     1
(constituent)  phase 1   texture   1   fraction 1.0
[Grain002]                                    # microstructure 2
crystallite     1
(constituent)  phase 1   texture   1   fraction 1.0

<crystallite>
[all]                                         # crystallite 1
(output) orientation
(output) p                                    # 1st Piola-Kichhoff stress tensor

<phase>
[Aluminum_phenopowerlaw]                      # phase 1
constitution            phenopowerlaw
(output)                shearrate_slip
lattice_structure       fcc
c11                     170.17e3
c12                     114.92e3
c44                     60.98e3
gdot0_slip              1.0

<texture>
[Grain001]                                    # texture 1
(gauss)  phi1 359.121452   Phi  82.319471   Phi2 347.729535   scatter 0   fraction 1
```

In the example given in listing 7.1, one homogenization scheme named `[SX]` of the type `isostrain` is defined. Thus, in the geometry file `*.geom` the keyword `homogenization` should be set to `1`, referring to the only available scheme. Two microstructures, named `[Grain001]` and `[Grain002]`, are specified. Since the parameters for both microstructures are identical, referring to the only available sections in the various parts, both microstructures are identical in their mechanical behavior.

In listing B.3 the material specification used for the simulations discussed in chapter 8 is given. For reasons of convenience, only five of the hundred differently oriented grains are defined in the given example.

7.1.3 Load case specification

The load case file (extension `.load`) holds information about stress and strain applied to the VE. Each line defines the BCs to be applied for a particular period of time. A sequence of loads can thus be specified by additional lines, allowing cyclic loading to be applied, for example. Per line, the nine components of the velocity gradient (keyword `l` or `velocitygrad`) and the stress (keyword `s` or `stress`), the period of time (keyword `t`, `time` or `delta`), and the number of time-discretizaion steps (keyword `n`, `incs`, `increments` or `steps`) are given. Since components for stress and velocity gradient are mutually exclusive, a # should be used for the stress component where the velocity gradient is given and vice versa.

An example file prescribing the four load cases used for the simulations discussed in chapter 8 is given in listing B.3.

7.2 Initialization

The routine starts with reading in the information from the files specifying the problem under consideration. The two arguments passed to the compiled executable are the geometry file and the load case file. The location of the load case file determines the working directory where the material configuration file `material.config` is searched for. There is the possibility of giving more parameters regarding numerics and debugging by putting the optional files `numerics.config` and `debug.config` into that directory. The parameters control the behavior of the different routines of the subroutines developed at the *MPIE*. If one of the three mandatory files is not available, the program will exit and report an error.

7.2.1 Load case

The information about the load case (`*.load`) is read in first. A sanity check is done to find out whether all load cases are completely defined, meaning they all have velocity gradient respectively stress BCs, loading period and steps defined. The program will exit with an error message if one component of the stress BC and the velocity gradient is doubly or not at all defined.

7.2.2 Geometry

For the undeformed reference configuration, the deformation is set to identity. To initialize the material point model the interface routine `CPFEM_general` is called. This then reads in the geometry file, generates the connections between the elements, sets up the constitutive laws, and returns the elastic stiffness for each material point.

7.2.3 FFTW

The FFT used in the presented implementation is *FFTW* in version 3.2.2. As described in chapter 5.3, it has to be initialized by creating a plan. This is done during the initialization by calling the function provided by *FFTW* as soon as the information about the FP grid is available. Each component of the 1st PIOLA–KIRCHHOFF stress must be transformed to the discrete FOURIER space (wavenumber space). The result of the operations in wavenumber space must be inversely transformed to give the change in the deformation gradient.

The quantities used in FOURIER space, the Γ-operator, the stress field, and the change of the deformation gradient at each point, are quantities originated in real space and without an imaginary part. As described in chapter 5, this allows the use of the efficient "real to complex" (r2c) and "complex to real" (c2r) interfaces of *FFTW* for the transformation from real to wavenumber space and inverse [19]. By using these interfaces, only $N_\mathrm{x}/2+1$ instead of N_x values for the first (or any other) dimension are transformed [26].

Like most of the FFTs, *FFTW* stores the information in wrap-around order. In first position, the value of wavenumber (angular frequency) $k=0$ is stored. It is followed by the value of the smallest positive wavenumber, the value of the second smallest one etc., up to the value of the most positive wavenumber (which is ambiguous with the value of the most negative wavenumber). Values of negative wavenumbers follow, from the value of the second-most negative wavenumber up to the value of the wavenumber just below zero.

7.2.4 Wavenumbers and Γ-operator

The wavenumbers are equally distributed and depend on the size of the VE and its discretization. Each component k_i of the wave vector $\boldsymbol{k}$ is a linear sequence from $-N_i/2$ to $N_i/2$ divided by the size of the VE in direction l_i [19, 26]. They must be arranged in wrap-around order since the transformed data is stored as described in section 7.2.3. The wavenumbers are used to calculate the Γ-operator according to eq. (6.33) and to determine equilibrium in FOURIER space as given in eq. (6.34).

The Γ-operator is directly calculated for each wavenumber as a product of reference stiffness and wavenumber as described in chapter 6.3. In the implementation presented here, the Γ-Operator is calculated only for $N_\mathrm{x}/2+1$ wavenumbers in the first dimension, using the symmetry resulting of the transformation of real-only data to FOURIER space[4] [19]. In combination with the r2c and c2r FFT this saves half of the computation time and almost half of the memory needed to store the Γ-operator and the stress field in FOURIER space.

[4] The Γ-operator is easier to compute in FOURIER space as shown in chapter 6.2. Nevertheless it is a quantity originated in real space.

The reference stiffness (average stiffness) for the VE is computed by summing up the stiffness tensor from each FP and dividing it by the total number of FPs:

$$\frac{1}{N_{\mathrm{FP}}}\sum_{n=1}^{N_{\mathrm{FP}}}\frac{\mathrm{d}\boldsymbol{\sigma}\left(\boldsymbol{F}(n)\right)}{\mathrm{d}\boldsymbol{\varepsilon}(n)}=\frac{1}{N_{\mathrm{FP}}}\sum_{n=1}^{N_{\mathrm{FP}}}\frac{\mathrm{d}\boldsymbol{P}\left(\boldsymbol{F}(n)\right)}{\mathrm{d}\boldsymbol{F}(n)}=\overline{\mathbb{C}},\qquad \text{for } \boldsymbol{F}(n)=\overline{\boldsymbol{F}}=\boldsymbol{I} \tag{7.1}$$

7.3 Execution loop

By the end of the initialization, the basic information about load cases and geometry is written to the results file (section 7.4). An outer loop over all load cases is performed. Inside the load case loop, an inner loop over the steps of the current load case is carried out. Two conditions have to be fulfilled by the end of each step: the stress BCs and the mechanical equilibrium. The average stress state is determined by the prescribed average deformation gradient $\overline{\boldsymbol{F}}$. The components of $\overline{\boldsymbol{F}}$ that are not directly given by the prescribed velocity gradient, i.e. the components where a stress BC is given, have to be adjusted in order to fulfill the applied stress BCs. Mechanical equilibrium depends on the deformation gradient on each point $\boldsymbol{F}(x)$. For each step, the calculations are done iteratively until the stress BCs are fulfilled and the mechanical equilibrium is reached within the given tolerance.

7.3.1 Global deformation gradient

The global deformation gradient $\overline{\boldsymbol{F}}$, i.e. the average over the whole VE, is directly calculated from the prescribed velocity gradient and the time stepping given in the load case file. Components where a stress BC applies (no velocity gradient given) are skipped at first. These components of $\overline{\boldsymbol{F}}$ have to be adjusted in order to fulfill the applied stress BCs, as the values of $\overline{\boldsymbol{F}}$ leading to a state fulfilling these BCs is a priori unknown.

The stress on each point is computed by calling `CPFEM_general`. The input arguments for the call are the deformation gradient at the end of the former step $\boldsymbol{F}^0(\boldsymbol{x})=\overline{\boldsymbol{F}}^0+\tilde{\boldsymbol{F}}^0(\boldsymbol{x})$, the newly predicted deformation gradient $\boldsymbol{F}^m(\boldsymbol{x})$ of the current iteration m, the duration of the deformation, the element number, the integration point (for the spectral method the FP), and the temperature. `CPFEM_general` returns $\boldsymbol{P}$, $\boldsymbol{\sigma}$, $\mathrm{d}\boldsymbol{P}/\mathrm{d}\boldsymbol{F}$ and $\mathrm{d}\boldsymbol{\sigma}/\mathrm{d}\boldsymbol{E}_{\mathrm{t}}$ at each point. Only the 1st Piola–Kirchhoff stress tensor $\boldsymbol{P}^m(\boldsymbol{x})$ is used for now to calculate the average stress $\overline{\boldsymbol{P}}^m$ on the VE. With the deviation between $\overline{\boldsymbol{P}}^m$ and given BC denoted as $\Delta\overline{\boldsymbol{P}}^m$, evaluating

$$\Delta\overline{\boldsymbol{F}}^{m+1}=\overline{\mathbb{S}}:\Delta\overline{\boldsymbol{P}}^m \tag{7.2}$$

for the components where a stress BC is located gives the correction of the deformation gradient. The tensor $\overline{\mathbb{S}}$ is the inverted stiffness $\overline{\mathbb{C}}$ calculated during initialization.

The value for the correction of $\overline{\boldsymbol{F}}$ calculated by eq. (7.2) tends to be too strong. For example, if uniaxial stress is applied, the first guess for the deformation gradient would be strain in the desired direction while the other directions are unstrained. This results in a volume expansion that leads to high stresses. The problem is therefore solved iteratively with the value of $\Delta\overline{\boldsymbol{F}}$ damped by a

factor ρ:

$$\overline{\boldsymbol{F}}^{m+1} = \overline{\boldsymbol{F}}^{m} + \rho\,\Delta\overline{\boldsymbol{F}}^{m+1} \tag{7.3}$$

A simple damping scheme is applied. The algorithm starts with a high damping factor, e.g. $\rho = 0.05$, i.e. the value of $\overline{\boldsymbol{F}}$ is only corrected by 5% of the value calculated by eq. (7.2). If the correction of on component ij is too strong, the algorithmic sign of $\Delta\overline{F}_{ij}^{m+1}$ differs from the one of $\Delta\overline{F}_{ij}^{m}$. In this case, a weaker correction is applied on the component for the next step $m+2$ to prevent oscillations around the correct solution. Similarly, the correction values for $m+2$ is increased if the previous correction step is to small. While being far from perfect, this simple algorithm significantly improves the speed of calculation. A more advanced and possibly more efficient scheme is not introduced so far. However, the described rather time consuming calculation is only needed for the first step of each load case, when the $\overline{F}_{ij}^{m=1}$ cannot be predicted from the former steps. On each following step of the load case a continuation along the trajectory of the two former steps is predicted for the components of the deformation gradient where stress BCs are applied. This assumption is correct for the linear elastic deformation and almost correct for plastic deformation as long as the change in slope is not too strong. It significantly improves the performance of the computation of the global deformation gradient.

Each local deformation gradient is corrected by the change of the average deformation gradient:

$$\boldsymbol{F}^{m+1}(\boldsymbol{x}) = \boldsymbol{F}^{m}(\boldsymbol{x}) + (\overline{\boldsymbol{F}}^{m+1} - \overline{\boldsymbol{F}}^{m}) \tag{7.4}$$

and a new stress field is computed by calling `CPFEM_general`.

The stress BCs are regarded as fulfilled if the largest deviation is less than 0.8% of the highest stress component. This relative tolerance considers that at low stress states more accuracy is needed than at high stress states. The criterion exceeds the desired accuracy by the end of each step as the calculation of the local deformation gradient is likely to amplify the error.

When a deformation state fulfilling the stress BCs is finally known, the local deformation gradient is calculated by means of the spectral method in order to achieve the mechanical equilibrium.

7.3.2 Local deformation gradient

At the first step of each load case a homogeneous deformation (i.e., vanishing fluctuation) is assumed on each FP for the first iteration. That means that the global deformation gradient is applied at each FP: $\boldsymbol{F}^{1}(\boldsymbol{x}) = \overline{\boldsymbol{F}}^{1}$. From the second to the last step of each load case, the predicted deformation gradient for the first iteration $\boldsymbol{F}^{1}(\boldsymbol{x}) = \overline{\boldsymbol{F}}^{1} + \tilde{\boldsymbol{F}}^{1}(\boldsymbol{x})$ is the continuation at the rate of the former step.

The calculation starts with a call to `CPFEM_general` to get the 1st PIOLA–KIRCHHOFF stress tensor $\boldsymbol{P}^{m}(\boldsymbol{x})$ on each point resulting from the current deformation gradient $\boldsymbol{F}^{m}(\boldsymbol{x})$. The stress field is transformed to FOURIER space and multiplied by the Γ-operator. The convergence is checked in FOURIER space according to eq. (6.34). As described in chapter 6.3, the inverse transform of $\hat{\mathbb{\Gamma}}(\boldsymbol{k}) : \hat{\boldsymbol{P}}^{m}(\boldsymbol{k})$ results in the change of the deformation gradient $\Delta\tilde{\boldsymbol{F}}^{m+1}(\boldsymbol{x}) = \Delta\boldsymbol{F}^{m+1}(\boldsymbol{x})$ that leads to mechanical equilibrium. The new average deformation gradient does not necessarily fulfill the applied BCs. The average part $\overline{\boldsymbol{F}}^{m+1}(\boldsymbol{x})$ on each FP is therefore corrected in

a way similar to the one applied in the calculation described in section 7.3.1 during each iteration. With the new deformation gradient $\boldsymbol{F}^{m+1}(\boldsymbol{x})$, a new stress $\boldsymbol{P}^{m+1}(\boldsymbol{x})$ is computed. The loop is performed until convergence is reached within the limits given as tolerance.

The calculation ensures that $\overline{\boldsymbol{F}}$ equals the value predicted in the calculation described in section 7.3.1. However, as the local fluctuations change the stress state, it does not necessarily lead to a state where the stress BCs are fulfilled. The average stress $\overline{\boldsymbol{P}}$ is therefore calculated during each iteration and compared to the applied stress BCs. The BCs are regarded as fulfilled if the largest deviation is less than 1.0% of the amount of largest stress component. This tolerance for the stress BCs is less accurate than the one of the scheme outlined in section 7.3.1. If the stress does not fit to the applied BCs, again a correction of $\overline{\boldsymbol{F}}$ is accomplished as described in section 7.3.1.

The different tolerances lead to a situation, where the calculation loops for global and local deformation gradient are usually employed only once. A switch back from the scheme for the local deformation gradient to the scheme for the global one is rarely needed, mostly for the first step of a load case with high applied deformations.

As soon as the equilibrium state is reached while the stress BCs are fulfilled, the results are written to the output file and the algorithm moves to the next step of the load case (or to the next load case if the calculated step was the last step of the current load case).

7.4 Output

The results obtained by the material subroutine are stored in the output file `results.out` in binary format. The results that can be written out depend on the constitutive model used. They can be specified in the file `material.config`. The current configuration of the VE has to be constructed from the deformation gradient on all FPs $\boldsymbol{F}(\boldsymbol{x})$. Thus, $\boldsymbol{F}(\boldsymbol{x})$ should be always chosen as an output option. The information can be converted to a file that can be viewed in *gmsh*[5].

7.5 Resulting algorithm

The resulting algorithm is implemented in *Fortran* 90/95. The compiled executable file is called *mpie_spectral*. Its source code is divided into two files:

- `mpie_spectral.f90`
- `mpie_spectral_interface.f90`

The two files are given in appendix A. The first file contains the information outlined in this chapter. It is schematically given as pseudocode in listing 7.2. The second file provides the information for `CPFEM_general` how to read in the geometry from the geometry file `*.geom`.

Besides these two source code files, different files containing the source code for the material models, standard mathematical tasks, etc. must be compiled and linked. Since most of these files are not altered for the implementation of the spectral method, they are not included in the

[5] `http://geuz.org/gmsh/`, accessed 14th November 2010.

appendix. Only the code file for the geometry generation (`mesh.f90`) was heavily modified and parts of it are given in listing A.3. Only the generic routines (for both, FEM- and spectral method-based solvers) and the routines especially written for the presented implementation are included in appendix A. The routines for *MSC.Marc* and *Abaqus* as well as the definitions for all elements except from *C3D8R* are removed from the file. To use *FFTW* with multiprocessor support, three files must be added. The compiled library files `libfftw3.a` and `libfftw3_threads.a` are linked to the compiled routines and the file storing the variable definitions, `fftw3.f`, is included in the source code.

The makefile storing the information how to compile and link all necessary files using the *Intel Fortran* compiler[6] is given in listing A.4.

[6] `http://software.intel.com/en-us/articles/intel-composer-xe/`, accessed 14th November 2010.

Listing 7.2: Summarized algorithm

Data: $geometry, material, load\_cases$
Result: $\overline{\mathbb{S}}, \boldsymbol{k}, \hat{\mathbb{\Gamma}}, n\_load\_cases$
for $j \leftarrow 1$ **to** n_load_cases **do** // looping over load cases
  $n\_steps \leftarrow \texttt{Func}(load\_cases(j))$ // with 'Func' denoting a generic function
  for $i \leftarrow 1$ **to** n_steps **do** // looping over steps of current load case
    if $i = 1$ **then** // homogeneous guess for first step
      $\overline{\boldsymbol{F}}^1 \leftarrow \texttt{Func}(load\_cases(j))$
      $\boldsymbol{F}^1(\boldsymbol{x}) \leftarrow \overline{\boldsymbol{F}}$
    else // continue along former trajectory
      $\overline{\boldsymbol{F}}^i \leftarrow (2 \cdot \overline{\boldsymbol{F}}^{i-1} - \overline{\boldsymbol{F}}^{i-2})$
      $\boldsymbol{F}^i(\boldsymbol{x}) \leftarrow (2 \cdot \boldsymbol{F}^{i-1}(\boldsymbol{x}) - \boldsymbol{F}^{i-2}(\boldsymbol{x}))$
    $calcmode \leftarrow 1$
    $m \leftarrow 0$
    while $error\_stress \geq tol$ **or** $error\_divergence \geq tol$ **do** // convergence loop
      switch $calcmode$ **do**
        case *1* // global deformation gradient (fulfill stress BCs)
          $m \leftarrow m + 1$ // increase number of iterations
          $\boldsymbol{P}^m(\boldsymbol{x}) \leftarrow \texttt{Func}(\boldsymbol{F}^{m-1}(\boldsymbol{x}))$ // constitutive law
          $\overline{\boldsymbol{P}}^m \leftarrow |\boldsymbol{P}^m(\boldsymbol{x})|$
          $\overline{\boldsymbol{F}}^m \leftarrow \texttt{Func}(\overline{\boldsymbol{P}}^m, \overline{\mathbb{S}})$ // correct average deformation gradient
          $error\_divergence \leftarrow (2 \cdot tol)$ // equilibrium never fulfilled
          $error\_stress \leftarrow \texttt{Func}(\overline{\boldsymbol{P}}^m, load\_cases(j))$ // compare to BCs
          if $error\_stress < tol$ **then** $calcmode \leftarrow 2$
        case *2* // local deformation gradient (fulfill equilibrium)
          $m \leftarrow m + 1$ // increase number of iterations
          $\boldsymbol{P}^m(\boldsymbol{x}) \leftarrow \texttt{Func}(\boldsymbol{F}^{m-1}(\boldsymbol{x}))$ // constitutive law
          $\hat{\boldsymbol{P}}^m(\boldsymbol{k}) \leftarrow \texttt{FFT}(\boldsymbol{P}^m(\boldsymbol{x}))$ // FT of stress field
          $\Delta\hat{\boldsymbol{F}}^m(\boldsymbol{k}) \leftarrow \hat{\mathbb{\Gamma}}(\boldsymbol{k}) : \hat{\boldsymbol{P}}^m(\boldsymbol{k})$
          $\boldsymbol{F}^m(\boldsymbol{x}) \leftarrow \boldsymbol{F}^{m-1}(\boldsymbol{x}) + \texttt{FFT}^{-1}(\Delta\boldsymbol{F}^m(k))$ // inverse FT
          $\overline{\boldsymbol{P}}^m \leftarrow |\boldsymbol{P}^m(\boldsymbol{x})|$
          $error\_stress \leftarrow \texttt{Func}(\overline{\boldsymbol{P}}^m, load\_cases(j))$
          $error\_divergence \leftarrow \texttt{Func}(\boldsymbol{P}^m(\boldsymbol{k}), \boldsymbol{k})$ // in Fourier space
          if $error\_stress \geq tol$ **then** $calcmode \leftarrow 1$
    Result: $\boldsymbol{P}^i(\boldsymbol{x}), \boldsymbol{F}^i(\boldsymbol{x})$ // store results of converged step

8 Simulation results

In order to test the implemented algorithm, several simulations are performed.Three test and the results achieved are presented in this chapter. The first simulations presented in section 8.1 are used to prove that the implementation works in general.The test conducted to see if large deformations are handled correctly is shown in section 8.2. The comparison between the solution achieved by the spectral method and the "de facto standard" FEM is outlined in section 8.3.

8.1 Proof of correct implementation

As a first step to validate the implementation, some results are compared to simulations carried out with an implementation of the spectral method written by R. LEBENSOHN. This implementation is called *evp5j*. It uses a small strain formulation and a viscoelastic constitutive law without hardening. A comparable constitutive model available in the routines of the *MPIE* is the *phenomenological powerlaw* (chapter 3.4.2) with the hardening set to zero. The elastic constants used are chosen in allusion to the values known for copper. The VE is a polycrystal consisting of 100 randomly orientated grains. It is discretized by 32 FPs in each direction.

A velocity gradient with component $L_{33} = 1\,\mathrm{s}^{-1}$ is prescribed. The global shear deformations are set to zero. The stress of the two remaining directions is set to zero (stress BCs). Thus, the prescribed load case results in a uniaxial stress state. The deformation is applied in 40 steps, each with a duration of $0.0001\,\mathrm{s}$, resulting in a final strain of $\varepsilon_{33} = 0.0004$. The load case is given in the first line of listing B.2.

Three different versions, called *mpie_spectral* v. 0.1, v. 0.2, and v. 0.3 are used for comparison. Version 0.1 of *mpie_spectral* differs only slightly from *evp5j* if a similar material model is used. The most serious differences are the prediction of the fluctuations and the calculation of the error. While in *evp5j* a completely homogeneous strain field $\boldsymbol{\varepsilon}^1(\boldsymbol{y}) = \overline{\boldsymbol{\varepsilon}}$ is the prediction at the first iteration of each step, in *mpie_spectral* v. 0.1 the fluctuations of the last step are stored and only the additionally prescribed deformation is homogeneous: $\boldsymbol{\varepsilon}^1(\boldsymbol{y}) = \boldsymbol{\varepsilon}^0(\boldsymbol{y}) + \overline{\boldsymbol{\varepsilon}} - \overline{\boldsymbol{\varepsilon}}^0$. Superscript 0 denotes the deformation at the end of the last step, i.e. no deformation at the first step. The abort criterion implemented in *evp5j* depends on the relative change of the deformation compared to the last iteration. The criterion implemented in *mpie_spectral* v. 0.1 is based on the divergence of the fluctuation field that is calculated in FOURIER space. The field is said to be divergence free, thus in the mechanical equilibrium, if

$$\max_{\boldsymbol{k}} \frac{|\boldsymbol{k} \cdot \hat{\boldsymbol{P}}(\boldsymbol{k})|}{|\hat{\boldsymbol{P}}(\boldsymbol{0})|} \leq a_{\mathrm{tol}} = 10^{-5} \tag{8.1}$$

This criterion is more accurate than the value of $a_{\mathrm{tol}} = 10^{-4}$ proposed in chapter 6.

Version 0.1 of *mpie_spectral* is not optimized. It uses complex-to-complex (c2c) FFTs for the FT and the inverse FT and makes no educated guess at the displacement. Versions 0.2 and 0.3 are optimized regarding theses points. The difference between versions 0.2 and 0.3 is the handling of the deformation gradient. As in *evp5j*, in version 0.2 the deformation gradient is symmetrized. Thus, the strain measure is the CAUCHY strain that is valid only for the small strain formulation. Version 0.3 of *mpie_spectral* does not symmetrize the deformation gradient. It is the final version, fully written in terms of the large strain framework introduced in chapter 2.

In tab. 8.1 the properties of the different versions of *mpie_spectral* are compared to *evp5j*. The iterations needed to converge for some steps of the applied load case are also given in this table.

Table 8.1: Properties of the different versions of *mpie_spectral* compared to *evp5j*

implementation		*evp5j*	*mpie_spectral*		
version		n/a	0.1	0.2	0.3
type of the FFT		c2c, c2c		r2c, c2r	
mech. framework		small strain		large strain	
deformation			symmetric		non-symmetric
predicted deformation		fully homog.	new homog. step	continuation at last rate	
iterations needed to converge	step 1:	14	32	23	23
	step 2:	16	21	2	2
	step 3:	16	24	2	2
	step 4:	17	18	2	2
	step 5:	17	17	2	2
	step 10:	17	13	2	2
	step 15:	20	12	4	4
	step 20:	20	13	5	5
	step 25:	21	16	4	4
	step 30:	21	16	3	3
	step 35:	21	17	3	3
	step 40:	21	17	2	2

It can be seen from tab. 8.1 that for the first step, more than twice the number of iterations are needed by *mpie_spectral* v. 0.1 when compared to *evp5j*. This is a result of the different abort criteria. By using the less strict criterion $a_{\mathrm{tol}} = 10^{-4}$ for *mpie_spectral*, fewer iterations compared to *evp5j* are needed, but the results differ significantly. For the following steps, fewer iterations are needed due to the better guess at the beginning of each step (homogeneous add-on instead of fully homogeneous deformation).

At the first step, the prediction is the same for all implementations. That fewer guesses are needed in version 0.2 and 0.3 compared to v. 0.1 can be explained by the use of the r2c/c2r interfaces. As the r2c/c2r FFT takes the not existing imaginary part of the quantities in real space into account, the numerical distortion is much lower. This results in a faster convergence.

Even with the homogeneous add-on, a faster convergence is reached compared to the reference implementation. With the prediction of the former rate as a guess (*mpie_spectral* v. 0.2 and v.

0.3) for the new step the performance is even significantly better.

The stress–strain curves computed by the different version of *mpie_spectral* are identical. In fig. 8.1 the stress–strain curve of *mpie_spectral* v. 0.3 is shown as an example and compared to the one computed by *evp5j*. The small observable deviations can be explained by the slightly different constitutive laws and by numerical distortions. Also, the small strain formulation in *evp5j* does not use a proper velocity gradient but an approximation valid only for small strains. This can explain that the curves differ more for higher strain.

The VON MISES equivalent of the displacement on one side of the VE is shown in figures 8.2 and 8.3. Fig. 8.2 shows the displacement field after the first step ($\varepsilon_{33} = 0.00001$). Fig. 8.3 shows it after the last step ($\varepsilon_{33} = 0.0004$). The fields are shown in undistorted configuration, each pixel corresponds to one FP. The deformation direction is denoted by arrows on the edges of the sketched VE in the legend. The perspective of the given view is outlined by double lines in the legend.

The displacement field computed after the first and last step of the example load case does not appreciably differ between *evp5j* and *mpie_spectral* v. 0.1 and v. 0.2. As expected, the displacement computed by *mpie_spectral* v. 0.3 is slightly different. The reason is that in this version the deformation gradient is not symmetrized.

The VON MISES equivalent of the CAUCHY stress tensor after the first and the last step is shown in figures 8.4 and 8.4. The stress fields computed by *evp5j*, *mpie_spectral* v. 0.1, and v. 0.2 differ slightly. The deviations can only be seen if they are visualized using image processing software. There is no difference between the results achieved by *mpie_spectral* v. 0.2 and v. 0.3. This can be explained by the fact that the symmetrization of the deformation gradient results only in a rotated stress tensor. The rotation does not affect the VON MISES equivalent of the tensor.

In summary, the results show that the integration of the spectral method into the existing material subroutines of the *MPIE* is done in such a way that the results do not significantly differ those results computed with the stand-alone version written by R. LEBENSOHN.

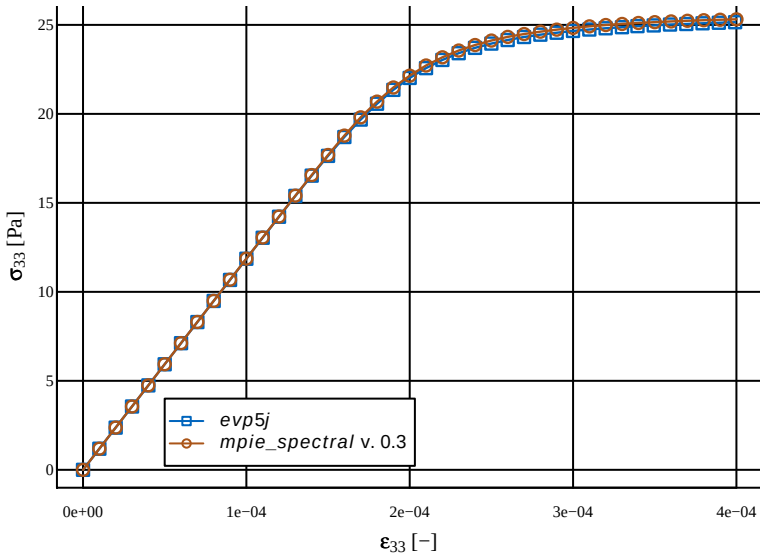
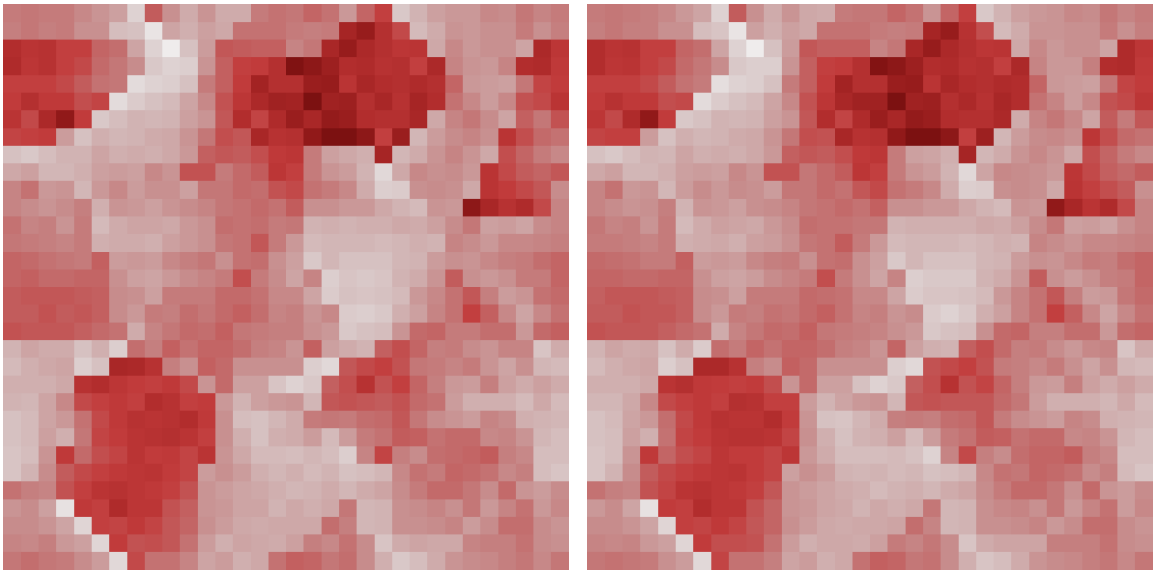
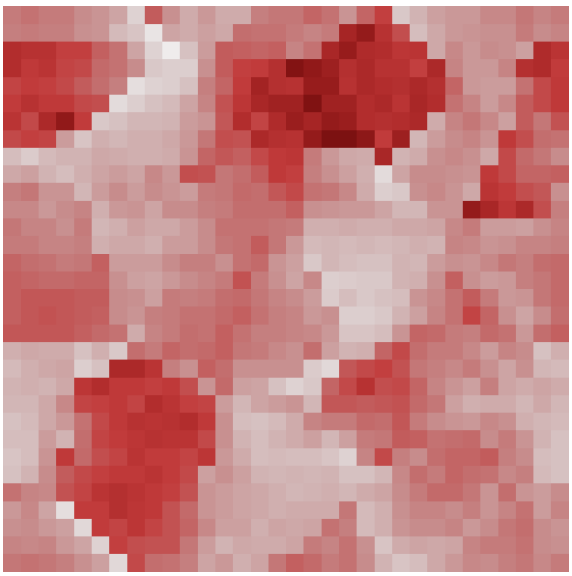


Figure 8.1: Stress–strain curves of *evp5j* and *mpie_spectral* v. 0.3

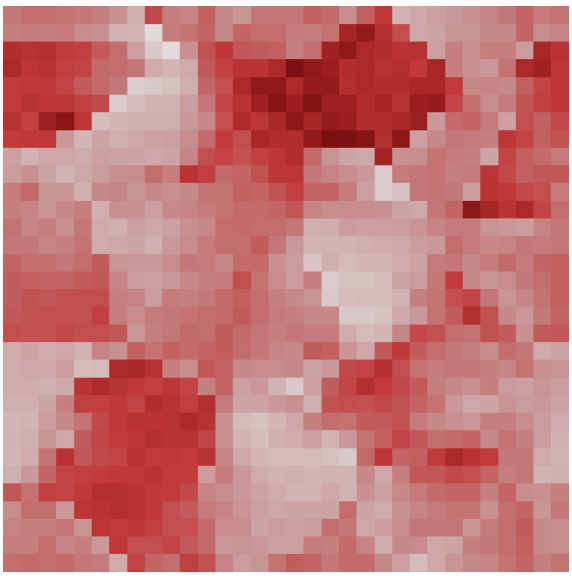


(a) *evp5j*

(b) *mpie_spectral* v 0.1

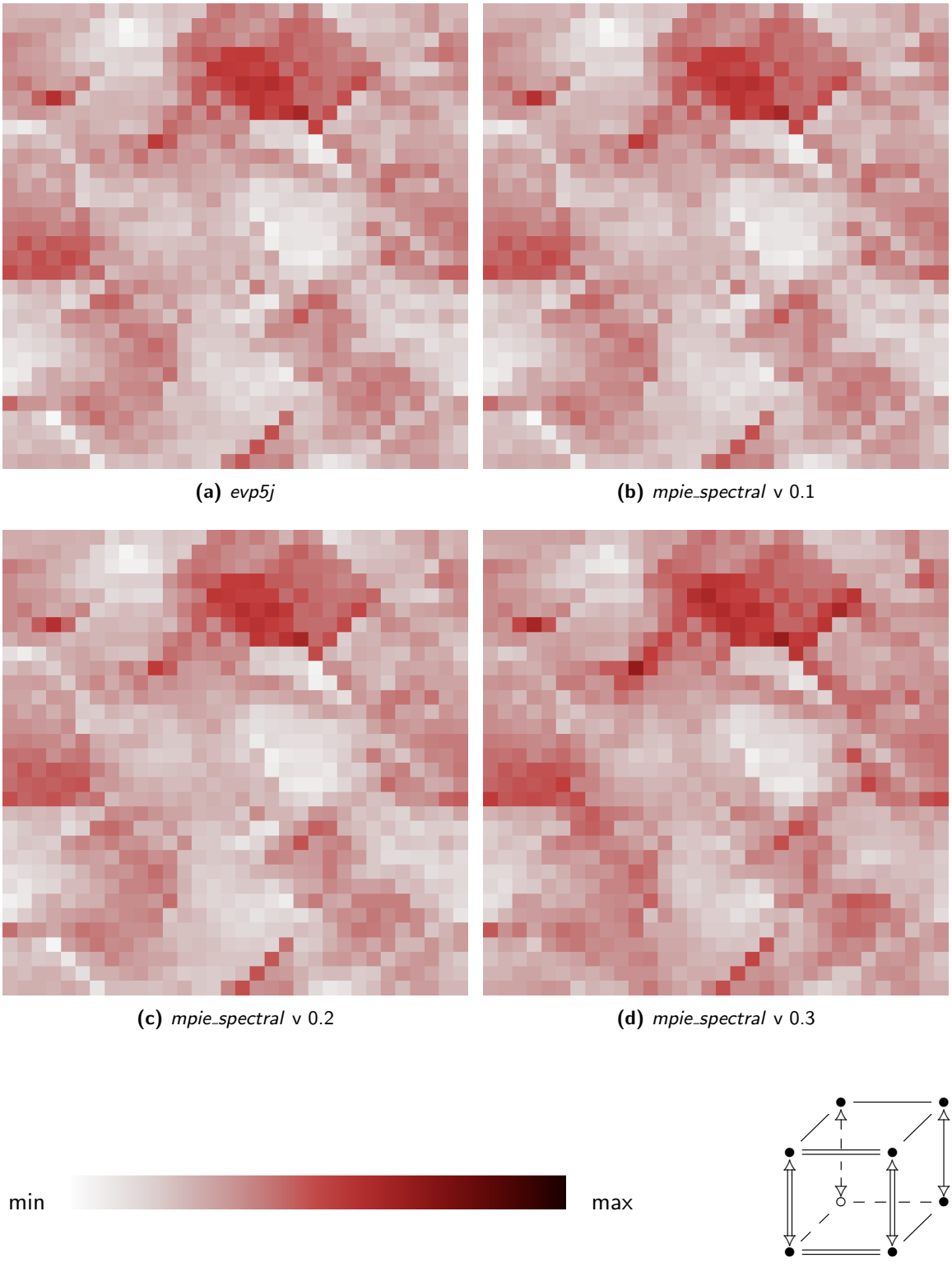


(c) *mpie_spectral* v 0.2



(d) *mpie_spectral* v 0.3

Figure 8.2: Displacement field $H_{0,\mathrm{vM}}$ at $\varepsilon_{33} = 0.00001$



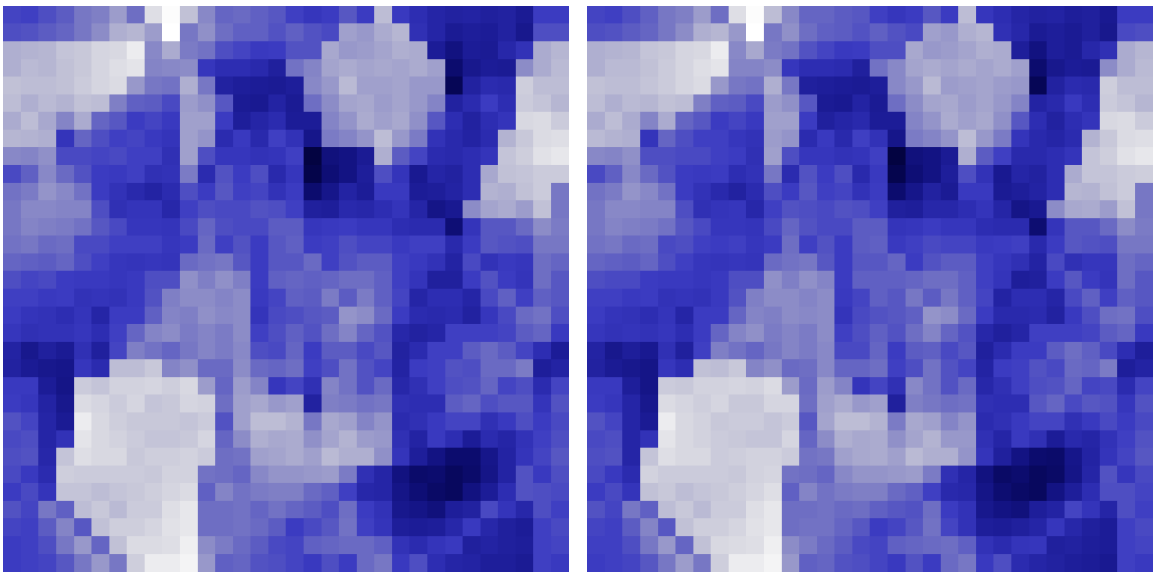
(a) *evp5j*

(b) *mpie_spectral* v 0.1

(c) *mpie_spectral* v 0.2

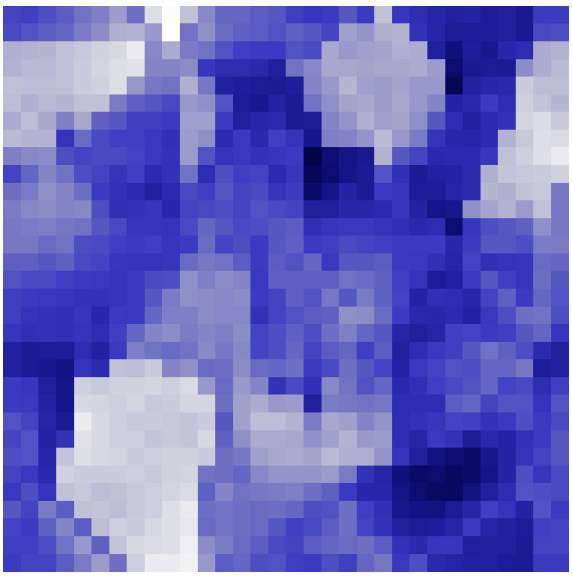
(d) *mpie_spectral* v 0.3

Figure 8.3: Displacement field $H_{0,\mathrm{vM}}$ at $\varepsilon_{33} = 0.0004$

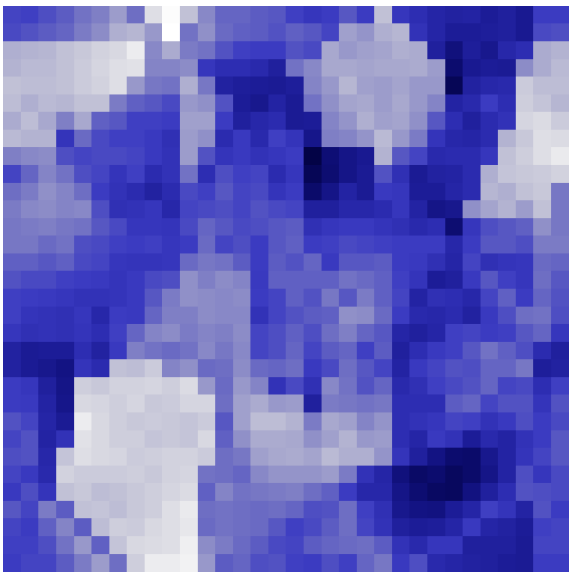


(a) *evp5j*

(b) *mpie_spectral* v 0.1



(c) *mpie_spectral* v 0.2



(d) *mpie_spectral* v 0.3

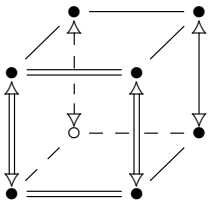
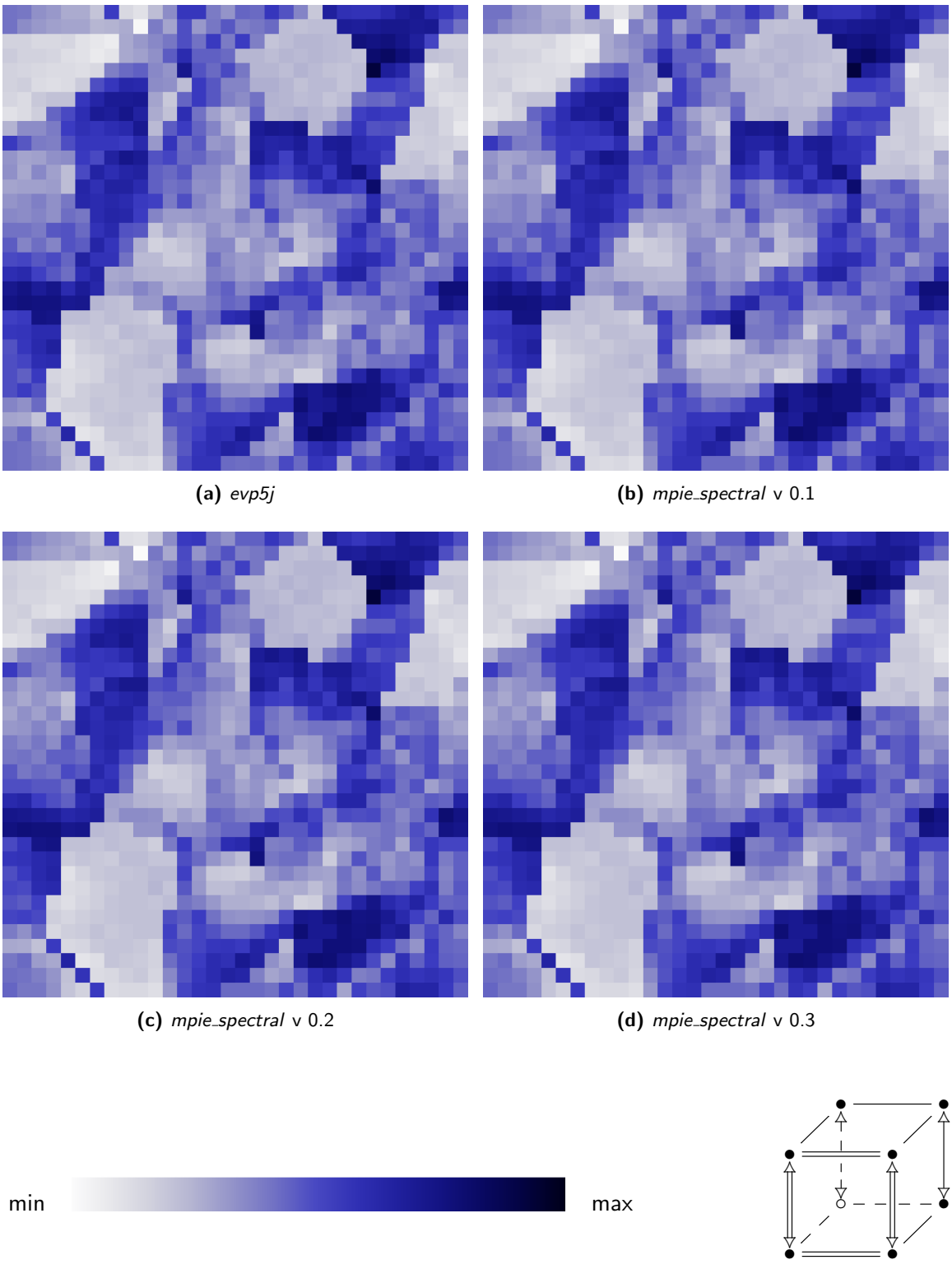


Figure 8.4: Stress field σ_{vM} at $\varepsilon_{33} = 0.00001$



(a) *evp5j* **(b)** *mpie_spectral* v 0.1

(c) *mpie_spectral* v 0.2 **(d)** *mpie_spectral* v 0.3

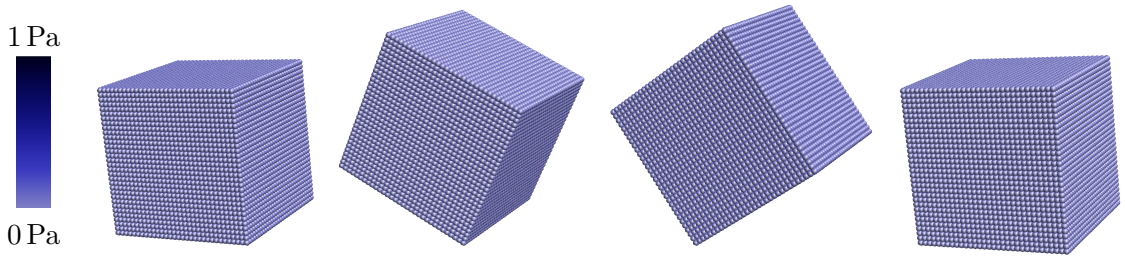
Figure 8.5: Stress field σ_{vM} at $\varepsilon_{33} = 0.0004$

8.2 Handling of large deformations

When using the large strain formulation, rotations must not result in strain or stress. A load case consisting of a pure rotation is therefore used as a test to see if the implementation handles large deformations correctly.

The velocity gradient applied has the form of a rotation matrix with the identity subtracted. The load case results in a rotation through the first axis, thus the polar decomposition reads as: $\boldsymbol{F} = \boldsymbol{R} \cdot \boldsymbol{I} = \boldsymbol{I} \cdot \boldsymbol{R}$, i.e. $\boldsymbol{V} = \boldsymbol{I}$ and $\boldsymbol{U} = \boldsymbol{I}$. The VE rotates through $90\,^\circ$ in 90 steps, i.e. in each step it rotates through $1\,^\circ$. Since all components of the velocity gradient are defined, no stress BCs are applied. The load case is given in the second line of listing B.2[1].

The VE used for the test is a single crystal. The constitutive model is the isotropic *J_2-plasticity* model that is outlined in chapter 3.4.1. The parameters are fitted in order to simulate the behavior of an aluminum alloy. The VE is shown at different rotation angles in fig. 8.6.



(a) $1\,^\circ$ **(b)** $30\,^\circ$ **(c)** $60\,^\circ$ **(d)** $90\,^\circ$

Figure 8.6: Stress field σ_{vM} resulting from pure rotation

From fig. 8.6 it can be seen that the stress is almost zero during the rotation. The small amount of stress remains constant during the rotation. If the approximation used for the values of the sines and cosines of the applied rotation is less accurate, the stress is much higher. Thus, the stress can be seen as a result of numerical inaccuracies. The test proves that the implementation handles large deformations correctly, i.e. only the stretch and not the rotational part of the applied deformation gradient results in stress.

8.3 Comparison with FEM solutions

The close integration of the spectral method into the existing framework for CPFEM allows it to compute the behavior of the same VE using the spectral method and the "de facto standard" FEM. A comparison between the solution achieved by the commercial FEM solver *MSC.Marc* 2010 and the solution computed by the spectral method is conducted to see how the results differ between the FEM and the spectral method. The period of time needed for the calculations is used to estimate the performance of the spectral method compared to the FEM. Two load cases are used for the comparison, applying plane strain (section 8.3.1) and uniaxial stress (section 8.3.2).

[1]The given values are truncated at seven decimal places. For the simulation, the applied values are three times more accurate.

The VE used for both simulations is a polycrystal consisting of 100 grains. It is discretized by 32 FPs in each direction. The geometry file describing the VE used for the spectral method is converted to a file format readable by the FEM-based solver. The hexaheral element that is pretended by the spectral method is also used for the solution by means of the FEM. It is a linear element with reduced integration capacity. The material model used is the *phenomenological powerlaw* (chapter 3.4.2) with parameters set to the values known for an exemplary aluminum alloy.

8.3.1 Plane strain

The load case used for the plane strain test consists of the stress BC $P_{22} = 0$ and a velocity gradient with $L_{33} = -0.001\,\mathrm{s}^{-1}$. All remaining components of $\boldsymbol{L}$ are set to zero. The load case is schematically shown in the legend of fig. 8.7 with a vertical bar denoting no strain in the respective direction. It description file is given in the third line of listing B.2.

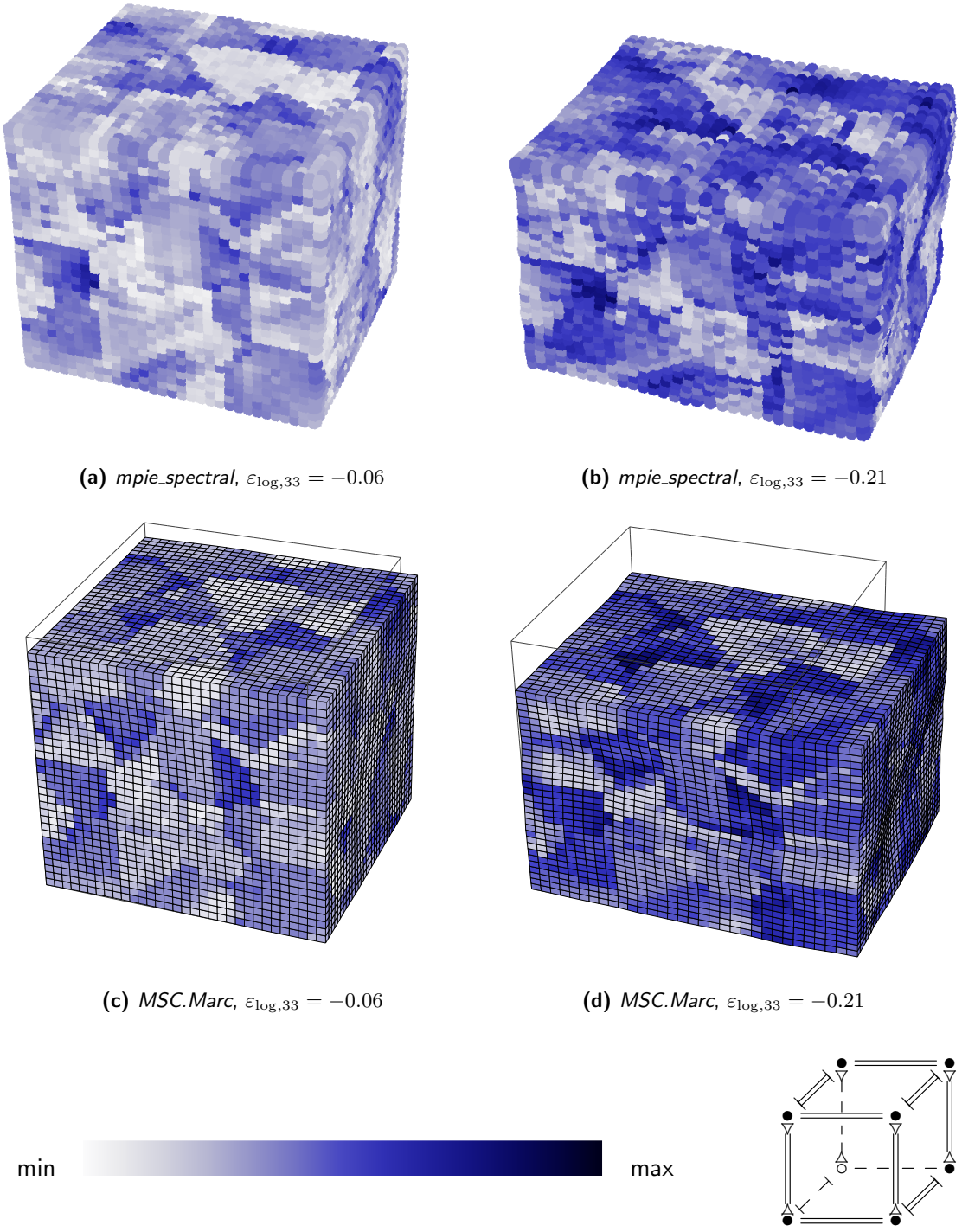
The resulting VON MISES equivalent of the CAUCHY stress at a strain of $\varepsilon_{\log,33} = -0.06$ (step 62) and $\varepsilon_{\log,33} = -0.21$ (step 208) is given in fig. 8.7. It can be seen that the solutions achieved by both solvers show good correlation. Two small differences are obvious: the stress computed by the spectral method is lower in general and its spatial distribution is more inhomogeneous. Unfortunately, the available postprocessing facilities are likely to amplify the effects. The black lines denoting the borders of the elements contribute to the darker appearance of the stress field computed by the FEM solver[2]. The postprocessing tool of *MSC.Marc* uses 30 discrete values for the coloration while *gmsh* uses a continuous map. This slightly contributes to the more inhomogeneous appearance of the stress field calculated by means of the spectral method.

The period of time needed for the spectral method to compute the deformation at maximum strain is approximately $4\,\mathrm{h}$. The period of the simulation is heavily determined by the calculation of the constitutive law. The FFT and the multiplication in FOURIER space, i.e. the actual spectral method, take only a fraction of the whole runtime. The solution by means of the FEM takes about $6\,\mathrm{d}$. Thus, the spectral method has shown a much better performance compared to the FEM.

Carrying out the simulation with the abort criterion $a_{\mathrm{tol}} = 10^{-5}$ instead of the standard value $a_{\mathrm{tol}} = 10^{-4}$ does not lead to significantly different results. Instead of needing an average of 2-3 iterations for each step, approximately 25 iterations are needed to achieve the higher accuracy. This results in a time period for calculations that is about ten times higher than for the usual tolerance.

Applying the load case works only up to a strain of approximately $\varepsilon_{\log,33} = -0.30$. It turns out that the convergence at high deformation states is not ensured. Until now it was not possible to undoubtedly identify the reason for this behavior. As the maximum strain at which the solution converges depend on the parameters of the selected VE, i.e. the phase contrast, this problem might be related to the fact that the spectral method does not converge for high phase contrasts. Two possible solutions to overcome this problem are outlined in chapter 9.

[2] This effect can be seen clearly if the digital version of this thesis is viewed at a small scale.



(a) *mpie_spectral*, $\varepsilon_{\log,33} = -0.06$

(b) *mpie_spectral*, $\varepsilon_{\log,33} = -0.21$

(c) *MSC.Marc*, $\varepsilon_{\log,33} = -0.06$

(d) *MSC.Marc*, $\varepsilon_{\log,33} = -0.21$

Figure 8.7: Stress field σ_{vM} resulting from plane strain

8.3.2 Uniaxial tension

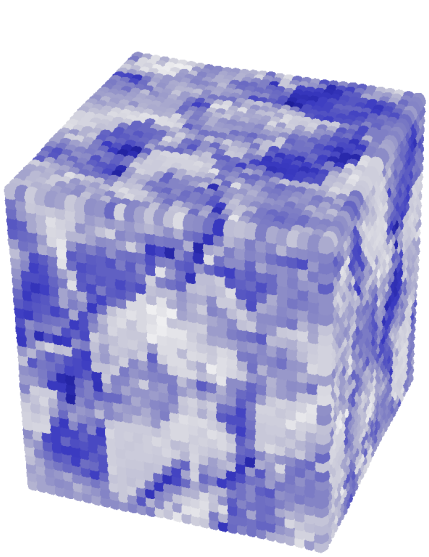
The second simulation that is compared to an FEM solution is a "tensile test". Because the implementation presented here is not able to handle arbitrary phase contrasts, it is not possible to model a layer of "air" (i.e. a material with $E = 0$) around the sample. Thus, necking phenomena that occur in real tensile tests cannot be examined.

The load case is similar to the one used for the comparison between *mpie_spectral* and *evp5j*. The VE deforms at a strain rate of $\dot{\varepsilon}_{33} = 0.001\, s^{-1}$. The load case is given in the last line of listing B.2.

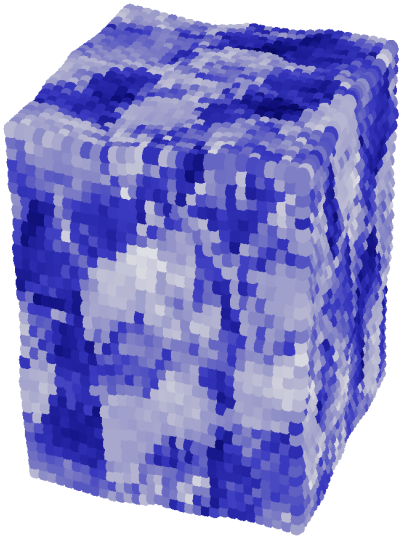
As for the plane strain load case, the resulting VON MISES equivalent of the CAUCHY stress is used for comparison. In fig. 8.8 the stress field is shown a strain of $\varepsilon_{\log,33} = 0.06$ (step 62) and $\varepsilon_{\log,33} = 0.21$ (step 208). From fig. 8.8 the same conclusions can be drawn as from fig. 8.8: the results show good correlation, but the stress computed by the spectral method in general is slightly lower and its spatial distribution is more inhomogeneous.

The performance of the spectral method is again between one and two orders of magnitude better compared to FEM.

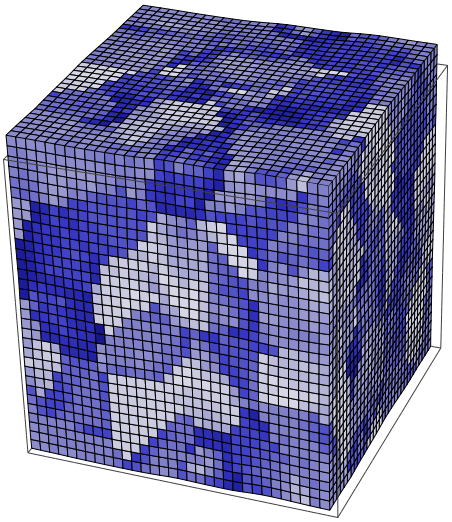
Again, convergence is not ensured if the deformation reaches a strain larger then $\varepsilon_{ij} \approx 0.30$. To check whether the discretization has an influence on the convergence, a VE consisting of 100 grains but discretized by 64 FPs in each direction is simulated. With this VE, the convergence is still not ensured at large strains.



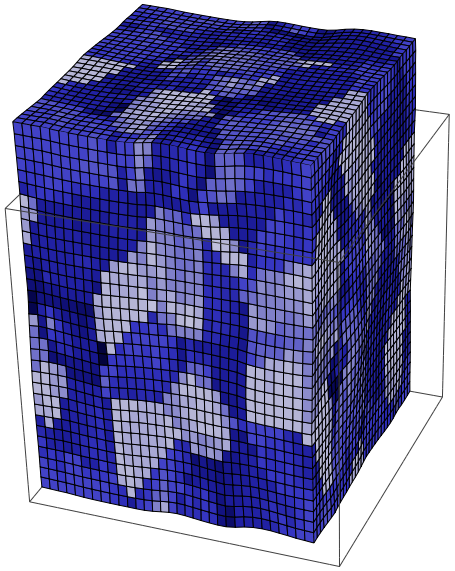
(a) *mpie_spectral*, $\varepsilon_{\log,33} = 0.06$



(b) *mpie_spectral*, $\varepsilon_{\log,33} = 0.21$



(c) *MSC.Marc*, $\varepsilon_{\log,33} = 0.06$



(d) *MSC.Marc*, $\varepsilon_{\log,33} = 0.21$

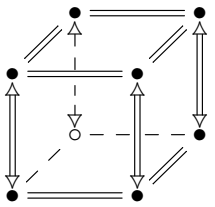


Figure 8.8: Stress field σ_{vM} resulting from uniaxial tension

9 Conclusions and outlook

This thesis discusses the implementation of a spectral method as an alternative solver into an existing framework for crystal plasticity. The framework in combination with the spectral method is used to examine the mechanical properties of materials by calculating the response of an RVE. The spectral method implemented has been proven to be a very suitable tool for the solution of elastoviscoplastic boundary value problems with periodic boundary conditions describing an RVE. The results achieved compare favorably with the solutions obtained by FEM, while the performance of the spectral method is much better compared to commercial FEM-based solvers. For the presented simulations, the spectral method is between one and two orders of magnitude faster than the FEM-based solver used for comparison. Thus, the spectral method is an excellent alternative for the examination of RVEs.

To further validate the implementation, more simulations should be carried out. For the comparison with FEM solutions more sophisticated postprocessing tools should be implemented to compare the various results obtained by the material subroutines in detail. Since there are deviations compared to solutions achieved by the FEM, not only comparisons to FEM simulations should be conducted. It would be also important to compare the resulting stress and strain fields to experimental data. Reading in data obtained by experiments is easy due to the simple data format used for the geometry description. However, so far it is not possible to use data from a pre-stressed state since the initial configuration is always stress free. An extended interface could be implemented to read in information about the stress state on each point to continue calculations from a pre-stressed configuration.

As outlined in chapter 8, convergence is not achieved for highly deformed configurations. The reason for this behavior is not identified. It is possible that it is related to the fact that the spectral method converges slowly for materials with high phase contrasts and does not converge at all for an infinite phase contrast. Two techniques are suggested to ensure the convergence for materials with infinite contrasted phases and speed up the rate of convergence in general. The two methods take totally different approaches. The first approach is the augmented Lagrange method proposed by J. C. Michel, H. Moulinec and P. Suquet in 2001. The second improvement is based on the use of a modified Γ-operator. It is proposed by S. Brisard and L. Dormieux in 2010. It is also possible to implement both methods at the same time, probably resulting in an even faster convergence.

The augmented Lagrange method is described in [17, 20]. It uses a modified algorithm consisting of three steps to solve a saddle-point problem describing the mechanical state of the VE. The first step consists in solving a linear elastic problem, using a similar method to the one presented in this thesis. An additional second step is required to solve a non-linear problem at each

point in the VE. In the third step the LAGRANGE multiplier that is used to solve the non-linear problem is updated.

Although their aim is the same, i.e. speeding up the calculation and ensure convergence at infinite phase contrasts, S. BRISARD and L. DORMIEUX take a different approach [5]. An optimized Γ–operator is used in an algorithm as simple as the scheme presented in this work. It is derived from an energy principle, not from the LIPPMANN–SCHWINGER equation that is the basis for the algorithm presented in chapter 6.2.

Both methods are likely to speed up the calculations, even if the LAGRANGE method takes longer for one iteration. But as fewer iterations are needed for convergence, the total runtime will decrease.

A Sourcecode

Listing A.1: `mpie_spectral.f90`

```
!* $Id: mpie_spectral.f90 683 2010-10-27 17:15:49Z MPIE\m.diehl $
!**************************************************************************
! Material subroutine for BVP solution using spectral method
!
! written by P. Eisenlohr,
!            F. Roters,
!            L. Hantcherli,
!            W.A. Counts
!            D.D. Tjahjanto
!            C. Kords
!            M. Diehl
!            R. Lebensohn
!
! MPI fuer Eisenforschung, Duesseldorf
!
!**************************************************************************
!     Usage:
!             - start program with mpie_spectral PathToGeomFile/NameOfGeom.geom
!               PathToLoadFile/NameOfLoadFile.load
!             - PathToLoadFile will be the working directory
!             - make sure the file "material.config" exists in the working
!               directory
!**************************************************************************
program mpie_spectral
!**************************************************************************

 use mpie_interface
 use prec, only: pInt, pReal
 use IO
 use math
 use CPFEM, only: CPFEM_general
 use numerics, only: err_div_tol, err_defgrad_tol, err_stress_tolrel, itmax
 use homogenization, only: materialpoint_sizeResults, materialpoint_results

 implicit none
 include 'fftw3.f' !header file for fftw3 (declaring variables). Library file is also needed

! variables to read from loadcase and geom file
 real(pReal), dimension(9) ::                    valuevector          ! stores information from loadcase file
 integer(pInt), parameter ::                     maxNchunksInput=24   ! 4 identifiers, 2 3x3 matrices, and 2 scalars
 integer(pInt), dimension (1+maxNchunksInput*2)::posInput
 integer(pInt), parameter ::                     maxNchunksGeom = 7   ! 4 identifiers, 3 values
 integer(pInt), dimension (1+2*maxNchunksGeom):: posGeom
 integer(pInt) unit, N_l, N_s, N_t, N_n                               ! numbers of identifiers
 character(len=1024) path, line
 logical gotResolution,gotDimension,gotHomogenization
 logical, dimension(9) :: bc_maskvector

! variables storing information from loadcase file
 real(pReal)                                     timeinc
 real(pReal), dimension (:,:,:), allocatable ::  bc_velocityGrad, &
                                                 bc_stress            ! velocity gradient and stress BC
 real(pReal), dimension(:), allocatable ::       bc_timeIncrement     ! length of increment
 integer(pInt)                                   N_Loadcases, steps
 integer(pInt), dimension(:), allocatable ::     bc_steps             ! number of steps
 logical, dimension(:,:,:), allocatable ::       bc_mask              ! mask of boundary conditions

! variables storing information from geom file
 real(pReal) wgt
 real(pReal), dimension(3) ::  geomdimension
 integer(pInt) homog, prodnn
 integer(pInt), dimension(3) :: resolution
```

```
  ! stress etc.
   real(pReal), dimension(3,3) ::                    ones, zeroes, temp33_Real, damper,&
                                                     pstress, pstress_av, cstress_av, defgrad_av,&
                                                     defgradAim, defgradAimOld, defgradAimCorr,&
                                                     defgradAimCorrPrev, mask_stress, mask_defgrad
   real(pReal), dimension(3,3,3) ::                  temp333_Real
   real(pReal), dimension(3,3,3,3) ::                dPdF, c0, s0
   real(pReal), dimension(6) ::                      cstress        ! cauchy stress in Mandel notation
   real(pReal), dimension(6,6) ::                    dsde, c066, s066
   real(pReal), dimension(:,:,:), allocatable ::     ddefgrad
   real(pReal), dimension(:,:,:,:,:), allocatable :: pstress_field, defgrad, defgradold, cstress_field

  ! variables storing information for spectral method
   complex(pReal), dimension(:,:,:,:,:), allocatable :: workfft
   complex(pReal), dimension(3,3) ::                 temp33_Complex
   real(pReal), dimension(3,3) ::                    xinormdyad
   real(pReal), dimension(:,:,:,:,:,:,:), allocatable:: gamma_hat
   real(pReal), dimension(:,:,:), allocatable ::     xi
   integer(pInt), dimension(3) ::                    k_s
   integer*8, dimension(2,3,3) ::                    plan_fft

  ! convergence etc.
   real(pReal) err_div, err_stress, err_defgrad, err_div_temp, err_stress_tol, sigma0
   integer(pInt) ierr
   logical errmatinv

  ! loop variables etc.
   real(pReal) guessmode                             ! flip-flop to guess defgrad fluctuation field evolution
   integer(pInt)  i, j, k, l, m, n, p
   integer(pInt)  loadcase, ielem, iter, calcmode, CPFEM_mode

   real(pReal) temperature                           ! not used, but needed for call to CPFEM_general

  !Initializing
   bc_maskvector = ''
   unit = 234_pInt

   ones = 1.0_pReal; zeroes = 0.0_pReal

   N_l = 0_pInt; N_s = 0_pInt
   N_t = 0_pInt; N_n = 0_pInt

   resolution = 1_pInt; geomdimension = 0.0_pReal

   temperature = 300.0_pReal

   gotResolution =.false.; gotDimension =.false.; gotHomogenization = .false.

   if (IargC() /= 2) call IO_error(102)              ! check for correct number of given arguments

  ! Reading the loadcase file and assign variables
   path = getLoadcaseName()
   print*,'Loadcase: ',trim(path)
   print*,'Workingdir: ',trim(getSolverWorkingDirectoryName())

   if (.not. IO_open_file(unit,path)) call IO_error(45,ext_msg = path)

   rewind(unit)
   do
     read(unit,'(a1024)',END = 101) line
     if (IO_isBlank(line)) cycle                     ! skip empty lines
     posInput = IO_stringPos(line,maxNchunksInput)
     do i = 1, maxNchunksInput, 1
         select case (IO_lc(IO_stringValue(line,posInput,i)))
              case('l','velocitygrad')
                   N_l = N_l+1
              case('s','stress')
                   N_s = N_s+1
              case('t','time','delta')
                   N_t = N_t+1
              case('n','incs','increments','steps')
                   N_n = N_n+1
          end select
     enddo                                           ! count all identifiers to allocate memory and do sanity check
     if((N_l /= N_s).or.(N_s /= N_t).or.(N_t /= N_n))& ! sanity check
       call IO_error(46,ext_msg = path)              ! error message for incomplete input file
   enddo

 101 N_Loadcases = N_l
```

```
! allocate memory depending on lines in input file
 allocate (bc_velocityGrad(3,3,N_Loadcases));        bc_velocityGrad = 0.0_pReal
 allocate (bc_stress(3,3,N_Loadcases));              bc_stress = 0.0_pReal
 allocate (bc_mask(3,3,2,N_Loadcases));              bc_mask = .false.
 allocate (bc_timeIncrement(N_Loadcases));           bc_timeIncrement = 0.0_pReal
 allocate (bc_steps(N_Loadcases));                   bc_steps = 0_pInt

150 rewind(unit)
 i = 0_pInt
 do
   read(unit,'(a1024)',END = 200) line
   if (IO_isBlank(line)) cycle                              ! skip empty lines
   i = i + 1
   posInput = IO_stringPos(line,maxNchunksInput)
   do j = 1,maxNchunksInput,2
     select case (IO_lc(IO_stringValue(line,posInput,j)))
       case('l','velocitygrad')
160      valuevector = 0.0_pReal
         forall (k = 1:9) bc_maskvector(k) = IO_stringValue(line,posInput,j+k) /= '#'
         do k = 1,9                                         ! assign values for the velocity gradient matrix
           if (bc_maskvector(k)) valuevector(k) = IO_floatValue(line,posInput,j+k)
         enddo
         bc_mask(:,:,1,i) = reshape(bc_maskvector,(/3,3/))
         bc_velocityGrad(:,:,i) = reshape(valuevector,(/3,3/))
       case('s','stress')
         valuevector = 0.0_pReal
         forall (k = 1:9) bc_maskvector(k) = IO_stringValue(line,posInput,j+k) /= '#'
170      do k = 1,9                                         ! assign values for the bc_stress matrix
           if (bc_maskvector(k)) valuevector(k) = IO_floatValue(line,posInput,j+k)
         enddo
         bc_mask(:,:,2,i) = reshape(bc_maskvector,(/3,3/))
         bc_stress(:,:,i) = reshape(valuevector,(/3,3/))
       case('t','time','delta')                                 ! increment time
           bc_timeIncrement(i) = IO_floatValue(line,posInput,j+1)
       case('n','incs','increments','steps')                    ! bc_steps
           bc_steps(i) = IO_intValue(line,posInput,j+1)
     end select
180  enddo; enddo
200 close(unit)

 do i = 1, N_Loadcases
   if (any(bc_mask(:,:,1,i) == bc_mask(:,:,2,i))) call IO_error(47,i)      ! bc_mask consistency
   print '(a,/,3(3(f12.6,x)/))','L',bc_velocityGrad(:,:,i)
   print '(a,/,3(3(f12.6,x)/))','bc_stress',bc_stress(:,:,i)
   print '(a,/,3(3(l,x)/))','bc_mask_for_velocitygrad',bc_mask(:,:,1,i)
   print '(a,/,3(3(l,x)/))','bc_mask_for_stress',bc_mask(:,:,2,i)
   print *,'time',bc_timeIncrement(i)
190  print *,'incs',bc_steps(i)
   print *, ''
 enddo

!read header of geom file to get the information needed before the complete geom file is intepretated by mesh.f90
 path = getSolverJobName()
 print*,'JobName: ',trim(path)
 if (.not. IO_open_file(unit,trim(path)//InputFileExtension)) call IO_error(101,ext_msg = path)

 rewind(unit)
200 do
   read(unit,'(a1024)',END = 100) line
   if (IO_isBlank(line)) cycle                              ! skip empty lines
   posGeom = IO_stringPos(line,maxNchunksGeom)

   select case ( IO_lc(IO_StringValue(line,posGeom,1)) )
     case ('dimension')
       gotDimension = .true.
       do i = 2,6,2
         select case (IO_lc(IO_stringValue(line,posGeom,i)))
210        case('x')
             geomdimension(1) = IO_floatValue(line,posGeom,i+1)
           case('y')
             geomdimension(2) = IO_floatValue(line,posGeom,i+1)
           case('z')
             geomdimension(3) = IO_floatValue(line,posGeom,i+1)
         end select
       enddo
     case ('homogenization')
       gotHomogenization = .true.
220    homog = IO_intValue(line,posGeom,2)
     case ('resolution')
       gotResolution = .true.
       do i = 2,6,2
```

```
            select case (IO_lc(IO_stringValue(line,posGeom,i)))
              case('a')
                resolution(1) = IO_intValue(line,posGeom,i+1)
              case('b')
                resolution(2) = IO_intValue(line,posGeom,i+1)
              case('c')
                resolution(3) = IO_intValue(line,posGeom,i+1)
            end select
          enddo
        end select
        if (gotDimension .and. gotHomogenization .and. gotResolution) exit
      enddo
      100 close(unit)

      print '(a,/,i4,i4,i4)','resolution a b c', resolution
      print '(a,/,f6.1,f6.1,f6.1)','dimension x y z', geomdimension
      print *,'homogenization',homog

      allocate (workfft(resolution(1)/2+1,resolution(2),resolution(3),3,3));          workfft             = 0.0_pReal
      allocate (gamma_hat(resolution(1)/2+1,resolution(2),resolution(3),3,3,3,3));    gamma_hat           = 0.0_pReal
      allocate (xi(resolution(1)/2+1,resolution(2),resolution(3),3));                 xi                  = 0.0_pReal
      allocate (pstress_field(resolution(1),resolution(2),resolution(3),3,3));        pstress_field       = 0.0_pReal
      allocate (cstress_field(resolution(1),resolution(2),resolution(3),3,3));        cstress_field       = 0.0_pReal
      allocate (defgrad(resolution(1),resolution(2),resolution(3),3,3));              defgrad             = 0.0_pReal
      allocate (defgradold(resolution(1),resolution(2),resolution(3),3,3));           defgradold          = 0.0_pReal
      allocate (ddefgrad(resolution(1),resolution(2),resolution(3)));                ddefgrad            = 0.0_pReal
      allocate (displacement(resolution(1),resolution(2),resolution(3),3));           displacement        = 0.0_pReal

    ! Initialization of fftw (see manual on fftw.org for more details)
      call dfftw_init_threads(ierr)
      call dfftw_plan_with_nthreads(12)                                   ! for machine with 12 cores

      do m = 1,3; do n = 1,3
        call dfftw_plan_dft_r2c_3d(plan_fft(1,m,n),resolution(1),resolution(2),resolution(3),&
                        pstress_field(:,:,:,m,n), workfft(:,:,:,m,n), FFTW_PATIENT)
        call dfftw_plan_dft_c2r_3d(plan_fft(2,m,n),resolution(1),resolution(2),resolution(3),&
                        workfft(:,:,:,m,n), ddefgrad(:,:,:), FFTW_PATIENT)
      enddo; enddo

      prodnn = resolution(1)*resolution(2)*resolution(3)
      wgt = 1_pReal/real(prodnn, pReal)
      defgradAim = math_I3
      defgradAimOld = math_I3
      defgrad_av = math_I3

    ! Initialization of CPFEM_general (= constitutive law) and of deformation gradient field
      ielem = 0_pInt
      c066 = 0.0_pReal
      do k = 1, resolution(3); do j = 1, resolution(2); do i = 1, resolution(1)
        defgradold(i,j,k,:,:) = math_I3                       !no deformation at the beginning
        defgrad(i,j,k,:,:) = math_I3
        ielem = ielem +1
        call CPFEM_general(2,math_I3,math_I3,temperature,0.0_pReal,ielem,1_pInt,cstress,dsde,pstress,dPdF)
        c066 = c066 + dsde
      enddo; enddo; enddo
        c066 = c066 * wgt
        c0 = math_mandel66to3333(c066)
        call math_invert(6, c066, s066,i, errmatinv)
        s0 = math_mandel66to3333(s066)

    !calculation of xinormdyad (to calculate gamma_hat) and xi (waves, for proof of equilibrium)
      do k = 1, resolution(3)
        k_s(3) = k-1
        if(k > resolution(3)/2+1) k_s(3) = k_s(3)-resolution(3)
        do j = 1, resolution(2)
          k_s(2) = j-1
          if(j > resolution(2)/2+1) k_s(2) = k_s(2)-resolution(2)
          do i = 1, resolution(1)/2+1
            k_s(1) = i-1
            xi(i,j,k,3) = 0.0_pReal
            if(resolution(3) > 1) xi(i,j,k,3) = real(k_s(3), pReal)/geomdimension(3)
                                  xi(i,j,k,2) = real(k_s(2), pReal)/geomdimension(2)
                                  xi(i,j,k,1) = real(k_s(1), pReal)/geomdimension(1)
            if (any(xi(i,j,k,:) /= 0.0_pReal)) then
              do l = 1,3; do m = 1,3
                 xinormdyad(l,m) = xi(i,j,k, l)*xi(i,j,k, m)/sum(xi(i,j,k,:)**2)
              enddo; enddo
            else
              xinormdyad = 0.0_pReal
            endif
            temp33_Real = math_mul3333xx33(c0, xinormdyad)
```

```
          temp33_Real = math_inv3x3(temp33_Real)
          do l=1,3; do m=1,3; do n=1,3; do p=1,3
           gamma_hat(i,j,k, l,m,n,p) = - (0.5*temp33_Real(l,n)+0.5*temp33_Real(n,l)) *&
                                        (0.5*xinormdyad(m,p)+0.5*xinormdyad(p,m))
          enddo; enddo; enddo; enddo
 310   enddo; enddo; enddo

      ! write header of output file
       open(538,file='results.out',form='UNFORMATTED')
       path = getLoadcaseName()
       write(538), 'Loadcase',trim(path)
       write(538), 'Workingdir',trim(getSolverWorkingDirectoryName())
       path = getSolverJobName()
       write(538), 'JobName',trim(path)//InputFileExtension
       write(538), 'resolution','a', resolution(1),'b', resolution(2),'c', resolution(3)
 320   write(538), 'geomdimension','x', geomdimension(1),'y', geomdimension(2),'z', geomdimension(3)
       write(538), 'materialpoint_sizeResults', materialpoint_sizeResults
       write(538), 'totalincs', sum(bc_steps)
       write(538)  materialpoint_results(:,1,:)
      ! Initialization done

      !*************************************************************
      !Loop over loadcases defined in the loadcase file
       do loadcase = 1, N_Loadcases
      !*************************************************************
 330
         timeinc = bc_timeIncrement(loadcase)/bc_steps(loadcase)
         guessmode = 0.0_pReal                          ! change of load case, homogeneous guess for the first step
         mask_defgrad =  merge(ones,zeroes,bc_mask(:,:,1,loadcase))
         mask_stress =  merge(ones,zeroes,bc_mask(:,:,2,loadcase))
         damper = ones/10
      !*************************************************************
      ! loop oper steps defined in input file for current loadcase
         do steps = 1, bc_steps(loadcase)
      !*************************************************************
 340       temp33_Real = defgradAim
           defgradAim = defgradAim &                    ! update macroscopic displacement gradient (defgrad BC)
                      + guessmode * mask_stress * (defgradAim - defgradAimOld) &
                      + math_mul33x33(bc_velocityGrad(:,:,loadcase), defgradAim)*timeinc
           defgradAimOld = temp33_Real

           do k = 1, resolution(3); do j = 1, resolution(2); do i = 1, resolution(1)
             temp33_Real = defgrad(i,j,k,:,:)
             defgrad(i,j,k,:,:) = defgrad(i,j,k,:,:)&                          ! old fluctuations as guess for new step
                  + guessmode * (defgrad(i,j,k,:,:) - defgradold(i,j,k,:,:))&  ! no fluctuations for new loadcase
 350              + (1.0_pReal-guessmode) * math_mul33x33(bc_velocityGrad(:,:,loadcase),defgradold(i,j,k,:,:))*timeinc
             defgradold(i,j,k,:,:) = temp33_Real
           enddo; enddo; enddo

           guessmode = 1.0_pReal                        ! keep guessing along former trajectory during same loadcase
           calcmode = 0_pInt                            ! start calculation of BC fulfillment
           CPFEM_mode = 1_pInt                          ! winding forward
           iter = 0_pInt
           err_div= 2_pReal * err_div_tol               ! go into loop
           defgradAimCorr = 0.0_pReal                   ! reset damping calculation
 360       damper = damper * 0.9_pReal

      !*************************************************************
      ! convergence loop
           do while( iter <= itmax .and. &
                   (err_div > err_div_tol .or. &
                    err_stress > err_stress_tol .or. &
                    err_defgrad > err_defgrad_tol))
             iter = iter + 1
             print '(3(A,I5.5,tr2))', ' Loadcase = ',loadcase, ' Step = ',steps,'Iteration = ',iter
 370   !*************************************************************

      ! adjust defgrad to fulfill BCs
           select case (calcmode)
           case (0)
             print *, 'Update Stress Field (constitutive evaluation P(F))'
             ielem = 0_pInt
             do k = 1, resolution(3); do j = 1, resolution(2); do i = 1, resolution(1)
               ielem = ielem + 1
               call CPFEM_general(3, defgradold(i,j,k,:,:), defgrad(i,j,k,:,:),&
 380                              temperature,timeinc,ielem,1_pInt,&
                                  cstress,dsde, pstress, dPdF)
             enddo; enddo; enddo

             ielem = 0_pInt
             do k = 1, resolution(3); do j = 1, resolution(2); do i = 1, resolution(1)
```

```
            ielem = ielem + 1_pInt
            call CPFEM_general(CPFEM_mode,&          ! first element in first iteration retains CPFEM_mode 1,
                                                     ! others get 2 (saves winding forward effort)
                               defgradold(i,j,k,:,:), defgrad(i,j,k,:,:),&
                               temperature,timeinc,ielem,1_pInt,&
                               cstress,dsde, pstress, dPdF)
            CPFEM_mode = 2_pInt
            pstress_field(i,j,k,:,:) = pstress
            cstress_field(i,j,k,:,:) = math_mandel6to33(cstress)
          enddo; enddo; enddo

          do m = 1,3; do n = 1,3
            pstress_av(m,n) = sum(pstress_field(:,:,:,m,n)) * wgt
            cstress_av(m,n) = sum(cstress_field(:,:,:,m,n)) * wgt
            defgrad_av(m,n) = sum(defgrad(:,:,:,m,n)) * wgt
          enddo; enddo

          err_stress = maxval(abs(mask_stress * (pstress_av - bc_stress(:,:,loadcase))))
          err_stress_tol = maxval(abs(pstress_av))*err_stress_tolrel

          print*, 'Correcting deformation gradient to fullfill BCs'
          defgradAimCorrPrev=defgradAimCorr
          defgradAimCorr     =-mask_stress*math_mul3333xx33(s0, (mask_stress*(pstress_av - bc_stress(:,:,loadcase))))

          do m=1,3; do n =1,3                    ! calculate damper (correction is far to strong)
            if ( sign(1.0_pReal,defgradAimCorr(m,n))/=sign(1.0_pReal,defgradAimCorrPrev(m,n))) then
              damper(m,n) = max(0.01_pReal,damper(m,n)*0.8)
            else
              damper(m,n) = min(1.0_pReal,damper(m,n) *1.2)
            endif
          enddo; enddo
          defgradAimCorr = mask_Stress*(damper * defgradAimCorr)
          defgradAim = defgradAim + defgradAimCorr

          do m = 1,3; do n = 1,3                 ! anticipated target minus current state
            defgrad(:,:,:,m,n) = defgrad(:,:,:,m,n) + (defgradAim(m,n) - defgrad_av(m,n))
          enddo; enddo
          err_div = 2 * err_div_tol
          err_defgrad = maxval(abs(mask_defgrad * (defgrad_av - defgradAim)))
          print '(a,/,3(3(f12.7,x)/))', ' Deformation Gradient:   ',defgrad_av(1:3,:)
          print '(a,/,3(3(f10.4,x)/))', ' Cauchy Stress [MPa]: ',cstress_av(1:3,:)/1.e6
          print '(2(a,E8.2))', ' error stress               ',err_stress,'   Tol. = ', err_stress_tol
          print '(2(a,E8.2))', ' error deformation gradient ',err_defgrad,'   Tol. = ', err_defgrad_tol*0.8
          if(err_stress < err_stress_tol*0.8) then
            calcmode = 1
          endif

      ! Using spectral method to calculate the change of deformation gradient, check divergence of stress field
        case (1)
          print *, 'Update Stress Field (constitutive evaluation P(F))'
          ielem = 0_pInt
          do k = 1, resolution(3); do j = 1, resolution(2); do i = 1, resolution(1)
            ielem = ielem + 1
            call CPFEM_general(3, defgradold(i,j,k,:,:), defgrad(i,j,k,:,:),&
                               temperature,timeinc,ielem,1_pInt,&
                               cstress,dsde, pstress, dPdF)
          enddo; enddo; enddo

          ielem = 0_pInt
          do k = 1, resolution(3); do j = 1, resolution(2); do i = 1, resolution(1)
            ielem = ielem + 1
            call CPFEM_general(2,&
                               defgradold(i,j,k,:,:), defgrad(i,j,k,:,:),&
                               temperature,timeinc,ielem,1_pInt,&
                               cstress,dsde, pstress, dPdF)
            pstress_field(i,j,k,:,:) = pstress
            cstress_field(i,j,k,:,:) = math_mandel6to33(cstress)
          enddo; enddo; enddo

          print *, 'Calculating equilibrium using spectral method'
          err_div = 0.0_pReal; sigma0 = 0.0_pReal
          do m = 1,3; do n = 1,3
            call dfftw_execute_dft_r2c(plan_fft(1,m,n), pstress_field(:,:,:,m,n),workfft(:,:,:,m,n))
            if(n==3) sigma0 = max(sigma0, sum(abs(workfft(1,1,1,m,:))))          ! L infinity Norm of stress tensor
          enddo; enddo                                                           ! L infinity Norm of div(stress)
          err_div=(maxval(abs(math_mul33x3_complex(workfft(resolution(1)/2+1,resolution(2)/2+1,&
                   resolution(3)/2+1,:,:),xi(resolution(1)/2+1,resolution(2)/2+1,resolution(3)/2+1,:)))))
          do k = 1, resolution(3); do j = 1, resolution(2); do i = 1, resolution(1)/2+1
            temp33_Complex = 0.0_pReal
            do m = 1,3; do n = 1,3
              temp33_Complex(m,n) = sum(gamma_hat(i,j,k,m,n,:,:) * workfft(i,j,k,:,:))
```

```
            enddo; enddo
            workfft(i,j,k,:,:) = temp33_Complex(:,:)
          enddo; enddo; enddo
          workfft(1,1,1,:,:) = defgrad_av - math_I3

          err_div = err_div/sigma0        ! weighting of error

          do m = 1,3; do n = 1,3
             call dfftw_execute_dft_c2r(plan_fft(2,m,n), workfft(:,:,:,m,n),ddefgrad(:,:,:))
             defgrad(:,:,:,m,n) = defgrad(:,:,:,m,n) + ddefgrad * wgt
             pstress_av(m,n) = sum(pstress_field(:,:,:,m,n))*wgt
             cstress_av(m,n) = sum(cstress_field(:,:,:,m,n))*wgt
             defgrad_av(m,n) = sum(defgrad(:,:,:,m,n))*wgt
             defgrad(:,:,:,m,n)=defgrad(:,:,:,m,n) + &
                                (defgradAim(m,n) - defgrad_av(m,n)) ! anticipated target minus current state
          enddo; enddo

          err_stress = maxval(abs(mask_stress * (pstress_av - bc_stress(:,:,loadcase))))
          err_stress_tol = maxval(abs(pstress_av))*err_stress_tolrel                ! accecpt relativ error
          err_defgrad = maxval(abs(mask_defgrad * (defgrad_av - defgradAim)))

          print '(2(a,E8.2))', ' error divergence ',err_div,'  Tol. ', err_div_tol
          print '(2(a,E8.2))', ' error stress ',err_stress,'  Tol. ', err_stress_tol
          print '(2(a,E8.2))', ' error deformation gradient ',err_defgrad,'  Tol. ', err_defgrad_tol
          if((err_stress > err_stress_tol .or. err_defgrad > err_defgrad_tol) .and. err_div < err_div_tol) then
            calcmode = 0                                  ! change to calculation of BCs, reset damper etc.
            defgradAimCorr = 0.0_pReal
            damper = damper * 0.9_pReal
          endif
        end select
      enddo    ! end looping when convergency is achieved

      write(538)  materialpoint_results(:,1,:)  !write to output file

      print '(a,/,3(3(f12.7,x)/))', ' Deformation Aim: ',defgradAim(1:3,:)
      print '(a,/,3(3(f12.7,x)/))', ' Deformation Gradient: ',defgrad_av(1:3,:)
      print '(a,/,3(3(f10.4,x)/))', ' Cauchy Stress [MPa]: ',cstress_av(1:3,:)/1.e6
      print '(A)', '************************************************************'
    enddo  ! end looping over steps in current loadcase
  enddo    ! end looping over loadcases
  close(538)

  do i=1,2; do m = 1,3; do n = 1,3
    call dfftw_destroy_plan(plan_fft(i,m,n))
  enddo; enddo; enddo

end program mpie_spectral

!********************************************************************
! quit subroutine to satisfy IO_error
!
!********************************************************************
subroutine quit(id)
 use prec
 implicit none

 integer(pInt) id

 stop
end subroutine
```

Listing A.2: `mpie_spectral_interface.f90`

```
!* $Id: mpie_spectral_interface.f90 650 2010-09-23 08:05:50Z MPIE\m.diehl $
!--------------------------------------------------------------------

MODULE mpie_interface
 use prec, only: pInt, pReal
 character(len=64), parameter :: FEsolver = 'Spectral'
 character(len=5),  parameter :: InputFileExtension = '.geom'

CONTAINS
!--------------------------------------------------------------------
! initialize interface module
!--------------------------------------------------------------------
subroutine mpie_interface_init()

 write(6,*)
 write(6,*) '<<<+-  mpie_spectral_init  -+>>>'
 write(6,*) '$Id: mpie_spectral_interface.f90 650 2010-09-23 08:05:50Z MPIE\m.diehl $'
 write(6,*)

 return
endsubroutine

!--------------------------------------------------------------------
! extract working directory from loadcase file, possibly based on current working dir
!--------------------------------------------------------------------
function getSolverWorkingDirectoryName()

 implicit none

 character(len=1024) cwd,outname,getSolverWorkingDirectoryName
 character(len=*), parameter :: pathSep = achar(47)//achar(92) ! forwardslash, backwardslash

 call getarg(2,outname)                                ! path to loadFile

 if (scan(outname,pathSep) == 1) then                  ! absolute path given as command line argument
   getSolverWorkingDirectoryName = outname(1:scan(outname,pathSep,back=.true.))
 else
   call getcwd(cwd)
   getSolverWorkingDirectoryName = trim(cwd)//'/'//outname(1:scan(outname,pathSep,back=.true.))
 endif

 getSolverWorkingDirectoryName = rectifyPath(getSolverWorkingDirectoryName)

 return
endfunction

!--------------------------------------------------------------------
! basename of geometry file from command line arguments
!--------------------------------------------------------------------
function getSolverJobName()

 use prec, only: pInt

 implicit none

 character(1024) getSolverJobName, outName, cwd
 character(len=*), parameter :: pathSep = achar(47)//achar(92) ! /, \
 integer(pInt) posExt,posSep

 getSolverJobName = ''

 call getarg(1,outName)
 posExt = scan(outName,'.',back=.true.)
 posSep = scan(outName,pathSep,back=.true.)

 if (posExt <= posSep) posExt = len_trim(outName)+1    ! no extension present
 getSolverJobName = outName(1:posExt-1)                ! path to geometry file (excl. extension)

 if (scan(getSolverJobName,pathSep) /= 1) then         ! relative path given as command line argument
   call getcwd(cwd)
   getSolverJobName = rectifyPath(trim(cwd)//'/'//getSolverJobName)
 else
   getSolverJobName = rectifyPath(getSolverJobName)
 endif

 getSolverJobName = makeRelativePath(getSolverWorkingDirectoryName(),&
                                     getSolverJobName)
 return
endfunction
```

```
!*********************************************************************
! relative path of loadcase from command line arguments
!*********************************************************************
function getLoadcaseName()

 use prec, only: pInt

 implicit none

 character(len=1024) getLoadcaseName, outName, cwd
 character(len=*), parameter :: pathSep = achar(47)//achar(92) ! /, \
 integer(pInt) posExt,posSep
 posExt = 0

 call getarg(2,getLoadcaseName)
 posExt = scan(getLoadcaseName,'.',back=.true.)
 posSep = scan(getLoadcaseName,pathSep,back=.true.)

 if (posExt <= posSep) getLoadcaseName = trim(getLoadcaseName)//('.load')    ! no extension present
 if (scan(getLoadcaseName,pathSep) /= 1) then           ! relative path given as command line argument
   call getcwd(cwd)
   getLoadcaseName = rectifyPath(trim(cwd)//'/'//getLoadcaseName)
 else
   getLoadcaseName = rectifyPath(getLoadcaseName)
 endif

 getLoadcaseName = makeRelativePath(getSolverWorkingDirectoryName(),&
                                    getLoadcaseName)
 return
endfunction


!*********************************************************************
! remove ../ and ./ from path
!*********************************************************************
function rectifyPath(path)

 use prec, only: pInt

 implicit none

 character(len=*) path
 character(len=len_trim(path)) rectifyPath
 integer(pInt) i,j,k,l

 !remove ./ from path
 l = len_trim(path)
 rectifyPath = path
 do i = l,2,-1
    if ( rectifyPath(i-1:i) == './' .and. rectifyPath(i-2:i-2) /= '.' ) &
      rectifyPath(i-1:l) = rectifyPath(i+1:l)//'␣␣'
 enddo

 !remove ../ and corresponding directory from rectifyPath
 l = len_trim(rectifyPath)
 i = index(rectifyPath(i:l),'../')
 j = 0_pInt
 do while (i > j)
    j = scan(rectifyPath(:i-2),'/',back=.true.)
    rectifyPath(j+1:l) = rectifyPath(i+3:l)//repeat('␣',2+i-j)
    i = j+index(rectifyPath(j+1:l),'../')
 enddo
 if(len_trim(rectifyPath) == 0) rectifyPath = '/'
 return
 endfunction rectifyPath


!*********************************************************************
! relative path from absolute a to absolute b
!*********************************************************************
function makeRelativePath(a,b)

 use prec, only: pInt

 implicit none

 character (len=*) :: a,b
 character (len=1024) :: makeRelativePath
 integer(pInt) i,posLastCommonSlash,remainingSlashes

 posLastCommonSlash = 0
```

```
160   remainingSlashes = 0
      do i = 1,min(1024,len_trim(a),len_trim(b))
        if (a(i:i) /= b(i:i)) exit
        if (a(i:i) == '/') posLastCommonSlash = i
      enddo
      do i = posLastCommonSlash+1,len_trim(a)
        if (a(i:i) == '/') remainingSlashes = remainingSlashes + 1
      enddo
      makeRelativePath = repeat('../',remainingSlashes)//b(posLastCommonSlash+1:len_trim(b))

170   return
    endfunction makeRelativePath

    END MODULE
```

Listing A.3: `mesh.f90`

```
!* $Id: mesh.f90 661 2010-10-01 10:42:15Z MPIE\m.diehl $
!##############################################################
MODULE mesh
!##############################################################

 use prec, only: pReal,pInt
 implicit none

 ! Generic functions, used independently of the chosen solver.
 !
 ! _Nelems     : total number of elements in mesh
 ! _NcpElems   : total number of CP elements in mesh
 ! _Nnodes     : total number of nodes in mesh
 ! _maxNnodes  : max number of nodes in any CP element
 ! _maxNips    : max number of IPs in any CP element
 ! _maxNipNeighbors    : max number of IP neighbors in any CP element
 ! _maxNsharedElems : max number of CP elements sharing a node
 !
 ! _element    : FEid, type(internal representation), material, texture, node indices
 ! _node       : x,y,z coordinates (initially!)
 ! _sharedElem : entryCount and list of elements containing node
 !
 ! _mapFEtoCPelem : [sorted FEid, corresponding CPid]
 ! _mapFEtoCPnode : [sorted FEid, corresponding CPid]
 !
 ! The following definitions are solver-dependent.
 ! The definitions for FEM are removed in order to get a small appendix.
 ! Only the routines that are especially written for the spectral method (named '*spectral*') are included.
 !
 ! _Nnodes             : # nodes in a specific type of element (how we use it)
 ! _NoriginalNodes     : # nodes in a specific type of element (how it is originally defined by marc)
 ! _Nips               : # IPs in a specific type of element
 ! _NipNeighbors       : # IP neighbors in a specific type of element
 ! _ipNeighbor         : +x,-x,+y,-y,+z,-z list of intra-element IPs and
 !     (negative) neighbor faces per own IP in a specific type of element
 ! _NfaceNodes         : # nodes per face in a specific type of element
 ! _nodeOnFace         : list of node indices on each face of a specific type of element
 ! _maxNnodesAtIP      : max number of (equivalent) nodes attached to an IP
 ! _nodesAtIP          : map IP index to two node indices in a specific type of element
 ! _ipNeighborhood     : 6 or less neighboring IPs as [element_num, IP_index]
 ! _NsubNodes         : # subnodes required to fully define all IP volumes

 !     order is +x,-x,+y,-y,+z,-z but meaning strongly depends on Elemtype
 ! -------------------------
 integer(pInt) mesh_Nelems,mesh_NcpElems,mesh_NelemSets,mesh_maxNelemInSet
 integer(pInt) mesh_Nmaterials
 integer(pInt) mesh_Nnodes,mesh_maxNnodes,mesh_maxNips,mesh_maxNipNeighbors,mesh_maxNsharedElems,mesh_maxNsubNodes
 integer(pInt), dimension(2) :: mesh_maxValStateVar = 0_pInt
 character(len=64), dimension(:),   allocatable :: mesh_nameElemSet, &     ! names of elementSet
                                                   mesh_nameMaterial, &  ! names of material in solid section
                                                   mesh_mapMaterial      ! name of elementSet for material
 integer(pInt), dimension(:,:),     allocatable :: mesh_mapElemSet        ! list of elements in elementSet
 integer(pInt), dimension(:,:),     allocatable, target :: mesh_mapFEtoCPelem, mesh_mapFEtoCPnode
 integer(pInt), dimension(:,:),     allocatable :: mesh_element, mesh_sharedElem
 integer(pInt), dimension(:,:,:,:), allocatable :: mesh_ipNeighborhood

 real(pReal),   dimension(:,:,:),   allocatable :: mesh_subNodeCoord       ! coordinates of subnodes per element
 real(pReal),   dimension(:,:),     allocatable :: mesh_ipVolume           ! volume associated with IP
 real(pReal),   dimension(:,:,:),   allocatable :: mesh_ipArea, &          ! area of interface to neighboring IP
                                                   mesh_ipCenterOfGravity ! center of gravity of IP
 real(pReal),   dimension(:,:,:,:), allocatable :: mesh_ipAreaNormal       ! area normal of interface to neighboring IP
 real(pReal),                       allocatable :: mesh_node (:,:)

 integer(pInt), dimension(:,:,:),   allocatable :: FE_nodesAtIP
 integer(pInt), dimension(:,:,:),   allocatable :: FE_ipNeighbor
 integer(pInt), dimension(:,:,:),   allocatable :: FE_subNodeParent
 integer(pInt), dimension(:,:,:,:), allocatable :: FE_subNodeOnIPFace

 integer(pInt) :: hypoelasticTableStyle
 integer(pInt) :: initialcondTableStyle
 integer(pInt), parameter :: FE_Nelemtypes = 1
 integer(pInt), parameter :: FE_maxNnodes = 8
 integer(pInt), parameter :: FE_maxNsubNodes = 0
 integer(pInt), parameter :: FE_maxNips = 1
 integer(pInt), parameter :: FE_maxNipNeighbors = 6
 integer(pInt), parameter :: FE_maxmaxNnodesAtIP = 8
 integer(pInt), parameter :: FE_NipFaceNodes = 4
```

```
  integer(pInt), dimension(FE_Nelemtypes), parameter :: FE_Nnodes = &
 80 (/8, & ! element 117
  /)
  integer(pInt), dimension(FE_Nelemtypes), parameter :: FE_NoriginalNodes = &
  (/8, & ! element 117
  /)
  integer(pInt), dimension(FE_Nelemtypes), parameter :: FE_Nips = &
  (/1, & ! element 117
  /)
  integer(pInt), dimension(FE_Nelemtypes), parameter :: FE_NipNeighbors = &
  (/6, & ! element 117
 90 /)
  integer(pInt), dimension(FE_Nelemtypes), parameter :: FE_NsubNodes = &
  (/0, & ! element 117
  /)
  integer(pInt), dimension(FE_maxNipNeighbors,FE_Nelemtypes), parameter :: FE_NfaceNodes = &
  reshape((/&
   4,4,4,4,4,4, & ! element 117
   /),(/FE_maxNipNeighbors,FE_Nelemtypes/))
  integer(pInt), dimension(FE_Nelemtypes), parameter :: FE_maxNnodesAtIP = &
  (/8, & ! element 117
100 /)
  integer(pInt), dimension(FE_NipFaceNodes,FE_maxNipNeighbors,FE_Nelemtypes), parameter :: FE_nodeOnFace = &
  reshape((/&
   1,2,3,4 , & ! element 117
   2,1,5,6 , &
   3,2,6,7 , &
   4,3,7,8 , &
   4,1,5,8 , &
   8,7,6,5 , &
   /),(/FE_NipFaceNodes,FE_maxNipNeighbors,FE_Nelemtypes/))
110
  CONTAINS
  !-------------------------
  ! subroutine mesh_init()
  ! function mesh_FEtoCPelement(FEid)
  ! function mesh_build_ipNeighorhood()
  !-------------------------

  !*************************************************************
  ! initialization
120 !*************************************************************
  subroutine mesh_init (ip,element)

  use mpie_interface
  use prec, only: pInt
  use IO, only: IO_error,IO_open_InputFile
  use FEsolving, only: parallelExecution, FEsolving_execElem, FEsolving_execIP, calcMode, lastMode

  implicit none

130 integer(pInt), parameter :: fileUnit = 222
  integer(pInt) e,element,ip

  write(6,*)
  write(6,*) '<<<+-  mesh_init  -+>>>'
  write(6,*) '$Id: mesh.f90 661 2010-10-01 10:42:15Z MPIE\m.diehl $'
  write(6,*)

  call mesh_build_FEdata()                          ! --- get properties of the different types of elements

140 if (IO_open_inputFile(fileUnit)) then             ! --- parse info from input file...

  select case (FEsolver)
      case ('Spectral')
                          call mesh_spectral_count_nodesAndElements(fileUnit)
                          call mesh_spectral_count_cpElements()
                          call mesh_spectral_map_elements()
                          call mesh_spectral_map_nodes()
                          call mesh_spectral_count_cpSizes()
                          call mesh_spectral_build_nodes(fileUnit)
150                       call mesh_spectral_build_elements(fileUnit)

    end select
    close (fileUnit)

    call mesh_build_sharedElems()
    call mesh_build_ipNeighborhood()
    call mesh_build_subNodeCoords()
    call mesh_build_ipVolumes()
    call mesh_build_ipAreas()
```

```
    call mesh_tell_statistics()

    parallelExecution = (parallelExecution .and. (mesh_Nelems == mesh_NcpElems))
  else
    call IO_error(101) ! cannot open input file
  endif

  FEsolving_execElem = (/1,mesh_NcpElems/)
  allocate(FEsolving_execIP(2,mesh_NcpElems)); FEsolving_execIP = 1_pInt
  forall (e = 1:mesh_NcpElems) FEsolving_execIP(2,e) = FE_Nips(mesh_element(2,e))

  allocate(calcMode(mesh_maxNips,mesh_NcpElems))
  write(6,*) '<<<+-  mesh_init done  +>>>'
  calcMode = .false.                                    ! pretend to have collected what first call is asking (F = I)
  calcMode(ip,mesh_FEasCP('elem',element)) = .true.     ! first ip,el needs to be already pingponged to "calc"
  lastMode = .true.                                     ! and its mode is already known...
  endsubroutine

!*************************************************************
! mapping of FE element types to internal representation
!*************************************************************
  function FE_mapElemtype(what)

  use IO, only: IO_lc

  implicit none

  character(len=*), intent(in) :: what
  integer(pInt) FE_mapElemtype

  select case (IO_lc(what))
     case ( '117', &
            '123', &
            'c3d8r')
       FE_mapElemtype = 8       ! Three-dimensional Arbitrarily Distorted linear hexahedral with reduced integration
     case default
       FE_mapElemtype = 0       ! unknown element --> should raise an error upstream..!
  endselect

  endfunction

!*************************************************************
! FE to CP id mapping by binary search thru lookup array
!
! valid questions are 'elem', 'node'
!*************************************************************
  function mesh_FEasCP(what,id)

  use prec, only: pInt
  use IO, only: IO_lc
  implicit none

  character(len=*), intent(in) :: what
  integer(pInt), intent(in) :: id
  integer(pInt), dimension(:,:), pointer :: lookupMap
  integer(pInt) mesh_FEasCP, lower,upper,center

  mesh_FEasCP = 0_pInt
  select case(IO_lc(what(1:4)))
    case('elem')
      lookupMap => mesh_mapFEtoCPelem
    case('node')
      lookupMap => mesh_mapFEtoCPnode
    case default
      return
  endselect

  lower = 1_pInt
  upper = size(lookupMap,2)

! check at bounds
  if (lookupMap(1,lower) == id) then
    mesh_FEasCP = lookupMap(2,lower)
    return
  elseif (lookupMap(1,upper) == id) then
    mesh_FEasCP = lookupMap(2,upper)
    return
  endif
```

```
  ! binary search in between bounds
  do while (upper-lower > 1)
    center = (lower+upper)/2
    if (lookupMap(1,center) < id) then
      lower = center
    elseif (lookupMap(1,center) > id) then
      upper = center
    else
      mesh_FEasCP = lookupMap(2,center)
      exit
    endif
  enddo
  return

  endfunction

!************************************************************
! find face-matching element of same type
!************************************************************
  function mesh_faceMatch(elem,face)

  use prec, only: pInt
  implicit none

  integer(pInt) face,elem
  integer(pInt), dimension(2) :: mesh_faceMatch                  ! matching element's ID and corresponding face ID
  integer(pInt), dimension(FE_NfaceNodes(face,mesh_element(2,elem))) :: myFaceNodes ! global node ids on my face
  integer(pInt) minN,NsharedElems, &
                t, lonelyNode, &
                candidateType,candidateElem, &
                i,f,n

  minN = mesh_maxNsharedElems+1          ! init to worst case
  mesh_faceMatch = 0_pInt                ! intialize to "no match found"
  t = mesh_element(2,elem)               ! figure elemType

  do n = 1,FE_NfaceNodes(face,t)         ! loop over nodes on face
    myFaceNodes(n) = mesh_FEasCP('node',mesh_element(4+FE_nodeOnFace(n,face,t),elem)) ! CP id of face node
    NsharedElems = mesh_sharedElem(1,myFaceNodes(n)) ! figure # shared elements for this node
    if (NsharedElems < minN) then
      minN = NsharedElems                ! remember min # shared elems
      lonelyNode = n                     ! remember most lonely node
    endif
  enddo


candidate: do i = 1,minN                 ! iterate over lonelyNode's shared elements
    candidateElem = mesh_sharedElem(1+i,myFaceNodes(lonelyNode)) ! present candidate elem
    if (candidateElem == elem) cycle candidate          ! my own element ?
    candidateType = mesh_element(2,candidateElem)       ! figure elemType of candidate

candidateFace: do f = 1,FE_maxNipNeighbors              ! check each face of candidate
        if (FE_NfaceNodes(f,candidateType) /= FE_NfaceNodes(face,t)) &
          cycle candidateFace                           ! incompatible face
        do n = 1,FE_NfaceNodes(f,candidateType)         ! loop through nodes on face
          if (all(myFaceNodes /= &
              mesh_FEasCP('node', &
                mesh_element(4+FE_nodeOnFace(n,f,candidateType),candidateElem)))) &
            cycle candidateFace  ! other face node not one of my face nodes
        enddo
        mesh_faceMatch(1) = f
        mesh_faceMatch(2) = candidateElem
        exit candidate                                  ! found my matching candidate
      enddo candidateFace
  enddo candidate

  return

  endfunction

!************************************************************************
! get properties of different types of finite elements
! assign globals:
! FE_nodesAtIP, FE_ipNeighbor, FE_subNodeParent, FE_subNodeOnIPFace
!************************************************************************
  subroutine mesh_build_FEdata ()

  use prec, only: pInt
  implicit none

  allocate(FE_nodesAtIP(FE_maxmaxNnodesAtIP,FE_maxNips,FE_Nelemtypes)); FE_nodesAtIP = 0_pInt
```

```
      allocate(FE_ipNeighbor(FE_maxNipNeighbors,FE_maxNips,FE_Nelemtypes)); FE_ipNeighbor = 0_pInt
      allocate(FE_subNodeParent(FE_maxNips,FE_maxNsubNodes,FE_Nelemtypes)); FE_subNodeParent = 0_pInt
      allocate(FE_subNodeOnIPFace(FE_NipFaceNodes,FE_maxNipNeighbors,FE_maxNips,FE_Nelemtypes))
      FE_subNodeOnIPFace=0_pInt

    ! fill FE_nodesAtIP with data
      FE_nodesAtIP(:,:FE_Nips(8),8) = &  ! element 117
         reshape((/&
330      1,2,3,4,5,6,7,8    &
         /),(/FE_maxNnodesAtIP(8),FE_Nips(8)/))

    ! fill FE_ipNeighbor with data
      FE_ipNeighbor(:FE_NipNeighbors(8),:FE_Nips(8),8) = &  ! element 117
         reshape((/&
         -3,-5,-4,-2,-6,-1    &
         /),(/FE_NipNeighbors(8),FE_Nips(8)/))

    ! fill FE_subNodeParent with data
340 !FE_subNodeParent(:FE_Nips(8),:FE_NsubNodes(8),8)       ! element 117 has no subnodes

    ! fill FE_subNodeOnIPFace with data
      FE_subNodeOnIPFace(:FE_NipFaceNodes,:FE_NipNeighbors(8),:FE_Nips(8),8) = &  ! element 117
         reshape((/&
          2, 3, 7, 6, & ! 1
          1, 5, 8, 4, &
          3, 4, 8, 7, &
          1, 2, 6, 5, &
          5, 6, 7, 8, &
350       1, 4, 3, 2  &
         /),(/FE_NipFaceNodes,FE_NipNeighbors(8),FE_Nips(8)/))
      return

      endsubroutine

    !*************************************************************************
    ! count overall number of nodes and elements in mesh
    !
    ! mesh_Nelems, mesh_Nnodes
360 !*************************************************************************
      subroutine mesh_spectral_count_nodesAndElements (unit)

      use prec, only: pInt
      use IO
      implicit none

      integer(pInt), parameter :: maxNchunks = 7
      integer(pInt), dimension (1+2*maxNchunks) :: pos
      integer(pInt) a,b,c,i
370
      integer(pInt) unit
      character(len=1024) line

      mesh_Nnodes = 0_pInt
      mesh_Nelems = 0_pInt

      rewind(unit)
      do
        read(unit,'(a1024)',END=100) line
380     if (IO_isBlank(line)) cycle                               ! skip empty lines
        pos = IO_stringPos(line,maxNchunks)

        if ( IO_lc(IO_StringValue(line,pos,1)) == 'resolution') then
          do i = 2,6,2
            select case (IO_lc(IO_stringValue(line,pos,i)))
              case('a')
                  a = IO_intValue(line,pos,i+1)
              case('b')
                  b = IO_intValue(line,pos,i+1)
390           case('c')
                  c = IO_intValue(line,pos,i+1)
            end select
          enddo
          mesh_Nelems = a * b * c
          mesh_Nnodes = (1 + a)*(1 + b)*(1 + c)
          exit
        endif
      enddo

400 100 return

      endsubroutine
```

```
!************************************************************************
! count overall number of cpElements in mesh
!
! mesh_NcpElems
!************************************************************************
 subroutine mesh_spectral_count_cpElements ()

 use prec, only: pInt
 implicit none

 mesh_NcpElems = mesh_Nelems
 return

 endsubroutine


!************************************************************************
! map nodes from FE id to internal (consecutive) representation
!
! allocate globals: mesh_mapFEtoCPnode
!************************************************************************
 subroutine mesh_spectral_map_nodes ()

 use prec, only: pInt

 implicit none
 integer(pInt) i

 allocate (mesh_mapFEtoCPnode(2,mesh_Nnodes)) ; mesh_mapFEtoCPnode = 0_pInt

 forall (i = 1:mesh_Nnodes) &
   mesh_mapFEtoCPnode(:,i) = i

 return

 endsubroutine

!************************************************************************
! map elements from FE id to internal (consecutive) representation
!
! allocate globals: mesh_mapFEtoCPelem
!************************************************************************
 subroutine mesh_spectral_map_elements ()

 use prec, only: pInt

 implicit none
 integer(pInt) i

 allocate (mesh_mapFEtoCPelem(2,mesh_NcpElems)) ; mesh_mapFEtoCPelem = 0_pInt

 forall (i = 1:mesh_NcpElems) &
   mesh_mapFEtoCPelem(:,i) = i

 return

 endsubroutine


!************************************************************************
! get maximum count of nodes, IPs, IP neighbors, and subNodes
! among cpElements
!
! _maxNnodes, _maxNips, _maxNipNeighbors, _maxNsubNodes
!************************************************************************
subroutine mesh_spectral_count_cpSizes ()

 use prec, only: pInt
 implicit none

 integer(pInt) t

 t = FE_mapElemtype('C3D8R')                    ! fake 3D hexahedral 8 node 1 IP element

 mesh_maxNnodes =       FE_Nnodes(t)
 mesh_maxNips =         FE_Nips(t)
 mesh_maxNipNeighbors = FE_NipNeighbors(t)
 mesh_maxNsubNodes =    FE_NsubNodes(t)

 endsubroutine
```

```
!************************************************************************
! store x,y,z coordinates of all nodes in mesh
!
! allocate globals:
! _node
!************************************************************************
 subroutine mesh_spectral_build_nodes (unit)

 use prec, only: pInt
 use IO
 implicit none

 integer(pInt), parameter :: maxNchunks = 7
 integer(pInt), dimension (1+2*maxNchunks) :: pos
 integer(pInt) a,b,c,n,i
 real(pReal)   x,y,z
 logical gotResolution,gotDimension
 integer(pInt) unit
 character(len=64) tag
 character(len=1024) line

 allocate ( mesh_node (3,mesh_Nnodes) ); mesh_node = 0_pInt

 a = 1_pInt
 b = 1_pInt
 c = 1_pInt
 x = 1.0_pReal
 y = 1.0_pReal
 z = 1.0_pReal
 gotResolution = .false.
 gotDimension =  .false.

 rewind(unit)
 do
   read(unit,'(a1024)',END=100) line
   if (IO_isBlank(line)) cycle                                  ! skip empty lines
   pos = IO_stringPos(line,maxNchunks)

   select case ( IO_lc(IO_StringValue(line,pos,1)) )
     case ('resolution')
         gotResolution = .true.
         do i = 2,6,2
           tag = IO_lc(IO_stringValue(line,pos,i))
           select case (tag)
             case('a')
                 a = 1 + IO_intValue(line,pos,i+1)
             case('b')
                 b = 1 + IO_intValue(line,pos,i+1)
             case('c')
                 c = 1 + IO_intValue(line,pos,i+1)
           end select
         enddo
     case ('dimension')
         gotDimension = .true.
         do i = 2,6,2
           tag = IO_lc(IO_stringValue(line,pos,i))
           select case (tag)
             case('x')
                 x = IO_floatValue(line,pos,i+1)
             case('y')
                 y = IO_floatValue(line,pos,i+1)
             case('z')
                 z = IO_floatValue(line,pos,i+1)
           end select
         enddo
   end select
   if (gotDimension .and. gotResolution) exit
 enddo

! ------ sanity checks ------
 if (.not. gotDimension .or. .not. gotResolution) call IO_error(42)
 if (a < 2 .or. b < 2 .or. c < 2) call IO_error(43)
 if (x <= 0.0_pReal .or. y <= 0.0_pReal .or. z <= 0.0_pReal) call IO_error(44)

 forall (n = 0:mesh_Nnodes-1)
   mesh_node(1,n+1) = x * dble(mod(n,a)     / (a-1.0_pReal))
   mesh_node(2,n+1) = y * dble(mod(n/a,b)   / (b-1.0_pReal))
   mesh_node(3,n+1) = z * dble(mod(n/a/b,c) / (c-1.0_pReal))
 end forall
100 return
 endsubroutine
```

```
!*************************************************************************
! store FEid, type, mat, tex, and node list per element
!
! allocate globals:
! _element
!*************************************************************************
 subroutine mesh_spectral_build_elements (unit)

 use prec, only: pInt
 use IO
 implicit none

 integer(pInt), parameter :: maxNchunks = 7
 integer(pInt), dimension (1+2*maxNchunks) :: pos
 integer(pInt) a,b,c,e,i,homog
 logical gotResolution,gotDimension,gotHomogenization
 integer(pInt) unit
 character(len=1024) line

 a = 1_pInt
 b = 1_pInt
 c = 1_pInt
 gotResolution =      .false.
 gotDimension =       .false.
 gotHomogenization =  .false.

 rewind(unit)
 do
   read(unit,'(a1024)',END=100) line
   if (IO_isBlank(line)) cycle                                ! skip empty lines
   pos = IO_stringPos(line,maxNchunks)

   select case ( IO_lc(IO_StringValue(line,pos,1)) )
     case ('dimension')
         gotDimension = .true.
     case ('homogenization')
         gotHomogenization = .true.
         homog = IO_intValue(line,pos,2)
     case ('resolution')
         gotResolution = .true.
         do i = 2,6,2
           select case (IO_lc(IO_stringValue(line,pos,i)))
             case('a')
                 a = IO_intValue(line,pos,i+1)
             case('b')
                 b = IO_intValue(line,pos,i+1)
             case('c')
                 c = IO_intValue(line,pos,i+1)
           end select
         enddo
   end select
   if (gotDimension .and. gotHomogenization .and. gotResolution) exit
 enddo

100 allocate (mesh_element (4+mesh_maxNnodes,mesh_NcpElems)) ; mesh_element = 0_pInt

 e = 0_pInt
 do while (e < mesh_NcpElems)
   read(unit,'(a1024)',END=110) line
   if (IO_isBlank(line)) cycle                                ! skip empty lines
   pos(1:1+2*1) = IO_stringPos(line,1)

   e = e+1                                                    ! valid element entry
   mesh_element ( 1,e) = e                                    ! FE id
   mesh_element ( 2,e) = FE_mapElemtype('C3D8R')              ! elem type
   mesh_element ( 3,e) = homog                                ! homogenization
   mesh_element ( 4,e) = IO_IntValue(line,pos,1)              ! microstructure
   mesh_element ( 5,e) = e + (e-1)/a + (e-1)/a/b*(a+1)        ! base node
   mesh_element ( 6,e) = mesh_element ( 5,e) + 1
   mesh_element ( 7,e) = mesh_element ( 5,e) + (a+1) + 1
   mesh_element ( 8,e) = mesh_element ( 5,e) + (a+1)
   mesh_element ( 9,e) = mesh_element ( 5,e) + (a+1)*(b+1) ! second floor base node
   mesh_element (10,e) = mesh_element ( 9,e) + 1
   mesh_element (11,e) = mesh_element ( 9,e) + (a+1) + 1
   mesh_element (12,e) = mesh_element ( 9,e) + (a+1)
   mesh_maxValStateVar(1) = max(mesh_maxValStateVar(1),mesh_element(3,e))    !needed for statistics
   mesh_maxValStateVar(2) = max(mesh_maxValStateVar(2),mesh_element(4,e))
 enddo

110 return
 endsubroutine
```

```
!*************************************************************
! get maximum count of shared elements among cpElements and
! build list of elements shared by each node in mesh
!
! _maxNsharedElems
! _sharedElem
!*************************************************************
 subroutine mesh_build_sharedElems ()

 use prec, only: pInt
 use IO
 implicit none

 integer(pInt) e,t,n,j
 integer(pInt), dimension (mesh_Nnodes) :: node_count
 integer(pInt), dimension (:), allocatable :: node_seen

 allocate(node_seen(maxval(FE_Nnodes)))
 node_count = 0_pInt

 do e = 1,mesh_NcpElems

   t = mesh_element(2,e)                                          ! my type
   node_seen = 0_pInt                                             ! reset node duplicates
   do j = 1,FE_Nnodes(t)                                          ! check each node of element
     n = mesh_FEasCP('node',mesh_element(4+j,e))
     if (all(node_seen /= n)) node_count(n) = node_count(n) + 1_pInt  ! if FE node not yet encountered -> count it
     node_seen(j) = n                                             ! remember this node to be counted already
   enddo

 enddo

 mesh_maxNsharedElems = maxval(node_count)                        ! most shared node

 allocate ( mesh_sharedElem( 1+mesh_maxNsharedElems,mesh_Nnodes) )
 mesh_sharedElem = 0_pInt

 do e = 1,mesh_NcpElems
   node_seen = 0_pInt
   do j = 1,FE_Nnodes(mesh_element(2,e))
     n = mesh_FEasCP('node',mesh_element(4+j,e))
     if (all(node_seen /= n)) then
       mesh_sharedElem(1,n) = mesh_sharedElem(1,n) + 1            ! count for each node the connected elements
       mesh_sharedElem(1+mesh_sharedElem(1,n),n) = e              ! store the respective element id
     endif
     node_seen(j) = n
   enddo
 enddo

 deallocate (node_seen)

 return

 endsubroutine

!*************************************************************
! build up of IP neighborhood
!
! allocate globals
! _ipNeighborhood
!*************************************************************
 subroutine mesh_build_ipNeighborhood()

 use prec, only: pInt
 implicit none

 integer(pInt) e,t,i,j,k,l,m,n,a,anchor, neighborType
 integer(pInt) neighbor,neighboringElem,neighboringIP
 integer(pInt), dimension(2) :: matchingElem                      ! face and id of matching elem
 integer(pInt), dimension(FE_maxmaxNnodesAtIP) :: linkedNodes, &
                                                  matchingNodes

 linkedNodes = 0_pInt

 allocate(mesh_ipNeighborhood(2,mesh_maxNipNeighbors,mesh_maxNips,mesh_NcpElems)) ; mesh_ipNeighborhood = 0_pInt

 do e = 1,mesh_NcpElems                                           ! loop over cpElems
   t = mesh_element(2,e)                                          ! get elemType
   do i = 1,FE_Nips(t)                                            ! loop over IPs of elem
     do n = 1,FE_NipNeighbors(t)                                  ! loop over neighbors of IP
       neighbor = FE_ipNeighbor(n,i,t)
```

```
        if (neighbor > 0) then                                          ! intra-element IP
          neighboringElem = e
          neighboringIP   = neighbor
730     else                                                            ! neighboring element's IP
          neighboringElem = 0_pInt
          neighboringIP = 0_pInt
          matchingElem = mesh_faceMatch(e,-neighbor)                    ! get face and CP elem id of face match
          if (matchingElem(2) > 0_pInt) then                            ! found match?
            neighborType = mesh_element(2,matchingElem(2))
            l = 0_pInt
            do a = 1,FE_maxNnodesAtIP(t)                                ! figure my anchor nodes on connecting face
              anchor = FE_nodesAtIP(a,i,t)
              if (any(FE_nodeOnFace(:,-neighbor,t) == anchor)) then     ! ip anchor sits on face?
740             l = l+1_pInt
                linkedNodes(l) = mesh_element(4+anchor,e)               ! FE id of anchor node
              endif
            enddo

    neighborIP: do j = 1,FE_Nips(neighborType)                          ! loop over neighboring ips
              m = 0_pInt
              matchingNodes = 0_pInt
              do a = 1,FE_maxNnodesAtIP(neighborType)                   ! check each anchor node of that ip
                anchor = FE_nodesAtIP(a,j,neighborType)
                if (     anchor /= 0_pInt .and. &                       ! valid anchor node
750                  any(FE_nodeOnFace(:,matchingElem(1),neighborType) == anchor) ) then
                  m = m+1_pInt
                  matchingNodes(m) = mesh_element(4+anchor,matchingElem(2)) ! FE id of neighbor's anchor node
                endif
              enddo
              if (m /= l) cycle neighborIP                              ! this ip has wrong count of anchors on face
              do a = 1,l
                if (all(matchingNodes /= linkedNodes(a))) cycle neighborIP
              enddo
760           neighboringElem = matchingElem(2)                         ! survival of the fittest
              neighboringIP   = j
              exit neighborIP
            enddo neighborIP
          endif                                                         ! end of valid external matching
        endif                                                           ! end of internal/external matching
        mesh_ipNeighborhood(1,n,i,e) = neighboringElem
        mesh_ipNeighborhood(2,n,i,e) = neighboringIP
      enddo
    enddo
770 enddo

    return
    endsubroutine

    !************************************************************
    ! assignment of coordinates for subnodes in each cp element
    !
    ! allocate globals
    ! _subNodeCoord
780 !************************************************************
    subroutine mesh_build_subNodeCoords()

    use prec, only: pInt,pReal
    implicit none

    integer(pInt) e,t,n,p

    allocate(mesh_subNodeCoord(3,mesh_maxNnodes+mesh_maxNsubNodes,mesh_NcpElems)) ; mesh_subNodeCoord = 0.0_pReal

790 do e = 1,mesh_NcpElems                  ! loop over cpElems
      t = mesh_element(2,e)                 ! get elemType
      do n = 1,FE_Nnodes(t)                 ! loop over nodes of this element type
        mesh_subNodeCoord(:,n,e) = mesh_node(:,mesh_FEasCP('node',mesh_element(4+n,e)))
      enddo
      do n = 1,FE_NsubNodes(t)              ! now for the true subnodes
        do p = 1,FE_Nips(t)                 ! loop through possible parent nodes
          if (FE_subNodeParent(p,n,t) > 0) & ! valid parent node
            mesh_subNodeCoord(:,n+FE_Nnodes(t),e) = &
            mesh_subNodeCoord(:,n+FE_Nnodes(t),e) + &
800         mesh_node(:,mesh_FEasCP('node',mesh_element(4+FE_subNodeParent(p,n,t),e))) ! add up parents
        enddo
        mesh_subNodeCoord(:,n+FE_Nnodes(t),e)=mesh_subNodeCoord(:,n+FE_Nnodes(t),e)/count(FE_subNodeParent(:,n,t)>0)
      enddo
    enddo

    return
    endsubroutine
```

```
!************************************************************
! calculation of IP volume
!
! allocate globals
! _ipVolume
!************************************************************
 subroutine mesh_build_ipVolumes()

 use prec, only: pInt
 use math, only: math_volTetrahedron
 implicit none

 integer(pInt) e,f,t,i,j,k,n
 integer(pInt),parameter::Ntriangles = FE_NipFaceNodes-2     ! each interface is made up of this many triangles
 logical(pInt),dimension(mesh_maxNnodes+mesh_maxNsubNodes)::gravityNode ! flagList to find subnodes determining CoG
 real(pReal),dimension(3,mesh_maxNnodes+mesh_maxNsubNodes)::gravityNodePos ! coord. of subnodes determining CoG
 real(pReal),dimension(3,FE_NipFaceNodes) :: nPos                        ! coord. of nodes on IP face
 real(pReal),dimension(Ntriangles,FE_NipFaceNodes) :: volume             ! volumes of possible tetrahedra
 real(pReal),dimension(3) :: centerOfGravity

 allocate(mesh_ipVolume(mesh_maxNips,mesh_NcpElems)) ;                    mesh_ipVolume = 0.0_pReal
 allocate(mesh_ipCenterOfGravity(3,mesh_maxNips,mesh_NcpElems)) ;  mesh_ipCenterOfGravity = 0.0_pReal

 do e = 1,mesh_NcpElems                                  ! loop over cpElems
   t = mesh_element(2,e)                                 ! get elemType
   do i = 1,FE_Nips(t)                                   ! loop over IPs of elem
     gravityNode = .false.                               ! reset flagList
     gravityNodePos = 0.0_pReal                          ! reset coordinates
     do f = 1,FE_NipNeighbors(t)                         ! loop over interfaces of IP
       do n = 1,FE_NipFaceNodes                          ! loop over nodes on interface
         gravityNode(FE_subNodeOnIPFace(n,f,i,t)) = .true.
         gravityNodePos(:,FE_subNodeOnIPFace(n,f,i,t)) = mesh_subNodeCoord(:,FE_subNodeOnIPFace(n,f,i,t),e)
       enddo
     enddo

     do j = 1,mesh_maxNnodes+mesh_maxNsubNodes-1         ! walk through entire flagList except last
       if (gravityNode(j)) then                          ! valid node index
         do k = j+1,mesh_maxNnodes+mesh_maxNsubNodes     ! walk through remainder of list to find duplicates
           if (gravityNode(k) .and. all(abs(gravityNodePos(:,j) - gravityNodePos(:,k)) < 1.0e-100_pReal)) then
             gravityNode(j) = .false.                    ! delete first instance
             gravityNodePos(:,j) = 0.0_pReal
             exit                                        ! continue with next suspect
           endif
         enddo
       endif
     enddo
     centerOfGravity = sum(gravityNodePos,2)/count(gravityNode)

     do f = 1,FE_NipNeighbors(t)          ! loop over interfaces of IP and add tetrahedra which connect to CoG
       forall (n = 1:FE_NipFaceNodes) nPos(:,n) = mesh_subNodeCoord(:,FE_subNodeOnIPFace(n,f,i,t),e)
       forall (n = 1:FE_NipFaceNodes, j = 1:Ntriangles) &
                                    ! start at each interface node and build valid triangles to cover interface
         volume(j,n) = math_volTetrahedron(nPos(:,n), &     ! calc volume of respective tetrahedron to CoG
                                           nPos(:,1+mod(n+j-1,FE_NipFaceNodes)), &
                                           nPos(:,1+mod(n+j-0,FE_NipFaceNodes)), &
                                           centerOfGravity)
       mesh_ipVolume(i,e) = mesh_ipVolume(i,e) + sum(volume)    ! add contribution from this interface
     enddo
     mesh_ipVolume(i,e) = mesh_ipVolume(i,e) / FE_NipFaceNodes ! renormalize with interfaceNodeNum due
     mesh_ipCenterOfGravity(:,i,e) = centerOfGravity                                  ! to loop over them
   enddo
 enddo
 return

 endsubroutine

!************************************************************
! calculation of IP interface areas
!
! allocate globals
! _ipArea, _ipAreaNormal
!************************************************************
 subroutine mesh_build_ipAreas()

 use prec, only: pInt,pReal
 use math
 implicit none

 integer(pInt) e,f,t,i,j,n
 integer(pInt), parameter :: Ntriangles = FE_NipFaceNodes-2     ! each interface is made up of this many triangles
 real(pReal), dimension (3,FE_NipFaceNodes) :: nPos             ! coordinates of nodes on IP face
```

```
    real(pReal), dimension(3,Ntriangles,FE_NipFaceNodes) :: normal
890 real(pReal), dimension(Ntriangles,FE_NipFaceNodes)   :: area

    allocate(mesh_ipArea(mesh_maxNipNeighbors,mesh_maxNips,mesh_NcpElems)) ;          mesh_ipArea       = 0.0_pReal
    allocate(mesh_ipAreaNormal(3,mesh_maxNipNeighbors,mesh_maxNips,mesh_NcpElems)) ; mesh_ipAreaNormal = 0.0_pReal

    do e = 1,mesh_NcpElems                                          ! loop over cpElems
      t = mesh_element(2,e)                                         ! get elemType
      do i = 1,FE_Nips(t)                                           ! loop over IPs of elem
        do f = 1,FE_NipNeighbors(t)                                 ! loop over interfaces of IP
          forall (n = 1:FE_NipFaceNodes) nPos(:,n) = mesh_subNodeCoord(:,FE_subNodeOnIPFace(n,f,i,t),e)
900       forall (n = 1:FE_NipFaceNodes, j = 1:Ntriangles)
                                  ! start at each interface node and build valid triangles to cover interface
            normal(:,j,n)=math_vectorproduct(nPos(:,1+mod(n+j-1,FE_NipFaceNodes))-nPos(:,n),&
                                             nPos(:,1+mod(n+j-0,FE_NipFaceNodes))-nPos(:,n)) ! calc their normal vectors
            area(j,n) = dsqrt(sum(normal(:,j,n)*normal(:,j,n)))                              ! and area
          end forall
          forall (n = 1:FE_NipFaceNodes, j = 1:Ntriangles, area(j,n) > 0.0_pReal) &
            normal(:,j,n) = normal(:,j,n) / area(j,n)               ! make unit normal

          mesh_ipArea(f,i,e) = sum(area) / (FE_NipFaceNodes*2.0_pReal)   ! area of parallelograms instead of triangles
910       mesh_ipAreaNormal(:,f,i,e) = sum(sum(normal,3),2) / count(area > 0.0_pReal)   ! average of all valid normals
        enddo
      enddo
    enddo
    return

    endsubroutine

    !*************************************************************
    ! write statistics regarding input file parsing
920 ! to the output file
    !
    !*************************************************************
    subroutine mesh_tell_statistics()

    use prec, only: pInt
    use math, only: math_range
    use IO, only: IO_error
    use debug, only: verboseDebugger

930 implicit none

    integer(pInt), dimension (:,:), allocatable :: mesh_HomogMicro
    character(len=64) fmt
    integer(pInt) i,e,n,f,t

    if (mesh_maxValStateVar(1) < 1_pInt) call IO_error(110) ! no homogenization specified
    if (mesh_maxValStateVar(2) < 1_pInt) call IO_error(120) ! no microstructure specified

    allocate (mesh_HomogMicro(mesh_maxValStateVar(1),mesh_maxValStateVar(2))); mesh_HomogMicro = 0_pInt
940 do e = 1,mesh_NcpElems
      if (mesh_element(3,e) < 1_pInt) call IO_error(110,e) ! no homogenization specified
      if (mesh_element(4,e) < 1_pInt) call IO_error(120,e) ! no homogenization specified
      mesh_HomogMicro(mesh_element(3,e),mesh_element(4,e)) = &
      mesh_HomogMicro(mesh_element(3,e),mesh_element(4,e))+1 ! count combinations of homog. and microstructure
    enddo

    if (verboseDebugger) then
     !$OMP CRITICAL (write2out)
      write(6,*)
950   write(6,*) 'Input Parser: IP COORDINATES'
      write(6,'(a5,x,a5,3(x,a12))') 'elem','IP','x','y','z'
      do e = 1,mesh_NcpElems
        do i = 1,FE_Nips(mesh_element(2,e))
          write (6,'(i5,x,i5,3(x,f12.8))') e, i, mesh_ipCenterOfGravity(:,i,e)
        enddo
      enddo
      write(6,*)
      write(6,*) "Input Parser: IP NEIGHBORHOOD"
      write(6,*)
960   write(6,"(a10,x,a10,x,a10,x,a3,x,a13,x,a13)") "elem","IP","neighbor","","elemNeighbor","ipNeighbor"
      do e = 1,mesh_NcpElems                    ! loop over cpElems
        t = mesh_element(2,e)                   ! get elemType
        do i = 1,FE_Nips(t)                     ! loop over IPs of elem
          do n = 1,FE_NipNeighbors(t)           ! loop over neighbors of IP
            write (6,"(i10,x,i10,x,i10,x,a3,x,i13,x,i13)") e,i,n,'-->',&
                                                  mesh_ipNeighborhood(1,n,i,e),mesh_ipNeighborhood(2,n,i,e)
          enddo
        enddo
      enddo
```

```
   write (6,*)
    write (6,*) "Input Parser: ELEMENT VOLUME"
    write (6,*)
    write (6,"(a13,x,e15.8)") "total volume", sum(mesh_ipVolume)
    write (6,*)
    write (6,"(a5,x,a5,x,a15,x,a5,x,a15,x,a16)") "elem","IP","volume","face","area","--- normal ---"
    do e = 1,mesh_NcpElems
      do i = 1,FE_Nips(mesh_element(2,e))
        write (6,"(i5,x,i5,x,e15.8)") e,i,mesh_IPvolume(i,e)
        do f = 1,FE_NipNeighbors(mesh_element(2,e))
          write (6,"(i33,x,e15.8,x,3(f6.3,x))") f,mesh_ipArea(f,i,e),mesh_ipAreaNormal(:,f,i,e)
        enddo
      enddo
    enddo
    write (6,*)
   write (6,*) "Input Parser: SUBNODE COORDINATES"
   write (6,*)
   write(6,'(a5,x,a5,x,a15,x,a15,x,a20,3(x,a12))')'elem','IP','IP neighbor','IPFaceNodes','&
                                                                                    subNodeOnIPFace','x','y','z'
   do e = 1,mesh_NcpElems                        ! loop over cpElems
     t = mesh_element(2,e)                       ! get elemType
     do i = 1,FE_Nips(t)                         ! loop over IPs of elem
       do f = 1,FE_NipNeighbors(t)               ! loop over interfaces of IP
         do n = 1,FE_NipFaceNodes                ! loop over nodes on interface
           write(6,'(i5,x,i5,x,i15,x,i15,x,i20,3(x,f12.8))') e,i,f,n,FE_subNodeOnIPFace(n,f,i,t),&
                                                    mesh_subNodeCoord(1,FE_subNodeOnIPFace(n,f,i,t),e),&
                                                    mesh_subNodeCoord(2,FE_subNodeOnIPFace(n,f,i,t),e),&
                                                    mesh_subNodeCoord(3,FE_subNodeOnIPFace(n,f,i,t),e)

         enddo
       enddo
     enddo
   enddo
  !$OMP END CRITICAL (write2out)
 endif


 !$OMP CRITICAL (write2out)
 write (6,*)
 write (6,*) "Input Parser: STATISTICS"
 write (6,*)
 write (6,*) mesh_Nelems,           " : total number of elements in mesh"
 write (6,*) mesh_NcpElems,         " : total number of CP elements in mesh"
 write (6,*) mesh_Nnodes,           " : total number of nodes in mesh"
 write (6,*) mesh_maxNnodes,        " : max number of nodes in any CP element"
 write (6,*) mesh_maxNips,          " : max number of IPs in any CP element"
 write (6,*) mesh_maxNipNeighbors,  " : max number of IP neighbors in any CP element"
 write (6,*) mesh_maxNsubNodes,     " : max number of (additional) subnodes in any CP element"
 write (6,*) mesh_maxNsharedElems,  " : max number of CP elements sharing a node"
 write (6,*)
 write (6,*) "Input Parser: HOMOGENIZATION/MICROSTRUCTURE"
 write (6,*)
 write (6,*) mesh_maxValStateVar(1), " : maximum homogenization index"
 write (6,*) mesh_maxValStateVar(2), " : maximum microstructure index"
 write (6,*)
 write (fmt,"(a,i5,a)") "(9(x),a2,x,",mesh_maxValStateVar(2),"(i8))"
 write (6,fmt) "+-",math_range(mesh_maxValStateVar(2))
 write (fmt,"(a,i5,a)") "(i8,x,a2,x,",mesh_maxValStateVar(2),"(i8))"
 do i=1,mesh_maxValStateVar(1)       ! loop over all (possibly assigned) homogenizations
   write (6,fmt) i,"| ",mesh_HomogMicro(i,:) ! loop over all (possibly assigned) microstrcutures
 enddo
 write (6,*)
 call flush(6)
 !$OMP END CRITICAL (write2out)

 deallocate(mesh_HomogMicro)
 return

 endsubroutine

 END MODULE mesh
```

Listing A.4: `makefile`

```
cpspectral.out: mpie_spectral.o CPFEM.a
	ifort -o cpspectral.out mpie_spectral.o CPFEM.a libfftw3_threads.a\
	libfftw3.a constitutive.a advanced.a basics.a -lpthread

mpie_spectral.o: mpie_spectral.f90 CPFEM.o
	ifort -c -O3 -heap-arrays 500000000 mpie_spectral.f90

CPFEM.a: CPFEM.o
	ar rc CPFEM.a homogenization.o homogenization_RGC.o homogenization_isostrain.o\
	crystallite.o CPFEM.o constitutive.o

CPFEM.o: CPFEM.f90 homogenization.o
	ifort -c -O3 -heap-arrays 500000000 CPFEM.f90
homogenization.o: homogenization.f90 homogenization_isostrain.o homogenization_RGC.o crystallite.o
	ifort -c -O3 -heap-arrays 500000000 homogenization.f90
homogenization_RGC.o: homogenization_RGC.f90 constitutive.a
	ifort -c -O3 -heap-arrays 500000000 homogenization_RGC.f90
homogenization_isostrain.o: homogenization_isostrain.f90 basics.a advanced.a
	ifort -c -O3 -heap-arrays 500000000 homogenization_isostrain.f90
crystallite.o: crystallite.f90 constitutive.a
	ifort -c -O3 -heap-arrays 500000000 crystallite.f90


constitutive.a: constitutive.o
	ar rc constitutive.a constitutive.o constitutive_titanmod.o constitutive_nonlocal.o\
	constitutive_dislotwin.o constitutive_j2.o constitutive_phenopowerlaw.o basics.a advanced.a

constitutive.o: constitutive.f90 constitutive_titanmod.o constitutive_nonlocal.o\
	constitutive_dislotwin.o constitutive_j2.o constitutive_phenopowerlaw.o
	ifort -c -O3 -heap-arrays 500000000 constitutive.f90
constitutive_titanmod.o: constitutive_titanmod.f90 basics.a advanced.a
	ifort -c -O3 -heap-arrays 500000000 constitutive_titanmod.f90
constitutive_nonlocal.o: constitutive_nonlocal.f90 basics.a advanced.a
	ifort -c -O3 -heap-arrays 500000000 constitutive_nonlocal.f90
constitutive_dislotwin.o: constitutive_dislotwin.f90 basics.a advanced.a
	ifort -c -O3 -heap-arrays 500000000 constitutive_dislotwin.f90
constitutive_j2.o: constitutive_j2.f90 basics.a advanced.a
	ifort -c -O3 -heap-arrays 500000000 constitutive_j2.f90
constitutive_phenopowerlaw.o: constitutive_phenopowerlaw.f90 basics.a advanced.a
	ifort -c -O3 -heap-arrays 500000000 constitutive_phenopowerlaw.f90

advanced.a: lattice.o
	ar rc advanced.a FEsolving.o mesh.o material.o lattice.o

lattice.o: lattice.f90 material.o
	ifort -c -O3 -heap-arrays 500000000 lattice.f90
material.o: material.f90 mesh.o
	ifort -c -O3 -heap-arrays 500000000 material.f90
mesh.o: mesh.f90 FEsolving.o
	ifort -c -O3 -heap-arrays 500000000 mesh.f90
FEsolving.o: FEsolving.f90 basics.a
	ifort -c -O3 -heap-arrays 500000000 FEsolving.f90

basics.a: debug.o math.o
	ar rc basics.a debug.o math.o numerics.o IO.o mpie_spectral_interface.o prec.o

debug.o: debug.f90 numerics.o
	ifort -c -O3 debug.f90
math.o: math.f90 numerics.o
	ifort -c -O3 math.f90
numerics.o: numerics.f90 IO.o
	ifort -c -O3 numerics.f90
IO.o: IO.f90 mpie_spectral_interface.o
	ifort -c -O3 IO.f90
mpie_spectral_interface.o: mpie_spectral_interface.f90 prec.o
	ifort -c -O3 mpie_spectral_interface.f90
prec.o: prec.f90
	ifort -c -O3 prec.f90
```

B Problem set-up example files

Listing B.1: `3x3x3x5.geom`

```
resolution  a 3    b 3    c 3
dimension   x 1.5  y 1.0  z 2.0
homogenization  1
1
1
3
1
2
3
4
4
4
5
5
1
1
2
2
4
2
2
1
3
3
1
3
3
5
5
2
```

Listing B.2: `example_loadcase.load`

```
velocitygrad # .0 .0 .0 # .0 .0 .0 1. stress .0 # # # .0 # # # # time 0.0004 steps 40
l .0 .0 .0 .0 -1.5230484e-4 -0.0174524 .0 0.0174524 -1.5230484e-4 s # # # # # # # # # t 90. n 90
velocitygrad 0 .0 .0 .0 # .0 .0 .0 -.001 s # # # # .0 # # # # t 300. incs 300
velocitygrad # .0 .0 .0 # .0 .0 .0 .001 s .0 # # # .0 # # # # delta 300. increments 300
```

Listing B.3: `material.config`

```
#-------------------#
<homogenization>
#-------------------#

[SX]
type     isostrain
Ngrains 1

#-------------------#
<microstructure>
#-------------------#

[Grain001]                                          # microstructure 1
crystallite     1
(constituent)  phase 1   texture   1   fraction 1.0    # one constituent with fraction = 100%
[Grain002]                                          # microstructure 2
crystallite     1
(constituent)  phase 1   texture   2   fraction 1.0
[Grain003]
crystallite     1
(constituent)  phase 1   texture   3   fraction 1.0
[Grain004]
crystallite     1
(constituent)  phase 2   texture   4   fraction 1.0
[Grain005]
crystallite     1
(constituent)  phase 2   texture   5   fraction 1.0

#-------------------#
<crystallite>
#-------------------#

[all]
(output) phase
(output) volume
(output) orientation
(output) eulerangles
(output) grainrotation  # deviation from initial orientation as axis (1-3) and angle in degree (4)
(output) f              # deformation gradient tensor; synonyms: "defgrad"
(output) fe             # elastic deformation gradient tensor
(output) fp             # plastic deformation gradient tensor
(output) ee             # elastic strain as Green-Lagrange tensor
(output) p              # first Piola-Kichhoff stress tensor; synonyms: "firstpiola", "1stpiola"
(output) s              # second Piola-Kichhoff stress tensor; synonyms: "tstar", "secondpiola", "2ndpiola"

#-------------------#
<phase>
#-------------------#

[Aluminum_phenopowerlaw]    # phase 1, grains 1,2, and 3 consist of this phase
# slip only
constitution            phenopowerlaw           # phenopowerlaw, used for simulations 8.1 and 8.3

(output)                resistance_slip
(output)                shearrate_slip
(output)                resolvedstress_slip
(output)                totalshear
(output)                resistance_twin
(output)                shearrate_twin
(output)                resolvedstress_twin
(output)                totalvolfrac

lattice_structure       fcc
Nslip                   12  0  0  0            # per family
Ntwin                    0  0  0  0            # per family

c11                     170.17e3                # copper variables
c12                     114.92e3
c44                     60.98e3

gdot0_slip              1.0
n_slip                  10
tau0_slip               10.0                      # per family
tausat_slip             63e6                      # per family
w0_slip                 2.25
gdot0_twin              0.001
n_twin                  20
tau0_twin               31e6                    # per family
s_pr                    0                       # push-up factor for slip saturation due to twinning
```

```
twin_b                  0
twin_c                  0
twin_d                  0
twin_e                  0
h0_slipslip             0
h0_sliptwin             0
h0_twinslip             0
h0_twintwin             0
interaction_slipslip    1 1 1.4 1.4 1.4 1.4
interaction_sliptwin    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
interaction_twinslip    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
interaction_twintwin    1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
relevantResistance      1

[Aluminum_J2isotropic]                  # phase 2, grains 4 and 5 consist of this phase

constitution            j2              # J2 plasticity, used for simulation 8.2

(output)                flowstress
(output)                strainrate

c11                     110.9e9
c12                     58.34e9
taylorfactor            3
tau0                    31e6
gdot0                   0.001
n                       20
h0                      75e6
tausat                  63e6
w0                      2.25
atol_resistance         1

#-------------------#
<texture>
#-------------------#

[Grain001]      # names are arbitrary, but as each grain has its own texture the same names are used
(gauss)  phi1 359.121452   Phi  82.319471   Phi2 347.729535   scatter 0   fraction 1
[Grain002]      # texture 2
(gauss)  phi1 269.253967   Phi 105.379919   Phi2 173.029284   scatter 0   fraction 1
[Grain003]
(gauss)  phi1  26.551535   Phi 171.606752   Phi2 124.949264   scatter 0   fraction 1
[Grain004]
(gauss)  phi1 123.207774   Phi 124.339577   Phi2  47.937748   scatter 0   fraction 1
[Grain005]
(gauss)  phi1 324.188825   Phi 103.089216   Phi2 160.373624   scatter 0   fraction 1
```

C License information

The copyright of all figures in this work belongs to author or the involved organization, except of the figures mentioned below. All files were accessed on 14th November 2010.

Fig. 2.1:
File is licensed under the Creative Commons Attribution 3.0 unported license. You are free:

- to share - to copy, distribute and transmit the work
- to remix - to adapt the work

Under the following conditions:

- attribution - You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

The license can be found at: `http://creativecommons.org/licenses/by/3.0/deed.en`
Source: `http://commons.wikimedia.org/wiki/File:Continuum_body_deformation.svg`

Fig. 3.1:
File is public domain.
Source: `http://commons.wikimedia.org/wiki/File:Close_packing.svg`

Fig. 3.2 and fig. 3.2:
Copyright belongs to the uploader, Baszoetekouw, all rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, with the name of the uploader, and this list of conditions;
- Redistributions in binary form must reproduce the above copyright notice, with the name of the uploader, and this list of conditions in the documentation and/or other materials provided with the distribution;
- Neither the name of the uploader nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

Source: `http://en.wikipedia.org/wiki/File:Lattice_body_centered_cubic.svg`
Source: `http://en.wikipedia.org/wiki/File:Lattice_body_centered_cubic.svg`

List of Figures

List of Tables

Listings

Bibliography

[1] H.-J. Bargel and G. Schulze. *Werkstoffkunde*. Springer Verlag, Berlin, 7th edition, 2000.

[2] K.-J. Bathe. *Finite-Elemente-Methoden. Aus dem Englischen von Peter Zimmermann*. Springer Verlag, Berlin, 2nd edition, 2002.

[3] T. Beth. *Verfahren der schnellen* FOURIER-*Transformation*. B. G. Teubner, Stuttgart, 1984.

[4] J. P. Boyd. CHEBYSHEV *and* FOURIER *spectral methods*. Dover, New York, 2001.

[5] S. Brisard and L. Dormieux. FFT-based methods for the mechanics of composites: A general variational framework. *Comp. Mater. Sci.*, 49:663–671, 2010.

[6] C. M. Brown, W. Dreyer, and W. H. Müller. Discrete FOURIER transforms and their application to stress–strain problems in composite mechanics. A convergence study. *Proc. R. Soc. Lond. A*, 458:1967–1987, 2002.

[7] J. W. Christian and S. Mahajan. Deformation twinning. *Progress in Materials Science*, 39: 1–157, 1995.

[8] Ş. S. Bayin. *Mathematical methods in science and engineering*. Wiley-Interscience, Hoboken, 2006.

[9] O. Föllinger. LAPLACE- *und* FOURIER-*Transformation*. Hüthig, Heidelberg, 1986.

[10] T. Kanit. *Notion of representative volume element for heterogeneous materials: Statistical and numerical approach*. PhD thesis, Ecole des mines de Paris, May 2003.

[11] J. K. Knowles. On the representation of the elasticity tensor for isotropic materials. *Journal of elasticity*, 39:175–180, 1995.

[12] N. Lahellec, J. C. Michel, H. Moulinec, and P. Suquet. Analysis of inhomogeneous materials at large strains using fast FOURIER transforms. In C. Miehe, editor, *IUTAM symposium on computational mechanics of solid materials at large strains*, pages 247–258. Kluwer Academic Publishers, Dordrecht, 2001.

[13] C. Lanczos. *Linear differential operators*. D. van Nostrand company limited, London, 1961.

[14] R. A. Lebensohn. N-site modeling of a 3D viscoplastic polycrystal using fast FOURIER transforms. *Acta Materalia*, 49:2723–2737, 2001.

[15] H. Lippman. *Mechanik des plastischen Fließens*. Springer Verlag, Berlin, 1981.

[16] F. Matteo and S. G. Johnson. The design and implementation of *FFTW*3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program generation, optimization, and platform adaptation".

[17] J. C. Michel, H. Moulinec, and P. Suquet. A computational scheme for linear and non-linear composites with arbitrary phase contrast. *Int. J. Numer. Meth. Engng.*, 52:139–160, 2001.

[18] J.C. Michel, H. Moulinec, and P. Suquet. Effective properties of composite materials with periodic microstructure. A computational approach. *Comput. Methods Appl. Mech. Engrg.*, 172:109–143, 1999.

[19] H. Moulinec and P. Suquet. A numerical method for computing the overall response of nonlinear composites with complex microstructure. *Comput. Methods Appl. Mech. Engrg.*, 157:69–94, 1998.

[20] H. Moulinec and P. Suquet. Comparison of FFT-based methods for computing the response of composites with highly contrasted mechanical properties. *Physica B*, 338:58–60, 2003.

[21] T. Mura. *Micromechanics of defects in solids*. Kluwer Academic Publishers, Dordrecht, 2nd edition, 1987.

[22] S. Neumann, K. P. Herrmann, and W. H. Müller. FOURIER transforms - An alternative to finite elements for elastic–plastic stress–strain analyses of heterogeneous materials. *Acta Mechanica*, 149:149–160, 2001.

[23] T. K. Nguyen, K. Sab, and G. Bonnet. GREEN's operator for a periodic medium with traction-free boundary conditions and computation of the effective properties of thin plates. *Int. J. of Solids and Structures*, 45:6518–6534, 2008.

[24] E. L. Ortiz. The τ method. *SIAM Journal on Numerical Analysis*, 6(2):480–492, 1969.

[25] A. Prakash and R. A. Lebensohn. Simulation of micromechanical behavior of polycrystals: Finite elements versus fast FOURIER transforms. *Modelling Simul. Mater. Sci. Eng.*, 17(6), 2009.

[26] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in Fortran 77. The art of scientific computing*. Cambridge University Press, Cambridge, 2nd edition, 1992.

[27] D. Rabenstein. *Fortran 90 Lehrbuch*. Carl Hanser Verlag, München/Wien, 1995.

[28] F. Roters. *Advanced material models for the crystal plasticity finite element method*. Handed in November 2010, presumably published in 2011. Habilitationsschrift RWTH Aachen.

[29] F. Roters, P. Eisenlohr, T. R. Bieler, and D. Raabe. *Crystal plasticity finite element methods in material science and engineering*. Wiley-VCH, Weinheim, 2010.

[30] A.A. Salem, S.R. Kalidindi, and S.L. Semiatin. Strain hardening due to deformation twinning in α-titanium: Constitutive relations and crystal-plasticity modeling. *Acta Materialia*, 53 (12):3495–3502, 2005.

[31] R. Sedláček. *Finite Elemente in der Werkstoffmechanik*. Verlag Dr. Hut, München, 2009.

[32] L. N. Trefethen. *Finite difference and spectral methods for ordinary and partial differential equations*. Unpublished text, 1996. URL `http://web.comlab.ox.ac.uk/people/Nick.Trefethen/pdetext.html`.

[33] V. Vinogradov and G. W. Milton. An accelerated FFT algorithm for thermoelastic and nonlinear composites. *Int. J. Numer. Meth. Engng.*, 67(11):1678–1695, 2009.

[34] W. A. Wall and B. Bornemann. *Vorlesungsbegleitendes Skript „Grundlagenfach Mechanik: Nichtlineare Kontinuumsmechanik“*. TU München, WS 2007.

[35] W. A. Wall, L. Wiechert, and S. Tinkl. *Vorlesungsbegleitendes Skript „Nichtlineare Finite-Element-Methoden“*. TU München, SS 2009.

[36] E. Werner. *Vorlesungsbegleitendes Skript „Werkstoffkunde 1“*. TU München, WS 2007.

[37] X. Wu, S. R. Kalidindi, C. Necker, and A. A. Salem. Prediction of crystallographic texture evolution and anisotropic stress–strain curves during large plastic strains in high purity α-titanium using a taylor-type crystal plasticity model. *Acta Materialia*, 55(2):423–432, 2007.