Technische Universität Graz
Institut für Festigkeitslehre
Kopernikusgasse 24/I
8010 Graz

Juraj Raič

Variational-Based Phase Field Modelling of Fracture

Masterarbeit
zur Erlangung des akademischen Grades
eines Diplom–Ingenieurs

Technische Universität Graz
Fakultät für Maschinenbau und Wirtschaftswissenschaften

Studienrichtung:
Maschinenbau

Betreuer und Beurteiler: O.Univ.–Prof. Dipl.–Ing. Dr.techn. Christian C. Celigoj

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Danksagung

Mein aufrichtiger Dank gilt Professor Christian Celigoj für die unterstützende und motivierende Betreuung, als auch die fachlichen, impulsgebenden Diskurse. Besonders möchte ich mich beim Institut für Festigkeitslehre der Technischen Universität Graz und dessen Mitarbeitern bedanken, allen voran bei Manfred Ulz, Benedikt Weger und Patrick Wurm, die stets produktive Diskussionen anregten und förderten.

Innigster Dank gilt all jenen, die an mich geglaubt haben, meinen Verwandten und Freunden, wobei ich an dieser Stelle meine Familie hervorheben möchte. Meine Eltern Barbara und Božo, meine Schwester Sara und meine Brüder Grgur und Niko Jeronim sind für mich allzeit währende Quellen der Motivation, Inspiration und Freude.

Juraj Raič, Graz, am 28.01.2018

Abstract

The goal of this thesis is to provide a numerical implementation of the variational-based phase field modelling of fracture, in the context of the gradient extended theory for dissipative continua. Classical continuum mechanics does not provide an adequate basis for the description of materials with microstructure. Hence, a multi-field formulation is adopted, which considers a macroscopic and a microscopic deformation field, as well as a dissipative thermodynamic force, accounting for the second law of thermodynamics. Consequently the theory of internal variables, which usually takes into account temporal derivatives, is extended by the geometrical gradient of the phase field, thus including nonlocal information. Hence, a thermodynamically consistent model is derived, applications of which can include not only damage and fracture modelling, but also plasticity. Regarding diffusive crack propagation, a time-discrete incremental variational principle is derived, based on the balance of the isotropic degradation of the stored bulk energy, the dissipation function and the external forces, illustrating the behaviour of the material for small deformations. To account for inelastic loading, the dissipation function is extended by a rate-dependent over-force, representing a viscous regularization of the rate-independent formulation. The theory is embedded into a numerical example, which is solved by way of using the finite element method.

Kurzfassung

In vorliegender Arbeit wird eine numerische Implementierung der variationsbasierenden Feldmodellierung von Rissen, im Kontext der gradientenerweiterten Theorie für dissipative Kontinua angestrebt. Materialien mit Mikrostrukturen werden von der klassischen Kontinuumsmechanik, welche lediglich makroskopische Verformungen zu beschreiben vermag, nicht erfasst. Abhilfe verschafft eine Mehrfeldfomulierung, unter Einbindung sowohl makroskopischer, als auch mikroskopischer Verformungsfelder, sowie eines dissipativen, thermodynamischen Kraftfeldes, welches im Einklang mit dem zweiten Hauptsatz der Thermodynamik steht. Infolgedessen werden interne Variablen, nicht nur einer zeitlichen Ableitung unterzogen, sondern um einen Term erweitert, der den geometrischen Gradienten berücksichtigt, um auch nichtlokale Informationen einzubetten. Folglich entsteht ein thermodynamisch konsistentes Modell, das sowohl zur Abbildung schadensmechanischer und bruchmechanischer Prozesse, als auch von Plastizität, verwendet werden kann. Für den Fall der diffusen Rissbildung wird auf Basis des inkrementellen, variationellen Ansatzes der Summe aus isotroper Abnahme der freien Energie im Kontinuum, der Dissipation und der externen Kräfte, ein stationäres Prinzip hergeleitet, welches das Materialverhalten beim Auftreten kleiner Verformungen wiederspiegelt. Hierfür wird die Dissipationsfunktion um eine Schwellfunktion erweitert, unter Einbettung der Viskosität, die inelastische Verformungen abbildet. Die numerische Implementierung der Theorie erfolgt an Hand eines Beispieles, unter Verwendung der Methode der finiten Elemente.

Contents

List of Figures

List of Tables

1. Motivation and Introductory Remarks

1.1. Motivation

Motivated by the will of applying theoretical postulates of continuum mechanics in a numerical environment, this thesis is dediacted to the gradient-extended theory for dissipative continua with microstructure. Two major modifications to the classical continuum mechanics, as introduced by Cauchy in the 19^{th} century, are emphasized in this theory. First, the introduction of dissipative thermodynamics, initially proposed in the 1960s by Coleman and Gurtin [9]. This approach is modelled by internal state variables, which unfortunately are defined locally and only have to satisfy the second law of thermodynamics, without resulting in additional balance equations or taking into account microstructural effects [37]. Contrary to that the second modification to the classical theory represents an extension by generalized state variables. Defined globally, they do not only take into account a macroscopic, but also a microscopic deformation field and a dual thermodynamic force, and when subjected to a variational principle result in additional coupled balance equations. The derivative of the microscopic deformation field extends the temporal derivative, by including the gradient, therefore attributing it with nonlocal information of the geometry itself.
Throughout the years a multi-field formulation, which is reiterated in this thesis, was gradually developed by Mielke and Roubiček [31], in the numerous contributions by Welschinger and Miehe [36], as well as Dimitrijević and Hackl [14]. In the procedure put forth by Welschinger and Miehe the total power is derived, balancing the internal and external power. Subjected to a time-discretization, upon which a variational principle is performed, the results manifest themselves in coupled Euler equations. Due to the global character of the formulation, the method can be applied to a variety of problems, such as damage and fracture modelling, as well as plasticity. With regard to crack propagation the microstructural field variable is taken as a fracture phase field, and constructed using Griffith's fracture criterion [15]. The dissipation is regularized by a threshold function, which depending on the rates of change of the crack phase field and the thermodynamic force, renders an elastic or an inelastic deformation. The isotropic stored bulk energy is treated with a degradation function and the calculation is performed for small deformations.

1.2. Additional Considerations

To gain a better understanding of the advances in continuum mechanics, which will be applied in this thesis, one has to first consider the linear elasticity theory, established by Cauchy. The underlying problem of continuum mechanics is to describe the deformation of a continuum body under external forces. To this end, the stresses $\boldsymbol{\sigma}$ and the displacements $\boldsymbol{u}$ in every point of the continuum have to be known. Since there is no direct correlation between those two quantities a so called *constitutive law*, between the stresses $\boldsymbol{\sigma}$ and the strains $\boldsymbol{\varepsilon}$, has to be introduced. The strains are an additional, mathematical auxiliary quantity, linked to the displacements via kinematic relations. The aforementioned correlations are depicted in Figure 1.1. In three dimensional space a total of 15 equations have to be derived and solved, 6 stress-strain equations and 6 strain-displacement equations, as well as additional 3 equilibrium equations.
Extensive documentation on the topic of linear continuum mechanics is found in the lecture notes by Celigoj [11], as well as the introductory chapters to the finite element handbook by Hammer [17]. Wide-ranging material on the topic of linear and nonlinear continuum mechanics was contributed by Bonet and Wood [4], Truesdell and Noll [33], Dimitrienko [13], Haupt [18], Iben [20] and Maugin [25].

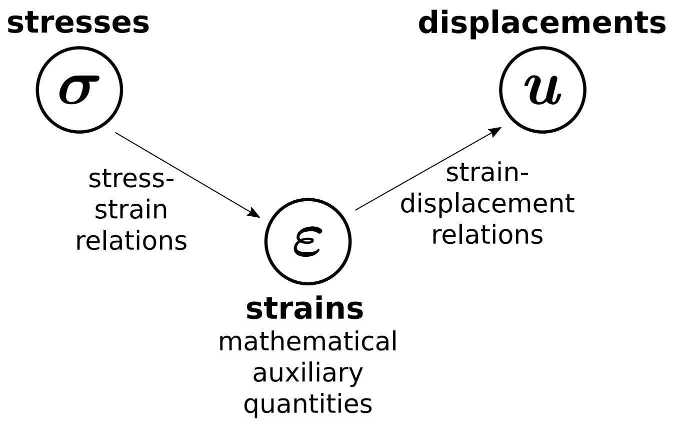

stresses
$\boldsymbol{\sigma}$
displacements
$\boldsymbol{u}$
stress-strain relations
$\boldsymbol{\varepsilon}$
strain-displacement relations
strains
mathematical auxiliary quantities

Figure 1.1. Stress-strain-displacement correlation.

1.3. Overview

Chapter 2 introduces the theory for variational-based phase field modelling of fracture for the small deformation case. The balance of total power is derived and a time-discrete potential introduced which is subjected to a variational principle. The chapter concludes with a finite element discretization, which is elaboratedmore precisely in Chapter 3. Chapter 4 is dedicated to the numerical implementation, discussing the results for a classical example. The concluding chapter includes the `C++` code.

2. Variational-Based Phase Field Modelling of Fracture

The theory for the variational-based phase field modelling of fracture, is based on generalized continuum mechanics of gradient-type dissipative continua. As suggested, two major modifications to the classical theory of continuum mechanics are postulated, both necessary for an adequate description of deformation processes in continua with microstructure. First, the theory now incorporates dissipative effects, making it thermodynamically consistent and second, the theory of internal variables is extended by so called generalized state variables, which are not only defined locally, but also include geometrical information, specific to their neighbourhood, by way of a gradient-type extension, regulated by length scale parameters. Up until this point only one field variable, namely the macroscopic displacement field $\boldsymbol{u}$ was considered. The modified theory suggests a complementation by microscopic field variables, accounting for microstructural effects, but defined globally, enabling not only a more convenient numerical implementation, but also wide ranging applications, such as damage and fracture modelling, as well as plasticity. The underlying theory, as summarized in the Ph.D thesis by Welschinger [37], the papers by Miehe [29], [30] and their joint contributions with Hofacker [26], [28], [27], [19], [36] and Zimmermann [35], is also exemplified for damage processes by Mielke and Roubiček [31], Dimitrijević and Hackl [14] and Waffenschmidt et al. [34]. Some of the basic semantics for continua with microstructure are found in the papers and textbooks by Capriz [6], [7], [8]. Of importance for the understanding of internal variables are also the treatises and papers by Maugin and Morro found in their separate and joint contributions [21], [22], [23], [24], as well as in the recent monograph [25]. The theoretical framework shall now be applied towards the modelling of crack propagation.

2.1. First and Second Law of Thermodynamics

In solid mechanics the balance of energy is derived from the first law of thermodynamics for a quasi-static thermodynamic process, which postulates for the change in total energy dE_{tot} of a system

$$dE_{tot} = \underbrace{dU \; + \; dE_{kin}}_{stored\ energies} = \underbrace{\delta W_{ext} \; + \; \delta Q_{ext}}_{transported\ energies} \; . \tag{2.1}$$

The changes in the stored energies are equal to the changes in the transported energies, or in other words, the change in internal energy dU plus the change in kinetic energy dE_{kin} equals the change in external, mechanical work δW_{ext} plus the

heat transfer over the boundary of the system δQ_{ext}. The symbol d denotes the infintesimal change of a state function, whereas δ denotes the infintesimal change of a process function. This type of symbolic differentiation of state and process functions can be traced back to Carl Gottfried Neumann. State functions describe the thermodynamical euqilibrium state of a system and are path independent. The value of the functions only depends on the initial state and the present state of a system, not on the path taken to reach those states. Process functions on the other hand are path dependent quantities.
A process is called quasi-static, if the system is in equilibrium at any given moment, an assumption which is granted, when the process is slower than the realxation time. The second law of thermodynamics introduces a new state function called entropy, which is defined as

$$dS := \frac{\delta Q_{rev}}{T} = \frac{\delta Q_{ext} + \delta Q_{fric}}{T}\,. \tag{2.2}$$

In a closed (no mass transport over the system boundary permitted, $dm = 0$), adiabatic (no heat transfer over the system boundary permitted, $\delta Q_{ext} = 0$) system the reversible heat δQ_{rev} equals the friction heat δQ_{fric} of a dissipative process. Without the presence of external energies dE the following is obtained

$$\delta Q_{rev} = \delta Q_{fric} = dU - \delta W_{ext}\,. \tag{2.3}$$

For a reversible process the friction heat becomes zero, for the irreversible process it is always positive. This leads to the defintion of the second law of thermodynamics which states that in a closed, adiabatic system the entropy will always increase, only for the special case of a reversible process it will stay the same

$$dS \geq 0\,. \tag{2.4}$$

It is important to note that any reversible process, is in fact quasi-static, whereas not every quasi-static process is reversible, since there exist irreversible, quasi-static processes. An example of a reversible process would be any elastic deformation, while inelastic behaviour is an indicator of plasticity and as a consequence irreversibilty.

2.2. Griffith's Fracture Criterion

In order to capture the nature of processes with crack propagation the first law of thermodynamics deduced in Section 2.1 has to be extended by an additional energetic term, taking into account the irreversibilty of the molecular bond breaking process. Taking the time derivative of (2.1), yields the total change in power as the balance of the change in internal power with the mechanical plus thermal external power

$$\underbrace{\frac{\mathrm{d}E_{tot}}{\mathrm{d}t}}_{\mathcal{E}_{tot}} = \underbrace{\frac{\mathrm{d}U}{\mathrm{d}t} + \frac{\mathrm{d}E_{kin}}{\mathrm{d}t}}_{P_{int}} = \underbrace{\frac{\delta W_{ext}}{\mathrm{d}t}}_{P_{ext}} + \underbrace{\frac{\delta Q_{ext}}{\mathrm{d}t}}_{\dot{Q}_{ext}}\,. \tag{2.5}$$

At this point the internal powers are extended by a term $\mathcal{T}$, which summarizes all processes related to crack propagation, with the main focus on microscopic plastic deformations and other chemical or electro-magnetical processes, as sources of undefined energies

$$\frac{\mathrm{d}E_{tot}}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}[U + E_{kin} + \mathcal{T}] = P_{ext} + \dot{Q}_{ext}\,. \tag{2.6}$$

The irreversibilty of such processes is captured by

$$\dot{\mathcal{T}} \geq 0\,. \tag{2.7}$$

Since fracture only occurs in a process zone A_{pro}, whose volume is small in comparison to the rest of the material body, the predominant power term

$$\dot{\mathcal{T}} =: -\mathcal{P}_{pro}\,, \tag{2.8}$$

is split from the internal power and accounted towards the external power

$$\frac{\mathrm{d}}{\mathrm{d}t}[U + E_{kin}] = P_{ext} + \dot{Q}_{ext} + \mathcal{P}_{pro}\,. \tag{2.9}$$

$\mathcal{P}_{pro}$ describes the energy transport into the process zone and the overall effect of the continuum on the process zones. It is defined in terms of the traction $\boldsymbol{t}$

$$\mathcal{P}_{pro} := \int\limits_{A_{pro}} \boldsymbol{t} \cdot \dot{\boldsymbol{u}}\;\mathrm{d}A\,. \tag{2.10}$$

In processes accompanied by fracture, new surface is created inside the crack domain Γ. The unloading process between to states t_I and t_{II} for an infinitesimally small time step $\mathrm{d}t$, is characterized by the creation of the crack surface $\mathrm{d}\Gamma$. This process will continue until the stress free state at $\boldsymbol{t} = \boldsymbol{0}$ attained. The work flow towards the process zone $\mathrm{d}W_{pro}$ is defined along both sides of the surface area $\mathrm{d}\Gamma^{\pm}$

$$\mathrm{d}W_{pro} = \mathcal{P}_{pro}\;\mathrm{d}t = \int\limits_{\mathrm{d}\Gamma^{\pm}} \int\limits_{I}^{II} \boldsymbol{t} \cdot \mathrm{d}\dot{\boldsymbol{u}}\;\mathrm{d}A\,. \tag{2.11}$$

The creation of $\mathrm{d}\Gamma$ is proportional to the change in fracture energy $\mathrm{d}\mathcal{T}$. Once the stress free state at t_{II} is reached, the energy along the fracture surface $\mathrm{d}\Gamma^{\pm}$ is defined in terms of the *specific fracture surface energy* γ as

$$\mathrm{d}\mathcal{T} = \dot{\mathcal{T}}\;\mathrm{d}t = 2\gamma\;\mathrm{d}\Gamma\,. \tag{2.12}$$

Note the dimension of γ. Since the differential term $\mathrm{d}\mathcal{T}$ has the dimension of an energy, γ has the dimension of energy per surface area, or of a force per length. The factor γ can be proportional to the crack lengthening, but in general it is chosen to be constant. Crack propagation, i.e. the motion of the process zone $\mathrm{d}\Gamma$ is always accompanied by the transformation of fracture energy $\mathrm{d}\mathcal{T}$ into other energy forms, such as thermal or surface energy. For the elastic material, subjected to a quasi-static fracture process, the process zone is identified as a plastic domain surrounding the crack tip, where inelastic processes ocur. Therefore, the fracture energy $\mathcal{T}$ compromises now both, the energy necessary for the separation process, as well as for the inelastic deformation process. The kinectic energy $\mathrm{d}E_{kin}$ has no influence in a quasi-static process, and non-mechanical loading Q is omitted as well. The inner energy term U is replaced by the strain energy Π_{int}, while the external mechanical power P_{ext} is assumed to carry a potential Π_{ext}. The equation

$$\mathrm{d}E_{tot} = \mathrm{d}[U + \mathcal{T}] = \mathrm{d}W_{ext}\,, \tag{2.13}$$

transforms with $\mathrm{d}U = \mathrm{d}\Pi_{int}$ and $\mathrm{d}W_{ext} = -\mathrm{d}\Pi_{ext}$, and the total potential $\Pi = \Pi_{int} + \Pi_{ext}$ into

$$\mathrm{d}\Pi_{int} + \mathrm{d}\Pi_{ext} + \mathrm{d}\mathcal{T} = 0\,, \tag{2.14a}$$

$$\text{or} \quad \frac{\mathrm{d}\Pi}{\mathrm{d}\Gamma} + \frac{\mathrm{d}\mathcal{T}}{\mathrm{d}\Gamma} = 0\,. \tag{2.14b}$$

Thus it can be deduced, that in a fracture process the change of the sum of the total potential and fracture energy $\mathrm{d}(\Pi + \mathcal{T})$ with respect to the fracture surface $\mathrm{d}\Gamma$ is zero. Equation (2.14*b*) was first established by Griffith [15] and is thereafter called *Griffith's fracture criterion.* An extensive review of said criterion can be found in Gross and Seelig [16]. Introducing the *energy release rate* as

$$g = -\frac{\mathrm{d}\Pi}{\mathrm{d}\Gamma}\,, \tag{2.15}$$

and the *critical energy release* rate as $g_c = 2\gamma$, which coincides with $\mathrm{d}\mathcal{T}/\mathrm{d}\Gamma$, equation (2.14*b*) yields

$$g = g_c\,. \tag{2.16}$$

Throughout the crack propagation process the energy release rate g has to be equal to the critical energy release rate g_c. Both share the dimension of the specific fracture surface energy, i.e. force per length.

2.3. Phase Field Approximation of Crack Topology. Crack Modelling

In the theory numerous models for crack modelling have been put forth, whereby two topoi, sharp and diffusive crack modelling, shall be discussed in detail for the case of an one-dimensional cracked bar with the body $\mathbb{B}$ and the cross-section Γ. The bar shall be infinitely expanded, a property captured by choosing the length as $L = [-\infty, +\infty]$, assuming the existence of a crack at the position $x = 0$. If the theory accounts only for the possibility of an unbroken state and a fully broken state, where for the fully broken state the crack surface occupies the entire cross-section Γ, it is attributed the *sharp crack topology* and described by the auxiliary field variable $d(x) \in [0, 1]$, denoted *crack phase field*

$$d(x) := \begin{cases} 1, & \text{for } x = 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.17}$$

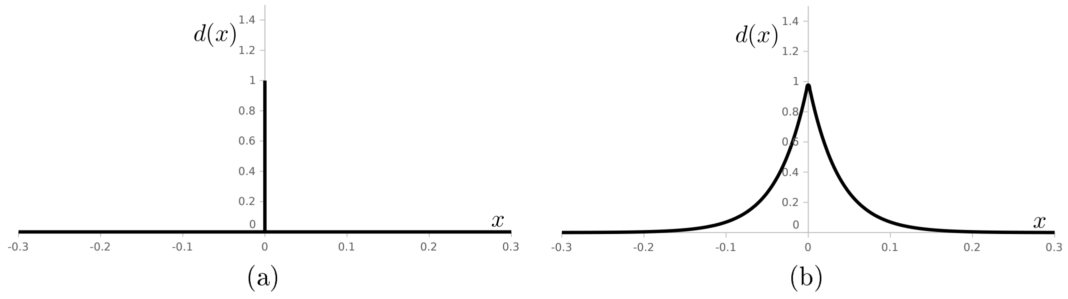


Figure 2.1. Graphic representation of the (a) sharp and (b) diffusive crack topology for a length-scale parameter $l = 0.0375\,mm$.

By approximating the aforementioned non-smooth phase field with an exponential function, a *regularized or diffusive crack topology* is introduced

$$d(x) = e^{-|x|/l} \quad \text{with} \quad d(0) = 1 \quad \text{and} \quad d(\pm\infty) = 0\,, \tag{2.18}$$

wherein the variable l is a length-scale parameter, which regulates the diffusivity of the crack and yields for $l \to 0$ the non-smooth approach (2.17). The two topoi are visibly represented in Figure 2.1. It can be shown that the diffusive crack phase field function, with the Dirichlet-type boundary conditions put forth in (2.18) emerges as the solution of the homogeneous differential equation

$$d(x) - l^2 d''(x) = 0 \quad \text{in } \mathbb{B}, \quad \text{with} \quad d(0) = 1 \quad \text{and} \quad d(\pm\infty) = 0\,, \tag{2.19}$$

which is expressed in terms of the functional

$$\Pi(d) = \frac{1}{2} \int_{\mathbb{B}} \left\{ d^2 + l^2 d'^2 \right\} \mathrm{d}V\,. \tag{2.20}$$

Setting $\mathrm{d}V = \Gamma \mathrm{d}x$ and integrating for $d(x) = e^{-|x|/l}$ results in the *crack surface functional*, representing the crack surface itself

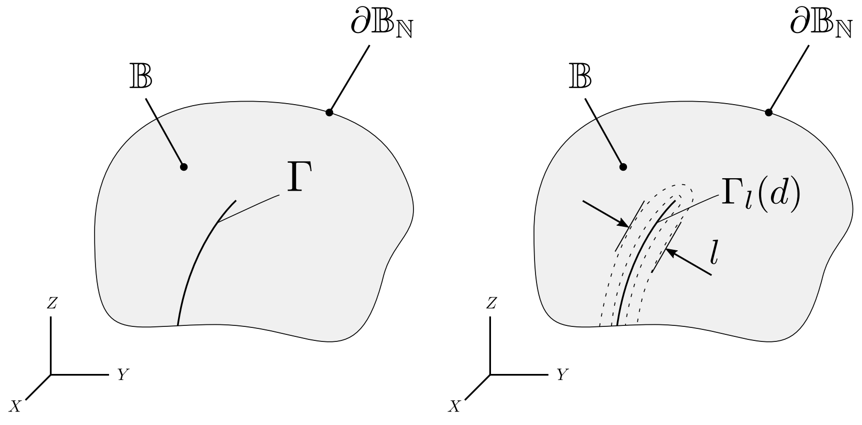
$$\Gamma_l(d) := \frac{1}{l}\Pi(d) = \frac{1}{2l} \int_{\mathbb{B}} \left\{ d^2 + l^2 d'^2 \right\} \mathrm{d}V\,. \tag{2.21}$$

In the multidimensional case, for $\mathbb{B} \subset \mathbb{R}^3$ and $\partial\mathbb{B} \subset \mathbb{R}^2$, the regularized crack surface functional is rewritten to yield

$$\Gamma_l(d) = \int_{\mathbb{B}} \gamma(d, \nabla d)\, \mathrm{d}V \quad \text{for} \quad d(\boldsymbol{x}, t) := \begin{cases} 1, & \text{on } \Gamma \\ 0, & \text{otherwise} \end{cases} \quad \text{and} \quad t \in \mathbb{R}_+\,, \tag{2.22}$$

with γ, the *crack surface density* per unit volume, defined in terms of a local term, and a nonlocal term associated with the gradient extended theory

$$\gamma(d, \nabla d) = \underbrace{\frac{1}{2l} d^2}_{=:\gamma_{loc}} + \underbrace{\frac{l}{2} |\nabla d|^2}_{=:\gamma_{non}}\,. \tag{2.23}$$



$\partial\mathbb{B}_{\mathbb{N}}$
$\mathbb{B}$
Γ
Z
Y
X
$\partial\mathbb{B}_{\mathbb{N}}$
$\mathbb{B}$
$\Gamma_l(d)$
l
Z
Y
X

(a) (b)

Figure 2.2. (a) Sharp crack topolgy Γ and (b) Regularized crack topology $\Gamma_l(d)$.

Notice that the dimension of the crack surface density γ has to be $[1/L]$. The sharp and diffusive crack topology are schematically depicted in Figure 2.2.

The irreversibility constraint, first established for the fracture energy in equation (2.7), consequently holds to be true for the regularized crack surface functional as well, since cracking processes in nature are in fact irreversible

$$\dot{\Gamma}_l(\dot{d};d) := \frac{\mathrm{d}}{\mathrm{d}t}\Gamma_l(d(\boldsymbol{x},t)) \geq 0\,. \tag{2.24}$$

Introducing the variational derivative of the crack surface density function

$$\delta_d\gamma := \partial_d\gamma - \nabla[\partial_{\nabla d}\gamma] = \frac{1}{l}[d - l^2\Delta d]\,, \tag{2.25}$$

into equation (2.21) for the regularized crack functional, allows for the irreversibilty constraint in (2.24) to be rewritten to yield

$$\dot{\Gamma}_l(\dot{d};d) := \int_{\mathbb{B}} \dot{\gamma}\,\mathrm{d}V = \int_{\mathbb{B}} (\delta_d\gamma)\,\dot{d}\,\mathrm{d}V \geq 0\,. \tag{2.26}$$

Wherein locally a positive variational derivative of the crack surface function and a positive evolution of the fracture phase field have to be granted

$$\delta_d\gamma \geq 0 \quad \text{and} \quad \dot{d} \geq 0\,. \tag{2.27}$$

2.4. Diffusive Fracture at Small Strains

In this section the weak and strong form of the total work for the small deformation case shall be derived. The weak form or variational form is an integral representation of the governing differential equations and boundary conditions, which are referred to as the strong form. An approximation of the aforementioned strong form is built by finite elements upon said weak form. The equations are derived for the quasi-static process, which contrary to dynamic systems, does not take into account the forces of inertia. The procedure of the classical approach, laid out in the literature by Bathe [2], Zienkiewicz and Taylor [38], or Belytschko et al. [3], is represented in a slightly modified version by Miehe in [29]. To this end the balance of total power shall be subjected to a time discretization, resulting in the balance of work, which represents the basis for the variational principle and ultimately the finite element method.

2.4.1. Balance of Total Power

The approach towards dissipative materials will rely on the model introduced by Miehe [29], which for the multi-field formulation considers two functionals

$$E(\boldsymbol{u}, d) := \int_{\mathbb{B}} \Psi \, \mathrm{d}V \quad \text{and} \quad D(\dot{d}; d) := \int_{\mathbb{B}} \phi \, \mathrm{d}V \, . \tag{2.28}$$

The *stored energy functional* E, also simply denoted as *stored energy* (as well as *energy storage*), incorporates the material law, in terms of the *stored energy function* Ψ, also *energy density*, which is derived empirically. The *dissipation potential functional* D is introduced as an additional term, accounting for dissipative processes in the material and is defined in terms of the *dissipation function* ϕ. It is noteworthy to mention that the stored energy function is of the dimension of work per unit volume resulting in a stored energy functional with the dimension of work. The dissipation function on the other hand is of the dimension of power per unit volume, resulting in a dissipation potential functional with the dimension of power. The two volume specific entities Ψ and ϕ are not only dependent on the macroscopic displacement field $\boldsymbol{u}$, but also a microscopic crack phase field d and the thermodynamic driving force β, which is dual to the fracture phase field. These quantities are summarized in the *global unknowns* and the *extended constitutive state vector*

$$\mathfrak{u} := \{u, d, \beta\} \quad \text{and} \quad \mathfrak{c} := \{\boldsymbol{\varepsilon}, d, \nabla d, \beta\} \, . \tag{2.29}$$

At this point it is necessary to emphasize the duality property. The same way the stresses represent the driving force for the strains, the dissipative force field β represents the driving force for the crack phase field d. The stored energy and the dissipation function are now defined in terms of the extended constitutive state vector $\mathfrak{c}$ and its time derivative $\dot{\mathfrak{c}}$

$$\Psi = \Psi(\mathfrak{c}) \quad \text{and} \quad \phi = \phi(\dot{\mathfrak{c}}; \mathfrak{c}) \tag{2.30}$$

The constitutive state is represented for the small deformation case and to this end the *small strain tensor* is introduced as

$$\boldsymbol{\varepsilon} = (\nabla_{\boldsymbol{x}} \boldsymbol{u})^{sym} = \frac{1}{2} \left[\boldsymbol{u} \otimes \nabla_{\boldsymbol{x}} + (\boldsymbol{u} \otimes \nabla_{\boldsymbol{x}})^T \right] \; , \tag{2.31}$$

or in componential notation

$$\varepsilon_{ij} = \frac{1}{2} \left(u_{i,j} + u_{j,i} \right) \, . \tag{2.32}$$

For the linear theory to be valid the components of the linearized strain tensor have to fulfill the following approximations

$$|\varepsilon_{ii}| \leq 0.02; \quad |\gamma_{ij}| \leq 0.245\,, \quad \text{with } \gamma_{ij} = 2 \cdot \varepsilon_{ij}\,. \tag{2.33}$$

The time derivative of the energy storage result in the so called *rate of energy storage* functional

$$\mathcal{E}(\dot{\boldsymbol{u}}, \dot{d}; \boldsymbol{u}, d) := \frac{\mathrm{d}}{\mathrm{d}t} E \tag{2.34}$$

The dissipation function is represented by the equivalent *dissipation functional*

$$\mathcal{D}(\dot{d}; d) \quad := \int_{\mathbb{B}} \{\partial_{\dot{\mathfrak{c}}}\phi : \dot{\mathfrak{c}}\}\ \mathrm{d}V = \int_{\mathbb{B}} \{\phi(\dot{\mathfrak{c}}; \mathfrak{c})\}\ \mathrm{d}V = D(\dot{d}; d)\,. \tag{2.35}$$

Combined with the external power, the balance equation for the total power is derived

$$\begin{aligned} 0 &= P_{int}(\dot{\boldsymbol{u}}, \dot{d}; \boldsymbol{u}, d) - P_{ext}(\dot{\boldsymbol{u}}, \dot{d})\,, \\ 0 &= \mathcal{E}(\dot{\boldsymbol{u}}, \dot{d}; \boldsymbol{u}, d) + \mathcal{D}(\dot{\boldsymbol{u}}, \dot{d}; \boldsymbol{u}, d) - P_{ext}(\dot{\boldsymbol{u}}, \dot{d})\ . \end{aligned} \tag{2.36}$$

In the following sections the functionals in this equation shall be specified for the fracture process.

2.4.2. Dissipation Function for the Extended Three-Field Setting

A dependency between the constitutive variables and the dissipation function is found in Maugin and Morro's 1989 article [22]

$$\frac{\partial\phi(\dot{\mathfrak{c}}; \mathfrak{c})}{\partial\dot{\mathfrak{c}}} : \dot{\mathfrak{c}} = \phi(\dot{\mathfrak{c}}; \mathfrak{c})\,. \tag{2.37}$$

The above relationssship is constructed around the fundamental thermodynamical property of dissipation and comes as a result of the second law of thermodynamics (2.4), which is a statement on the entropy in a system, dependent on the reversible heat, and consequently translates into a condition called the *irreversibilty constraint* for the dissipation functional

$$\mathcal{D} \geq 0\,. \tag{2.38}$$

Consequently, the dissipation function itself has to be positive convex, but not necessarily continuously differentiable. Moreover, for a process, irreversible in nature, the dissipation function has to be homogeneous of degree one. In equation (2.37) the first term is referred to as a *thermodynamic force array* $\mathfrak{f}$

$$\frac{\partial\phi(\dot{\mathfrak{c}}; \mathfrak{c})}{\partial\dot{\mathfrak{c}}} = \partial_{\dot{\mathfrak{c}}}\phi =: \mathfrak{f}\ , \tag{2.39}$$

while the term $\dot{\mathfrak{c}} = \partial\mathfrak{c}/\partial t$ is denoted a flux or generalized velocity and is equivalent to the time derivative of the crack phase field d. Integration of postulate (2.37)

results in the equality put forth in equation (2.35).
Up until this point the dissipation function ϕ itself has not been discussed for a specific application. It is now required to decompose the dissipation function into two separate terms, accounting for homogeneously distributed rates of the fracture phase field, and inhomogeneously distributed rates, as a result of the gradient extended theory

$$\phi(\dot{\mathfrak{c}};\mathfrak{c}) = \underbrace{\check{\phi}_{loc}(\dot{d};d)}_{homogeneous} + \underbrace{\check{\phi}_{non}(\nabla\dot{d};\nabla d)}_{inhomogeneous} . \tag{2.40}$$

The nonlocal term is defined in terms of the critical energy release rate multiplied by the first order derivative of the gradient term of the crack surface density $\check{\gamma}_{non}$, first introduced in equation (2.23)

$$\check{\phi}_{non}(\nabla\dot{d};\nabla d) := g_c\dot{\check{\gamma}}_{non}(\nabla\dot{d};\nabla d) = g_c\,\partial_{\nabla\dot{d}}\gamma\cdot\nabla\dot{d} = (g_c l\nabla d)\cdot\nabla\dot{d} . \tag{2.41}$$

Legendre-Fenchel transformations, thoroughly discussed in [22] and [23], represent a tool, which is used to determine the supremum of the local dissipation function, i.e. the maximum dissipation function $\check{\phi}_{loc}(\dot{d};d)$ for $\beta\in\mathbb{V}\subset\mathbb{R}^n$, which is the first to fulfill the condition

$$\check{\phi}_{loc}(\dot{d};d) = \sup_{\beta}\left[\beta\cdot\dot{d} - \check{\phi}^*_{loc}(\dot{d};d)\right] . \tag{2.42}$$

Here $\phi^*(\dot{d};d)$ represents the conjugate dissipation function, which constrains the thermodynamic driving force field β by a threshold function $\varphi(\beta;d)$,

$$\varphi(\beta;d) = \beta - \psi_c(d) \leq 0 \quad \text{with} \quad \psi_c(d) = \frac{g_c}{l}d , \tag{2.43}$$

and describes for $\varphi(\beta;d)\leq 0$ an elastic, for $\varphi(\beta;d)>0$ an inelastic domain. In this context it can be interpreted as being equivalent to a flow criterion in plasticity theory, which similarly limits the stresses to a yield surface. Once more the field variables d and β have to be distinguished from the internal variables in plasticity theory. The microscopic field variables are present in any given material point, at any given moment and are constantly interacting, i.e. coupled.
The local term of the dissipation function is now introduced in terms of the kinematic viscosity η and as the *viscous regularization* of the rate-independent formulation, now denoted the rate-dependent dissipation function

$$\check{\phi}^{\eta}_{loc}(\dot{d};d) = \sup_{\beta}\Big[\beta\cdot\dot{d} - \underbrace{\frac{1}{2\eta}\left\langle\varphi(\beta;d)\right\rangle^2_+}_{\check{\phi}^{*\eta}_{loc}(\beta;d)}\Big] = \sup_{\beta}\Big[\beta\cdot\dot{d} - \frac{1}{2\eta}\left\langle\beta-\psi_c(d)\right\rangle^2_+\Big] . \tag{2.44}$$

The Macaulay bracket $\langle(x)\rangle_+ = \frac{1}{2}(x+|x|)$ meets the condition

$$\begin{cases}\langle x\rangle_+ = 0, & \text{for } x\leq 0\\ \langle x\rangle_+ > 0, & \text{for } x>0\end{cases} \quad\Rightarrow\quad \begin{cases}\langle\varphi(\beta;d)\rangle_+ = 0, & \text{for } \varphi(\beta;d)\leq 0\\ \langle\varphi(\beta;d)\rangle_+ > 0, & \text{for } \varphi(\beta;d)>0\end{cases} . \tag{2.45}$$

By way of defining a Lagrange function, the necessary evolution equation is obtained as a solution of the minimization problem

$$\Lambda(\beta, \dot{d}, d) = \beta \cdot \dot{d} - \frac{1}{2\eta} \langle \beta - \frac{g_c}{l} d \rangle_+^2 \tag{2.46a}$$

$$\nabla_{\beta,\dot{d},d} \Lambda(\beta, \dot{d}, d) \overset{!}{=} 0 \tag{2.46b}$$

Specifically the partial differentation with respect to β provides a value for the rate of the crack phase field $\dot{d}$, which via the second order derivative with respect to β is proven to be the maximum and therefore supremum

$$\frac{\partial \Lambda}{\partial \beta} = \dot{d} - \frac{1}{\eta} \langle \beta - \psi_c(d) \rangle_+ \overset{!}{=} 0 \quad \rightarrow \quad \dot{d} = \frac{1}{\eta} \langle \beta - \psi_c(d) \rangle_+ \, , \tag{2.47}$$

$$\frac{\partial^2 \Lambda}{\partial \beta^2} = -\frac{1}{\eta} < 0 \ \rightarrow \ \text{maximum!} \, . \tag{2.48}$$

Observing equations (2.44) and (2.45) the properties of the yield function are fully understood, as it is interpreted as a penalty-type term, which acts on the material, once the condition $\beta > \psi_c(d) = \frac{g_c}{l} d$ is satisfied. This is the case, once the rate of increase of the dissipative forces β is significantly larger than the growth of the fracture phase field d. This attribute manifests itself in a drastic decrease of the stresses, mimicing first nonlinear elastic behaviour at the initiation of the crack propagation, and later plastic degradation. The conjugate local dissipation function $\check{\phi}^{*\eta}_{loc}$ is graphically illustrated in Figure 2.3.

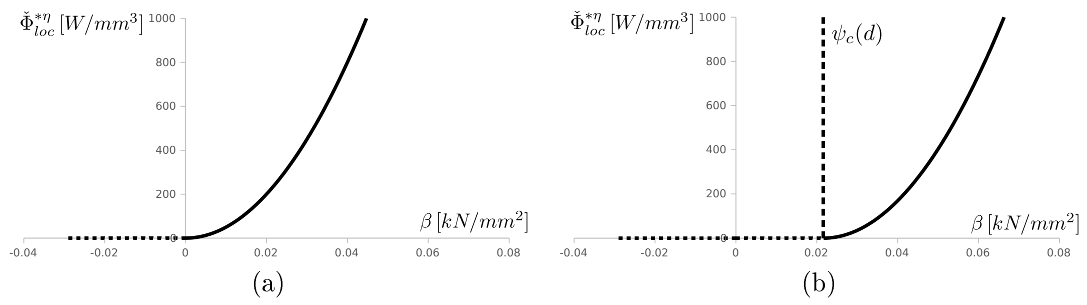


Figure 2.3. Conjugate local dissipation function $\check{\phi}^{*\eta}_{loc}$ for (a) $d = 0$ and (b) $d = 0.3$, a viscosity of $\eta = 1.0 \times 10^{-6}\, kNs/mm^2$, a critical energy release rate of $g_c = 2.7 \times 10^{-3}\, kN/mm$ and a length-scale parameter of $l = 0.0375\, mm$. As can be seen, the function is only activated once the condition $\beta > \psi_c(d) = g_c d/l$ is satisfied.

The resulting dissipation function defined in (2.40), produces with the equations put forth for the nonlocal term in (2.41) and the local term in (2.44) the following expression

$$\phi(\nabla \dot{d}, \dot{d}; \nabla d, d) = \underbrace{\sup_{\beta} \Big[\beta \cdot \dot{d} - \frac{1}{2\eta} \langle \beta - \frac{g_c}{l} d \rangle_+^2 \Big]}_{\check{\phi}^{\eta}_{loc}(\dot{d}; d)} + \underbrace{(g_c l \nabla d) \cdot \nabla \dot{d}}_{\check{\phi}_{non}(\nabla \dot{d}; \nabla d)} \, . \tag{2.49}$$

Inserting above equation into (2.35) allows for the dissipation functional to be formulated as

$$
\begin{aligned}
\mathcal{D}(\dot{d};d;\beta) &:= \int\limits_{\mathbb{B}} \{\partial_{\dot{\mathfrak{c}}}\phi : \dot{\mathfrak{c}}\}\ \mathrm{d}V = \int\limits_{\mathbb{B}} \left\{\partial_{\dot{d}}\phi \cdot \dot{d} + \partial_{\nabla\dot{d}}\phi \cdot \nabla\dot{d} + \partial_{\beta}\phi \cdot \dot{\beta}\right\}\ \mathrm{d}V \\
&= \int\limits_{\mathbb{B}} \left\{\delta_{\dot{d}}\phi \cdot \dot{d} + +\partial_{\beta}\phi \cdot \dot{\beta}\right\}\ \mathrm{d}V + \int\limits_{\partial\mathbb{B}} \left\{\boldsymbol{n}[\partial_{\nabla\dot{d}}\phi \cdot \dot{d}]\right\}\ \mathrm{d}A \\
&= \int\limits_{\mathbb{B}} \left\{[\beta - g_c l \Delta d] \cdot \dot{d} + [\dot{d} - \frac{1}{\eta}\langle \beta - \psi_c(d)\rangle_+] \cdot \dot{\beta}\right\}\ \mathrm{d}V + \int\limits_{\partial\mathbb{B}} \left\{\boldsymbol{n}[\partial_{\nabla\dot{d}}\phi \cdot \dot{d}]\right\}\ \mathrm{d}A\,.
\end{aligned}
\tag{2.50}
$$

In above equation the variational derivative of the dissipation function was introduced as

$$
\delta_{\dot{d}}\phi = \frac{\partial\phi}{\partial\dot{d}} - \nabla^T\left[\frac{\partial\phi}{\partial\nabla\dot{d}}\right]\,. \tag{2.51}
$$

2.4.3. Isotropic Degradation of Stored Bulk Energy

For the applicational purpose the decrease in free energy stored inside the material body $\mathbb{B}$, which is a direct consequence of the fracturing process being disspative in nature, is modeled in an isotropic way. This translates to the constitutive variables being identical in any given direction and the degradation manifesting itself in tension and compression, contrary to the anisotropic model, where only the tension-dependent parts of the constitutive variables are affected by a degradation function. The stored energy functional was first defined in equation (2.28) in terms of the stored energy function and is now specified as dependent on the crack phase field

$$
E(\boldsymbol{u},d) := \int\limits_{\mathbb{B}} \Psi(\mathfrak{c})\ \mathrm{d}V\,, \tag{2.52a}
$$

$$
\text{with}\quad \Psi(\mathfrak{c}) = \bar{\Psi}_{loc}(\boldsymbol{\varepsilon},d)\,. \tag{2.52b}
$$

Chosen for an ideal elastic, isotropic material, the elastic potential only has two independent ealstic moduli, which are the first Lamé constant λ and the *shear modulus* μ, also denoted the *second Lamé constant*, defined in terms of the bluk modulus κ, *elastic modulus* E and *Poisson's ratio* ν

$$
\lambda := \varkappa - \tfrac{2}{3}\mu \tag{2.53}
$$

$$
\text{with}\quad \varkappa = K = \frac{E}{3(1-2\nu)} \qquad \text{and} \qquad \mu = G = \frac{E}{2(1+\nu)}\,. \tag{2.54}
$$

The elastic potential reflects a linear proportionality between the stresses and the strains

$$
\Psi_0(\boldsymbol{\varepsilon}) = \frac{1}{2}\lambda((\mathrm{tr}(\boldsymbol{\varepsilon}))^2 + \mu\,\mathrm{tr}(\boldsymbol{\varepsilon}^2)\,. \tag{2.55}
$$

With the identity $\mathrm{tr}(\boldsymbol{\varepsilon}^2) = \mathrm{tr}(\boldsymbol{\varepsilon}^T \boldsymbol{\varepsilon}) = \boldsymbol{\varepsilon} : \boldsymbol{\varepsilon}$. The degradation is accounted for by multiplying Ψ_0 by a monotionically decreasing function $g(d)$ regulated by the parameter k

$$\bar{\Psi}_{loc}(\boldsymbol{\varepsilon}, d) = [g(d) + k]\, \Psi_0(\boldsymbol{\varepsilon})\,. \tag{2.56}$$

The function $g(d)$ is supposed the render the properties of the free energy, when effected by the evolving crack propagation. The free energy is maximal in the uncracked state for $d = 0$ and minimal in the fully cracked state for $d = 1$. Moreover the function $g(d)$ accounts for the field property of d and its effect on the elastic driving force $\beta^e := \delta_d \Psi$, which for $d = 1$ converges. All of the aforementioned properties are encompassed in the function

$$g(d) = (1-d)^2\,. \tag{2.57}$$

The parameter k is an internal variable, which is chosen to be as small as possible. Integrating the newly aquired correlations and expanding on equation (2.34) produces

$$\mathcal{E}(\dot{\boldsymbol{u}}, \dot{d}; \boldsymbol{u}, d) := \frac{\mathrm{d}}{\mathrm{d}t} E(\boldsymbol{u}, d) = \frac{\mathrm{d}}{\mathrm{d}t} \int_{\mathbb{B}} \Psi(\mathfrak{c})\, \mathrm{d}V = \int_{\mathbb{B}} \left\{ \partial_{\boldsymbol{\varepsilon}} \Psi : \dot{\boldsymbol{\varepsilon}} + \delta_d \Psi \cdot \dot{d} \right\} \mathrm{d}V\,. \tag{2.58}$$

Note that contrary to the partial differential of an internal variable, the variational derivative in $\delta_d \Psi$ emphasizes the significance of the field property of the crack phase field and as a generalized state variable accounts for its gradient. The contributing stresses and the elastic driving force β^e are calculated to

$$\boldsymbol{\sigma} := \frac{\partial \Psi}{\partial \boldsymbol{\varepsilon}} = [(1-d)^2 + k]\, \boldsymbol{\sigma}_0(\boldsymbol{\varepsilon}) \quad \text{and} \quad \beta^e := \frac{\delta \Psi}{\delta d} = -2(1-d)\Psi_0(\boldsymbol{\varepsilon})\,. \tag{2.59}$$

The undamaged stresses follow as the differentation of $\Psi_0(\boldsymbol{\varepsilon})$, with respect to $\boldsymbol{\varepsilon}$

$$\boldsymbol{\sigma}_0(\boldsymbol{\varepsilon}) = \frac{\partial \Psi_0}{\partial \boldsymbol{\varepsilon}} = \lambda\, \mathrm{tr}(\boldsymbol{\varepsilon}) \cdot \boldsymbol{I} + 2\mu \boldsymbol{\varepsilon}\,. \tag{2.60}$$

Of equal significance is the elasticity tensor defined in terms of the differential

$$\mathbb{C} = \frac{\partial \boldsymbol{\sigma}}{\partial \boldsymbol{\varepsilon}} = [(1-d)^2 + k]\, \mathbb{C}_0 \quad \text{with} \quad \mathbb{C}_0 = \frac{\partial \boldsymbol{\sigma}_0}{\partial \boldsymbol{\varepsilon}} = \lambda\, \boldsymbol{I} \otimes \boldsymbol{I} + 2\mu\, \mathbb{I}\,. \tag{2.61}$$

The rate of energy storage for an isotropic material with fracture yields

$$\begin{aligned} \mathcal{E}(\dot{\boldsymbol{u}}, \dot{d}; \boldsymbol{u}, d) = & \int_{\mathbb{B}} \left\{ -\nabla^T \left\{ [(1-d)^2 + k]\, \boldsymbol{\sigma}_0(\boldsymbol{\varepsilon}) \right\} \cdot \dot{\boldsymbol{u}} - 2(1-d)\Psi_0(\boldsymbol{\varepsilon}) \cdot \dot{d} \right\} \mathrm{d}V \\ & + \int_{\partial\mathbb{B}} \left\{ \boldsymbol{n}^T \cdot [(1-d)^2 + k]\, \boldsymbol{\sigma}_0(\boldsymbol{\varepsilon}) \cdot \dot{\boldsymbol{u}} \right\} \mathrm{d}A\,. \end{aligned} \tag{2.62}$$

In this equation the Gauß-theorem was applied to the term

$$\begin{aligned} [(1-d)^2 + k]\, \boldsymbol{\sigma}_0(\boldsymbol{\varepsilon}) : \nabla \dot{\boldsymbol{u}} = & \nabla \{ [(1-d)^2 + k]\, \boldsymbol{\sigma}_0(\boldsymbol{\varepsilon}) \cdot \dot{\boldsymbol{u}} \} \\ & - \nabla^T \{ [(1-d)^2 + k]\, \boldsymbol{\sigma}_0(\boldsymbol{\varepsilon}) \} \cdot \dot{\boldsymbol{u}}\,, \end{aligned} \tag{2.63}$$

wherein the second term represent the functional derivative of Ψ with respect to the displacement field $\boldsymbol{u}$

$$\delta_{\boldsymbol{u}} \Psi := -\nabla^T [\partial_{\boldsymbol{\varepsilon}} \Psi]\;. \tag{2.64}$$

2.4.4. Governing Equations

The resulting governing equations are a total of five differential equations, three for the three components of the macroscopic displacement field $\boldsymbol{u}$ and one each for the two scalar entities, the crack phase field d and the thermodynamic driving force β. They are derived from the equilibrium of power, first established in equation (2.36). External power is induced by mechanical body and surface forces

$$P_{ext}(\dot{\boldsymbol{u}}) = \int_{\mathbb{B}} \boldsymbol{f}^B \cdot \dot{\boldsymbol{u}} \, \mathrm{d}V + \int_{\partial\mathbb{B}} \boldsymbol{f}^S \cdot \dot{\boldsymbol{u}} \, \mathrm{d}A \,. \tag{2.65}$$

The crack phase field, which is assumed to take the value $d = 0$ at the start of the fracture process, is coupled with the displacement field $\boldsymbol{u}$, thus no external loading terms are prescribed. The euilibrium of total power combines the rate of energy storage, dissipation and the external load (2.65) into

$$0 \;=\; \mathcal{E}(\dot{\boldsymbol{u}}, \dot{d}; \boldsymbol{u}, d) + \mathcal{D}(\dot{d}; d) - P_{ext}(\dot{\boldsymbol{u}}) \,. \tag{2.66}$$

Leading to the integral forms

$$\begin{aligned}
0 = & \int_{\mathbb{B}} \underbrace{\left\{-\nabla^T[((1-d)^2 + k)\, \boldsymbol{\sigma}_0(\boldsymbol{\varepsilon})] - \boldsymbol{f}^B\right\}}_{I.} \cdot \dot{\boldsymbol{u}} \, \mathrm{d}V \\
& + \int_{\mathbb{B}} \underbrace{\left\{-2(1-d)\Psi_0(\boldsymbol{\varepsilon}) + \beta - g_c l \Delta d\right\}}_{II.} \cdot \dot{d} \, \mathrm{d}V \\
& + \int_{\mathbb{B}} \underbrace{\left\{[\dot{d} - \frac{1}{\eta}\langle \beta - \psi_c(d)\rangle_+]\right\}}_{III.} \cdot \dot{\beta} \, \mathrm{d}V \\
& + \int_{\partial\mathbb{B}} \left\{\boldsymbol{n}^T \cdot [((1-d)^2 + k)\, \boldsymbol{\sigma}_0(\boldsymbol{\varepsilon}) - \boldsymbol{f}^S] \cdot \dot{\boldsymbol{u}} + \boldsymbol{n}^T \cdot g_c l \nabla d \cdot \dot{d}\right\} \mathrm{d}A \,.
\end{aligned} \tag{2.67}$$

The coupled Euler equations are constructed from the conditions $I.$ to $III.$ from above equation

$$\begin{aligned}
\text{I.} \quad & \boldsymbol{0} = \nabla^T[((1-d)^2 + k)\, \boldsymbol{\sigma}_0(\boldsymbol{\varepsilon})] + \boldsymbol{f}^B \\
\text{II.} \quad & 0 = g_c l \Delta d + 2(1-d)\Psi_0(\boldsymbol{\varepsilon}) - \beta \\
\text{III.} \quad & 0 = \dot{d} - \frac{1}{\eta}\langle \beta - \psi_c(d)\rangle_+ \,.
\end{aligned} \tag{2.68}$$

2.4.5. Time-Discrete Incremental Variational Principles for Phase Field Fracture

Aiming towards a numerical implementation of the balance equations derived in the previous section a time-discrete representation has to be introduced. Consider the time interval $t_i \in [t_0, t_k]$ with $i \in \mathbb{Z}_+$. The increment, i.e. the finite, constant difference, between the time of the previous step t_n, attached with the known solutions $\mathfrak{c}_n$ of the constitutive state variables and the current time t_{n+1}, with the unknown, yet to be determined solutions $\mathfrak{c}_{n+1}$ is defined as the time step length

$$\tau := t_{n+1} - t_n > 0\,. \tag{2.69}$$

For convenience the subscript $n+1$ is omitted from all further notations, yielding the field variables and constitutive state vector as

$$\begin{aligned}
&\boldsymbol{u} := \boldsymbol{u}(t_{n+1}) && \text{and} && \boldsymbol{u}_n := \boldsymbol{u}(t_n) && \text{(2.70a)}\\
&d := d(t_{n+1}) && \text{and} && d_n := d(t_n) && \text{(2.70b)}\\
&\beta := \beta(t_{n+1}) && \text{and} && \beta_n := \beta(t_n)\,, && \text{(2.70c)}\\
t_{n+1}:\ &\mathfrak{c} := \{\boldsymbol{\varepsilon}, d, \nabla d, \beta\} && \text{and} && t_n:\ \ \mathfrak{c}_n := \{\boldsymbol{\varepsilon}_n, d_n, \nabla d_n, \beta_n\}\,. && \text{(2.70d)}
\end{aligned}$$

The rates of change of the deformation vector and constitutive state vectors are linear

$$\dot{\mathfrak{u}} := \frac{1}{\tau}\left(\mathfrak{u} - \mathfrak{u}_n\right) \quad \text{and} \quad \dot{\mathfrak{c}} \ := \tfrac{1}{\tau}\left(\mathfrak{c} - \mathfrak{c}_n\right)\,. \tag{2.71}$$

Using this notation the incremental dissipation functional is expressed in terms of the local and nonlocal, gradient-type, contributions to the dissipation function

$$D^\tau_\eta \ = \int_{t_n}^{t_{n+1}} D\ \mathrm{d}t = \int_{t_n}^{t_{n+1}} \left\{ \int_{\mathbb{B}} \phi(\dot{\mathfrak{c}};\mathfrak{c})\ \mathrm{d}V \right\}\ \mathrm{d}t = \int_{t_n}^{t_{n+1}} \left\{ \int_{\mathbb{B}} [\check{\phi}_{loc} + \check{\phi}_{non}]\ \mathrm{d}V \right\}\ \mathrm{d}t\,. \tag{2.72}$$

The contributing, time-discrete dissipation functions are constructed in a straight forward fashion

$$\check{\phi}_{loc} = \beta \cdot \frac{(d-d_n)}{\tau} - \frac{1}{2\eta}\left\langle \beta - \psi_c(d_n)\right\rangle^2_+ \tag{2.73}$$

$$\check{\phi}_{non} = g_c \cdot \frac{\check{\gamma}_{non} - \check{\gamma}_{non,n}}{\tau} = \frac{g_c l}{2\tau}[\|\nabla d\|^2 - \|\nabla d_n\|^2],\,, \tag{2.74}$$

yielding the incremental dissipation functional, with respect to the two field variables d and β

$$D^\tau_\eta(d,\beta) = \int_{\mathbb{B}} \left\{ \beta(d-d_n) - \frac{\tau}{2\eta}\left\langle \beta - \psi_c(d_n)\right\rangle^2_+ + \frac{g_c l}{2}[\|\nabla d\|^2 - \|\nabla d_n\|^2] \right\}\ \mathrm{d}V \tag{2.75}$$

The incremental stored energy is obtained by integrating the free energy over the discrete time interval $t \in [t_n, t_{n+1}]$

$$E^\tau(\boldsymbol{u}, d) = \int_{\mathbb{B}} \left\{ \Psi(\mathfrak{c}) - \Psi(\mathfrak{c}_n) \right\}\ \mathrm{d}V\,. \tag{2.76}$$

Once more the body and surface forces are considered to be constant throughtout the integration interval, which is a valid assumption for a quasi-static process

$$W^\tau_{ext}(\boldsymbol{u}) = \int_{t_n}^{t_{n+1}} P_{ext}\ \mathrm{d}t = \int_{\mathbb{B}} \boldsymbol{f}^B \cdot (\boldsymbol{u} - \boldsymbol{u}_n)\ \mathrm{d}V + \int_{\partial\mathbb{B}} \boldsymbol{f}^S \cdot (\boldsymbol{u} - \boldsymbol{u}_n)\ \mathrm{d}A\,. \tag{2.77}$$

The potential, as the resulting quantity of the balance of the internal and external work for the quasi-static process, is defined in terms of the incremental energy, dissipation and work functionals as

$$\Pi_\eta^\tau(\boldsymbol{u}, d, \beta) = E^\tau(\boldsymbol{u}, d) + D_\eta^\tau(d, \beta) - W_{ext}^\tau(\boldsymbol{u}) \tag{2.78}$$

$$= \int\limits_{\mathbb{B}} \left\{ \pi_\eta^\tau(\mathfrak{c}; \mathfrak{c}_n) - \boldsymbol{f}^B \cdot (\boldsymbol{u} - \boldsymbol{u}_n) \right\} \,\mathrm{d}V - \int\limits_{\partial\mathbb{B}} \boldsymbol{f}^S \cdot (\boldsymbol{u} - \boldsymbol{u}_n) \,\mathrm{d}A\,. \tag{2.79}$$

In the previous equation the *incremental internal work density* combines terms associated with the energy and dissipation functional with respect to the constitutive state vectors in the time interval under consideration

$$\pi_\eta^\tau = \Psi(\mathfrak{c}) - \Psi(\mathfrak{c}_n) + \beta(d - d_n) - \frac{\tau}{2\eta} \langle \beta - \psi_c(d_n) \rangle_+^2 + \frac{g_c l}{2} [\|\nabla d\|^2 - \|\nabla d_n\|^2]\,. \tag{2.80}$$

Application of the Variational Principle

It is now of interest to find the variation of the incremental potential, which is equivalent to the *minimization principle.* In this context it is referred to as the *stationary principle* with respect to the current time t_{n+1}, since its result yields a stationary formulation for the potential Π_η^τ, when subjected to virtual displacements $\delta\mathfrak{u}$. To this end the arguments of the potential Π_η^τ are wanted, which fulfill the condition

$$\mathfrak{S} = \{\boldsymbol{u}, d, \beta\} = \arg\left\{ \inf_{\boldsymbol{u}} \inf_{d} \sup_{\beta} \Pi_\eta^\tau(\boldsymbol{u}, d, \beta) \right\}\,. \tag{2.81}$$

In conjunction with a numerical application, the resulting function $\delta\Pi_\eta^\tau$ aims towards zero for arbitrary displacements $\delta\mathfrak{u}$

$$0 = \delta\Pi_\eta^\tau(\mathfrak{u}) = \frac{\partial \Pi_\eta^\tau(\mathfrak{u})}{\partial \mathfrak{u}}\,\delta\mathfrak{u} = \delta E^\tau + \delta D_\eta^\tau - \delta W_{ext}^\tau =: \mathcal{R}(\mathfrak{u})\,\delta\mathfrak{u}\,. \tag{2.82}$$

This equation implies that for any small displacement $\delta\mathfrak{u}$, superimposed onto the continuum in an arbitrary direction, a function has to be found, for which the state of equilibrium is preserved. This state is equivalent to a *saddle point* $\mathfrak{S}$ in the function of the potential, as depicted in Figure 2.4.

Applying the variational principle to the algorithmic potential with respect to the field variables at t_{n+1}, results in

$$\delta\Pi_\eta^\tau = \int\limits_{\mathbb{B}} \left\{ \delta\pi_\eta^\tau(\mathfrak{c}; \mathfrak{c}_n) - \boldsymbol{f}^B \cdot \delta(\boldsymbol{u} - \boldsymbol{u}_n) \right\} \,\mathrm{d}V - \int\limits_{\partial\mathbb{B}} \boldsymbol{f}^S \cdot \delta(\boldsymbol{u} - \boldsymbol{u}_n) \,\mathrm{d}A \overset{!}{=} 0 \tag{2.83}$$

$$\text{with} \quad \delta\pi_\eta^\tau = \frac{\partial \pi_\eta^\tau}{\partial \mathfrak{c}} : \delta\mathfrak{c} = \frac{\partial \pi_\eta^\tau}{\partial \boldsymbol{\varepsilon}} : \delta\boldsymbol{\varepsilon} + \frac{\partial \pi_\eta^\tau}{\partial d} \cdot \delta d + \frac{\partial \pi_\eta^\tau}{\partial (\nabla d)} \cdot \delta(\nabla d) + \frac{\partial \pi_\eta^\tau}{\partial \beta} \cdot \delta\beta\,. \tag{2.84}$$

Defining the second order tensor as $\mathcal{S} := \partial\pi_\eta^\tau / \partial\mathfrak{c}$ and introducing the differential

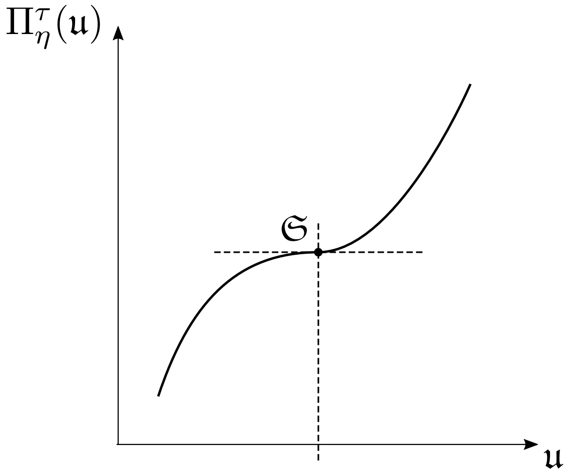


Figure 2.4. Saddle point $\mathfrak{S}$ of the potential $\Pi_\eta^\tau(\mathfrak{u})$

operator matrix $\boldsymbol{D}$ for the three dimensional case, the following formulation is found

$$\delta\Pi_\eta^\tau(\mathfrak{u}) = \int\limits_{\mathbb{B}} \left\{(\boldsymbol{D}\,\delta\mathfrak{u})^T \cdot \mathcal{S} - \delta\mathfrak{u}^T \cdot \mathcal{F}^B\right\}\ \mathrm{d}V - \int\limits_{\partial\mathbb{B}_\mathbb{N}} \left\{\delta\mathfrak{u}^T \cdot \mathcal{F}^S\right\}\ \mathrm{d}A \tag{2.85}$$

$$\text{with}\quad \mathcal{S}^{Voigt} = \begin{bmatrix} \left[\partial_{\nabla^s \boldsymbol{u}}\pi_\eta^\tau\right]_{[6\times1]} \\ \left[\partial_d \pi_\eta^\tau\right]_{[1\times1]} \\ \left[\partial_{\nabla d}\pi_\eta^\tau\right]_{[3\times1]} \\ \left[\partial_\beta \pi_\eta^\tau\right]_{[1\times1]} \end{bmatrix} \tag{2.86}$$

$$\text{and}\quad \delta\mathfrak{c}^{Voigt} = \delta\mathfrak{c} = \boldsymbol{D}\cdot\delta\mathfrak{u} \tag{2.87}$$

$$= \underbrace{\begin{bmatrix} [\nabla^{sym}]_{[6\times3]} & [\boldsymbol{0}]_{[6\times1]} & [\boldsymbol{0}]_{[6\times1]} \\ [\boldsymbol{0}]_{[1\times3]} & [1]_{[1\times1]} & [0]_{[1\times1]} \\ [\boldsymbol{0}]_{[3\times3]} & [\nabla]_{[3\times1]} & [\boldsymbol{0}]_{[3\times1]} \\ [\boldsymbol{0}]_{[1\times3]} & [0]_{[1\times1]} & [1]_{[1\times1]} \end{bmatrix}}_{=:\boldsymbol{D}} \cdot \begin{bmatrix} \delta\boldsymbol{u} \\ \delta d \\ \delta\beta \end{bmatrix} \tag{2.88}$$

$$= \begin{bmatrix} [\delta\boldsymbol{\varepsilon}]_{[6\times1]} \\ [\delta d]_{[1\times1]} \\ [\nabla\delta d]_{[3\times1]} \\ [\delta\beta]_{[1\times1]} \end{bmatrix}. \tag{2.89}$$

In equation (2.85) the extended body force vector $\mathcal{F}^B$ and surface force vector $\mathcal{F}^S$ were defined as

$$\mathcal{F}^B := [\boldsymbol{f}^B \mid 0 \mid 0]^T_{[5\times1]} \quad\text{and}\quad \mathcal{F}^S := [\boldsymbol{f}^S \mid 0 \mid 0]^T_{[5\times1]}\ . \tag{2.90}$$

Evaluating the partial differentations and collecting the terms associated with each field variable results in the integral expression

$$\begin{aligned}\delta\Pi^{\tau}_{\eta} = 0 = &\int_{\mathbb{B}} \underbrace{\left\{-\nabla^T[((1-d)^2+k)\,\boldsymbol{\sigma}_0(\boldsymbol{\varepsilon})] - \boldsymbol{f}^B\right\}}_{I.} \cdot \delta\boldsymbol{u}\ \mathrm{d}V \\ + &\int_{\mathbb{B}} \underbrace{\left\{-2(1-d)\Psi_0(\boldsymbol{\varepsilon}) + \beta - g_c l \Delta d\right\}}_{II.} \cdot \delta\dot{d}\ \mathrm{d}V \\ + &\int_{\mathbb{B}} \underbrace{\left\{d - d_n - \frac{\tau}{\eta}\langle \beta - \psi_c(d_n)\rangle_+\right\}}_{III.} \cdot \delta\beta\ \mathrm{d}V \\ + &\int_{\partial\mathbb{B}} \left\{\boldsymbol{n}^T \cdot [((1-d)^2+k)\,\boldsymbol{\sigma}_0(\boldsymbol{\varepsilon}) - \boldsymbol{f}^S] \cdot \delta\boldsymbol{u} + \boldsymbol{n}^T \cdot g_c l \nabla d \cdot \delta d\right\} \mathrm{d}A\,. \qquad (2.91)\end{aligned}$$

For the stationary of the potential to be zero, each of the three expressions labeled, has to be zero, allowing the coupled balance equations to be formulated

$$\begin{aligned} \text{I.}\quad & \boldsymbol{0} = \nabla^T[((1-d)^2+k)\,\boldsymbol{\sigma}_0(\boldsymbol{\varepsilon})] + \boldsymbol{f}^B \\ \text{II.}\quad & 0 = g_c l \Delta d + 2(1-d)\Psi_0(\boldsymbol{\varepsilon}) - \beta \\ \text{III.}\quad & 0 = d - d_n - \frac{\tau}{\eta}\langle \beta - \psi_c(d_n)\rangle_+\,. \end{aligned} \qquad (2.92)$$

2.5. FE-Discretization of Incremental Variational Principle

With regards to a finite element discretization a three dimensional case is considered, for which the elemental generalized deformation vector and extended constitutive state vector are constructed as

$$\begin{aligned} \mathfrak{u} &= [u, v, w \mid d \mid \beta]^T \\ \mathfrak{c}^{Voigt} = \mathfrak{c} &= [\varepsilon_{11}, \varepsilon_{22}, \varepsilon_{33}, \varepsilon_{12}, \varepsilon_{23}, \varepsilon_{31} \mid d \mid d_{,1}, d_{,2}, d_{,3} \mid \beta]^T\,. \end{aligned} \qquad (2.93)$$

For hexahedral elements, the displacement interpolation matrix is identical to the one put forth in equation (3.7), but for practical purposes the shape functions are chosen to be identical for the displacement field $\boldsymbol{u}$, the fracture phase field d and the dissipative force field β, i.e.

$$h_i := h_i(r,s,t) = \check{h}_i(r,s,t) = \tilde{h}_i(r,s,t)\,. \qquad (2.94)$$

The shape functions act upon the local nodal state vector and yield the exact values of the displacement vector in the nodes. The relation is characterized by the following statement

$$\mathfrak{u}^{(m)}(r,s,t;\mathrm{t}) = \boldsymbol{H}^{(m)}(r,s,t)_{[5\times 40]} \cdot \hat{\mathfrak{u}}(\mathrm{t})_{[40\times 1]}\,, \qquad (2.95)$$

$$\text{with}\quad \hat{\mathfrak{u}}(\mathrm{t}) = [u_1, v_1, w_1, d_1, \beta_1 \mid (\ldots)_k \mid u_8, v_8, w_8, d_8, \beta_8]^T_{[40\times 1]}\,. \qquad (2.96)$$

The strain-displacement matrix is constructed from an expression, which will be introduced in equation (3.14) of the following chapter, but with the specification of identical shape functions for all three field variables as mentioned in (2.94)

$$\mathfrak{c}^{(m)}(r,s,t;\mathrm{t}) = \boldsymbol{B}^{(m)}(r,s,t)_{[5\times 40]} \cdot \hat{\mathfrak{u}}(\mathrm{t})_{[40\times 1]} \,. \tag{2.97}$$

The components of the second order tensor $\mathcal{S}$ put forth in equation (2.89) as well as the forth order material elasticity tensor $\mathbb{C}^{\tau}_{\eta}$, which is represented a 11×11 matrix, have to be further specified

$$\begin{aligned}\mathbb{C}^{\tau}_{\eta} &= \begin{bmatrix} \left[\partial_{\boldsymbol{\varepsilon}}\pi^{\tau}_{\eta}\right]_{[6\times1]} \\ \left[\partial_{d}\pi^{\tau}_{\eta}\right]_{[1\times1]} \\ \left[\partial_{\nabla d}\pi^{\tau}_{\eta}\right]_{[3\times1]} \\ \left[\partial_{\beta}\pi^{\tau}_{\eta}\right]_{[1\times1]} \end{bmatrix} \left[[\partial_{\boldsymbol{\varepsilon}}]_{[1\times6]}\,, [\partial_{d}]_{[1\times1]}\,, [\partial_{\nabla d}]_{[3\times1]}\,, [\partial_{\beta}]_{[1\times1]} \right] \\ &= \begin{bmatrix} \left[\partial^2_{\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}}\pi^{\tau}_{\eta}\right]_{[6\times6]} & \left[\partial^2_{\boldsymbol{\varepsilon}d}\pi^{\tau}_{\eta}\right]_{[6\times1]} & [\boldsymbol{0}]_{[6\times3]} & [\boldsymbol{0}]_{[6\times1]} \\ \left[\partial^2_{d\boldsymbol{\varepsilon}}\pi^{\tau}_{\eta}\right]_{[1\times6]} & \left[\partial^2_{dd}\pi^{\tau}_{\eta}\right]_{[1\times1]} & [\boldsymbol{0}]_{[1\times3]} & [1]_{[1\times1]} \\ [\boldsymbol{0}]_{[3\times6]} & [\boldsymbol{0}]_{[3\times1]} & \left[\partial^2_{\nabla d\nabla d}\pi^{\tau}_{\eta}\right]_{[3\times3]} & [\boldsymbol{0}]_{[3\times1]} \\ [\boldsymbol{0}]_{[1\times6]} & [1]_{[1\times1]} & [\boldsymbol{0}]_{[1\times3]} & \left[\partial^2_{\beta\beta}\pi^{\tau}_{\eta}\right]_{[1\times1]} \end{bmatrix} .\end{aligned} \tag{2.98}$$

The first and second order derivatives of the incremental internal work density read for the case of elastic loading $\varphi(\beta; d_n) < 0$

$$\begin{aligned} \partial_{\boldsymbol{\varepsilon}}\pi^{\tau}_{\eta} &= [(1-d)^2 + k]\boldsymbol{\sigma}_0(\boldsymbol{\varepsilon}) & \partial^2_{\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}}\pi^{\tau}_{\eta} &= [(1-d)^2 + k]\mathbb{C}_0 \\ \partial_{d}\pi^{\tau}_{\eta} &= \beta - 2(1-d)\Psi_0(\boldsymbol{\varepsilon}) & \partial^2_{\boldsymbol{\varepsilon}d}\pi^{\tau}_{\eta} &= \partial^2_{d\boldsymbol{\varepsilon}}\pi^{\tau}_{\eta} = -2(1-d)\boldsymbol{\sigma}_0(\boldsymbol{\varepsilon}) \\ \partial_{\nabla d}\pi^{\tau}_{\eta} &= g_c l \nabla d & \partial^2_{dd}\pi^{\tau}_{\eta} &= 2\Psi_0(\boldsymbol{\varepsilon}) \\ \partial_{\beta}\pi^{\tau}_{\eta} &= d - d_n & \partial^2_{\nabla d\nabla d}\pi^{\tau}_{\eta} &= g_c l \\ & & \partial^2_{\beta\beta}\pi^{\tau}_{\eta} &= 0 \end{aligned} \tag{2.99}$$

The dissipation function for the rate-dependet viscous regularization comes into effect for the case of inelastic loading $\varphi(\beta; d_n) \geq 0$, updating the first and second order derivatives with respect to β

$$\partial_{\beta}\pi^{\tau}_{\eta} \;=\; \begin{cases} d - d_n, & \text{for } \ \varphi(\beta; d_n) < 0 \\ d - d_n - \frac{\tau}{\eta}\,\langle \beta - \psi_c(d_n)\rangle_{+}, & \text{for } \ \varphi(\beta; d_n) \geq 0 \end{cases} \tag{2.100}$$

$$\partial^2_{\beta\beta}\pi^{\tau}_{\eta} \;=\; \begin{cases} 0, & \text{for } \ \varphi(\beta; d_n) < 0 \\ -\frac{\tau}{\eta}, & \text{for } \ \varphi(\beta; d_n) \geq 0 \end{cases} . \tag{2.101}$$

3. The Finite Element Method

This chapter shall give an implementation oriented introduction on the topic of the finite element method. Literature on this subject is plentiful, the most significant for this thesis have proven to be the treatises by Zienkiewicz and Taylor [38], Bathe [2], Belytschko et al. [3], de Borst et al. [5], the book on numerical implementation methods by Press et al. [32], as well as the lecture notes by Celigoj [10] and Hammer [17].
The storage, which is available on a computer's hard disk is always limited. Hence, a mathematical model, which renders a material body as the sum of, theoretically, an infinit number of material points, each of which contains information on the constitutive state variables, represents a problem, which is not feasible. To reduce the space and time needed for a computation, a spatial discretization of the material body by way of a finite number of elements (m) is applied. These elements share the dimension of the body of discussion, and the larger the number of elements (m), the more accurate the geometric description will be. The spatial discretization is implemented into the weak formulation of the virtual work. The nodes are enumerated, first locally, specific to each finite element, defined in a local coordinate system, then globally. Hence, each element has to be subjected to a mapping, transforming the locally defined integration to the global coordinate system. Finally, the resulting integrals are subjected to a numerical integration, denoted *quadrature*. The resulting sparse matrices define the building blocks of a system of algebraic equations, which is solved for the constitutive state variables.

3.1. Interpolation with Lagrange Polynomials for Hexahedral Finite Elements

As described in the introduction to this chapter the first discretization is of a spatial character, where the material body $\mathbb{B}$ is regarded as the sum of a finite number of n volume elements $\mathbb{B}^{(m)}$ and the elemental border domains $\partial\mathbb{B}^{(m)}$

$$\mathbb{B} \approx \sum_{m=1}^{n} \mathbb{B}^{(m)} \quad \text{and} \quad \partial\mathbb{B} \approx \sum_{m=1}^{n} \partial\mathbb{B}^{(m)} . \tag{3.1}$$

The approach is schematically illustrated in Figure 3.1.

Consider a three-dimensional finite element with the local elemental, natural coordinates $\{r, s, t\}$, resulting in three independent components for the macroscopic displacement field $\boldsymbol{u}$, and one component each for the crack phase field d and the thermodynmaic force β. Taking into account time dependency, the vector of global unknowns yields the same values in the nodal points in both, the global coordinate system $\{x, y, z\}$, as well as the local coordinate system $\{r, s, t\}$

$$\mathfrak{u}(x, y, z; \mathrm{t}) = \mathfrak{u}[x(r), y(s), z(t); \mathrm{t})] . \tag{3.2}$$

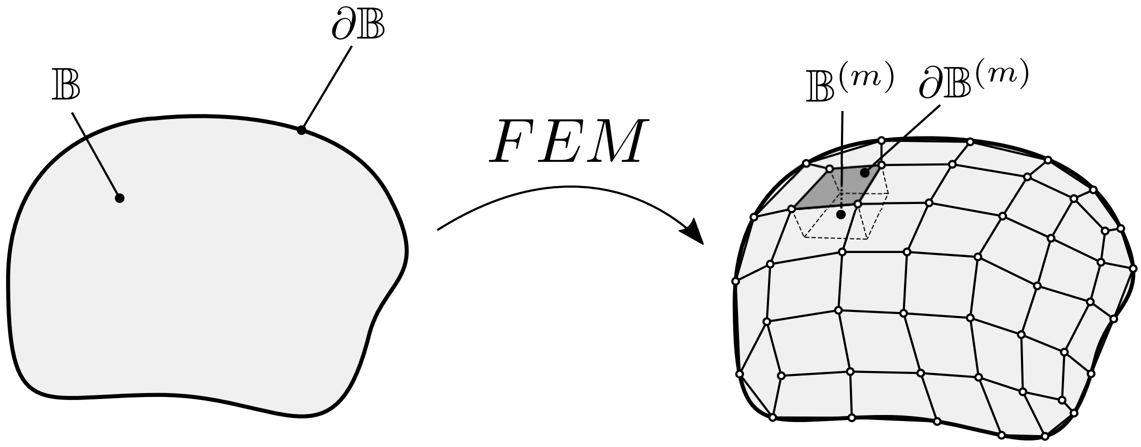


Figure 3.1. Spatial discretization of a material body $\mathbb{B}$ with finite elements.

The discretization towards the finite elements (m) demands for the generalized deformation vector inside the domain $\mathbb{B}^{(m)}$ to be estimated by the *displacement interpolation matrix* $\boldsymbol{H}(r,s,t)$ and the *local nodal state vector* $\hat{\mathfrak{u}}(\mathrm{t})$

$$\mathfrak{u}^{(m)}(r,s,t;\mathrm{t}) = \boldsymbol{H}^{(m)}(r,s,t)\cdot\hat{\mathfrak{u}}(\mathrm{t})\,. \tag{3.3}$$

The displacement interpolation matrix contains the so called shape functions $h_i(r,s,t)$, implemented as piecewise linear hat functions, which for the one dimensional case take the form

$$h_i(r_k) = \delta_{ik} = \begin{cases} 0, & \text{if } i \neq k \\ 1, & \text{if } i = k \end{cases}\,. \tag{3.4}$$

For higher dimensional elements, such as the three-dimensional hexahedral element, as depicted in figure (3.2), the shape function array is constructed from the multiplication of three one-dimensional shape functions

$$h_{\mathrm{i}}(r,s,t) = \frac{1}{2}(1+r_i\cdot r)\cdot\frac{1}{2}(1+s_i\cdot s)\cdot\frac{1}{2}(1+t_i\cdot t)\,. \tag{3.5}$$

An interpolation is denoted isoparametric, if the geometry and the displacements share the same interpolation matrix, which yields

$$\boldsymbol{x}^{(m)}(r,s,t;\mathrm{t}) = \boldsymbol{H}^{(m)}(r,s,t)\cdot\hat{\boldsymbol{x}}(\mathrm{t})\,. \tag{3.6}$$

Table 3.1 lists the local indicies assigned to the natural nodal coordinates for the hexahedral shape functions .

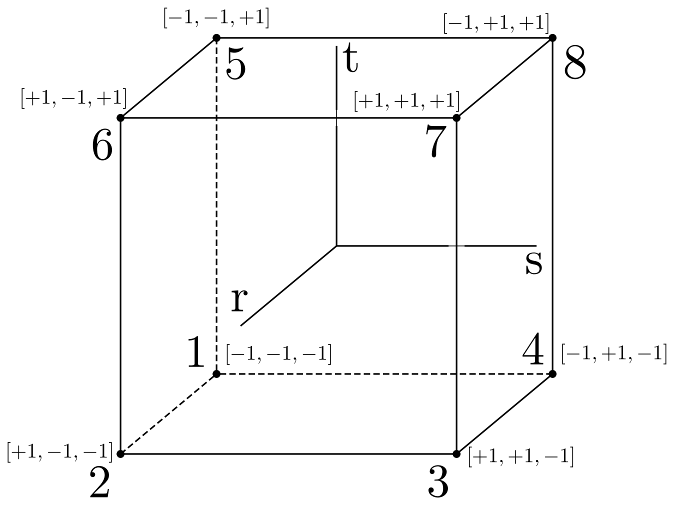


Figure 3.2. Hexahedral finite element with nodal coordinates and local indices.

local index	**node coordinate**
1	$[-1, -1, -1]$
2	$[+1, -1, -1]$
3	$[+1, +1, -1]$
4	$[-1, +1, -1]$
5	$[-1, -1, +1]$
6	$[+1, -1, +1]$
7	$[+1, +1, +1]$
8	$[-1, +1, +1]$

Table 3.1. Local indices with corresponding nodal coordinates for hexahedral shape functions.

3.2. Assembling the Equation System for Hexahedral Finite Elements

3.2.1. Displacement Interpolation Matrix

Using equations (3.3) and (3.5), the full elemental displacement interpolation matrix is expressed as

$$\mathfrak{u}^{(m)}(r,s,t;\mathrm{t}) = \underbrace{\left[\begin{array}{ccc:cc|c|c} h_1 & 0 & 0 & 0 & 0 & & \\ 0 & h_1 & 0 & 0 & 0 & & \\ 0 & 0 & h_1 & 0 & 0 & (...)_i & (...)_8 \\ \hdashline 0 & 0 & 0 & h_1 & 0 & & \\ 0 & 0 & 0 & 0 & h_1 & & \end{array}\right]}_{=:\boldsymbol{H}^{(m)}(r,s,t)_{[5x40]}} \cdot \hat{\mathfrak{u}}(\mathrm{t})\,. \tag{3.7}$$

The local nodal state vector of the finite element transforms into a $[40 \times 1]$ vector, which manifests itself in a $[5 \times 40]$ displacement interpolation matrix

$$\hat{\mathfrak{u}}(\mathrm{t}) = [u_1, v_1, w_1, d_1, \beta_1 \mid (...)_i \mid u_8, v_8, w_8, d_8, \beta_8]^T_{[40\times 1]}\;. \tag{3.8}$$

3.2.2. Strain-Displacement Matrix

As is shown in equation (2.88) the elemental constitutive state vector is constructed by multiplication of a differential operator matrix $\boldsymbol{D}$ with the elemental generalized displacement vector $\mathfrak{u}$. In Voigt notation this translates to

$$\mathfrak{c}^{(m)}(\mathfrak{u};t) = \boldsymbol{D} \cdot \mathfrak{u}^{(m)}(t)\,. \tag{3.9}$$

The differential operator matrix has already been outlined in equation (2.88). Acting on the higher dimensional shape function arrays, introduced in equation (3.5), this reads in componential notation

$$\boldsymbol{D} = \left[\begin{array}{ccc|cc} \frac{\partial}{\partial x} & 0 & 0 & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 & 0 & 0 \\ 0 & 0 & \frac{\partial}{\partial z} & 0 & 0 \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 & 0 & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} & 0 & 0 \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{\partial}{\partial x} & 0 \\ 0 & 0 & 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & 0 & \frac{\partial}{\partial z} & 0 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right] . \tag{3.10}$$

Inserting into above equation the relation between the displacement field and the nodal displacements (3.7), a new matrix $\boldsymbol{B}$, connecting the general constitutive state variables with the nodal displacements is introduced.

$$\begin{aligned} \mathfrak{c}^{(m)}(r,s,t;\mathrm{t}) &\approx \boldsymbol{B}^{(m)}(r,s,t)\cdot\hat{\mathfrak{u}}(\mathrm{t}) && (3.11)\\ \text{with}\quad \mathfrak{c}^{(m)} &= [\varepsilon_{xx},\varepsilon_{yy},\varepsilon_{zz},\varepsilon_{xy},\varepsilon_{yz},\varepsilon_{zx} \mid d,\partial_x d,\partial_y d,\partial_z d,\beta]^T . && (3.12) \end{aligned}$$

The expression is generated by multiplication of the differential operator matrix with the displacement interpolation matrix, resulting in a $[11 \times 40]$ array

$$[\boldsymbol{B}^{(m)}(r,s,t)]_{11x40} = [\boldsymbol{D}]_{11x5}\cdot[\boldsymbol{H}^{(m)}(r,s,t)]_{5x40} , \tag{3.13}$$

with the componential notation

$$\boldsymbol{B}^{(m)}(r,s,t;\mathrm{t}) = \left[\begin{array}{ccc|cc|c|c} \frac{\partial h_1}{\partial x} & 0 & 0 & 0 & 0 & & \\ 0 & \frac{\partial h_1}{\partial y} & 0 & 0 & 0 & & \\ 0 & 0 & \frac{\partial h_1}{\partial z} & 0 & 0 & & \\ \frac{\partial h_1}{\partial y} & \frac{\partial h_1}{\partial x} & 0 & 0 & 0 & & \\ 0 & \frac{\partial h_1}{\partial z} & \frac{\partial h_1}{\partial y} & 0 & 0 & & \\ \frac{\partial h_1}{\partial z} & 0 & \frac{\partial h_1}{\partial x} & 0 & 0 & (...)_i & (...)_8 \\ \hline 0 & 0 & 0 & h_1 & 0 & & \\ 0 & 0 & 0 & \frac{\partial h_1}{\partial x} & 0 & & \\ 0 & 0 & 0 & \frac{\partial h_1}{\partial y} & 0 & & \\ 0 & 0 & 0 & \frac{\partial h_1}{\partial z} & 0 & & \\ 0 & 0 & 0 & 0 & h_1 & & \end{array}\right] . \tag{3.14}$$

The partial differentials

$$\frac{\partial h_i}{\partial x_j} = \frac{\partial h_i}{\partial r_k}\frac{\partial r_k}{\partial x_j} = h_{i,k}(\boldsymbol{J}^{-1})_{k,j}\,, \tag{3.15}$$

contain the inverse of the Jacobian, which is equivalent to a deformation gradient and maps functions and vectors defined in the local coordinate system $\{r, s, t\}$ onto the global coordinate system $\{x, y, z\}$

$$\boldsymbol{J} = \boldsymbol{x} \otimes \nabla_{\boldsymbol{r}} \quad \text{with} \quad \boldsymbol{x} = [x\,y\,z]^T \quad \text{and} \quad \nabla_{\boldsymbol{r}} = [\frac{\partial}{\partial r}\,\frac{\partial}{\partial s}\,\frac{\partial}{\partial t}]^T\,, \tag{3.16a}$$

$$\Rightarrow \quad \boldsymbol{J} = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial s} & \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial s} & \frac{\partial y}{\partial t} \\ \frac{\partial z}{\partial r} & \frac{\partial z}{\partial s} & \frac{\partial z}{\partial t} \end{bmatrix} \quad \text{and} \quad \boldsymbol{J}^{-1} = \begin{bmatrix} \frac{\partial r}{\partial x} & \frac{\partial r}{\partial y} & \frac{\partial r}{\partial z} \\ \frac{\partial s}{\partial x} & \frac{\partial s}{\partial y} & \frac{\partial s}{\partial z} \\ \frac{\partial t}{\partial x} & \frac{\partial t}{\partial y} & \frac{\partial t}{\partial z} \end{bmatrix}. \tag{3.16b}$$

These relations hold to be equally true for the virtual unknowns and the virtual extended constitutive state vector

$$\delta\mathfrak{u}^{(m)}(r,s,t;\mathrm{t}) = \boldsymbol{H}^{(m)}(r,s,t)\cdot\delta\hat{\mathfrak{u}}(\mathrm{t})\,, \tag{3.17a}$$

$$\delta\mathfrak{c}^{(m)}(r,s,t;\mathrm{t}) = \boldsymbol{B}^{(m)}(r,s,t)\cdot\delta\hat{\mathfrak{u}}(\mathrm{t})\,. \tag{3.17b}$$

3.2.3. Coordinate Transformation

Subsequently integrals defined in the global coordinate system have to be transformed into integrals over the natural coordinates. For volume integrals this yields

$$\int\limits_{\mathbb{B}^{(m)}} \mathrm{d}V^{(m)} = \int\limits_{-1}^{+1}\int\limits_{-1}^{+1}\int\limits_{-1}^{+1} \det(\boldsymbol{J}(r,s,t))\;\mathrm{d}r\,\mathrm{d}s\,\mathrm{d}t\,. \tag{3.18}$$

Here the calculation of the determinant $\det(\boldsymbol{J}(r,s,t))$ is straight forward.

3.2.4. Residual Matrix and Stiffness Matrix

Omitting the surface forces and inserting the results of the finite element discretization into the equation for the residual function (2.85), subsequently transforms to

$$\begin{aligned}
&\delta\Pi^{\tau}_{\eta}(\mathfrak{u}) = \\
&\int\limits_{\mathbb{B}} \left\{\delta\mathfrak{c}^T\cdot\mathcal{S} - \delta\mathfrak{u}^T\cdot\mathcal{F}^B\right\}\;\mathrm{d}V \approx \\
&\delta\hat{\mathfrak{u}}^T\Big[\sum_{(m)}\Big(\int\limits_{\mathbb{B}^{(m)}}\left\{\boldsymbol{B}^{(m)^T}\cdot\mathcal{S}^{(m)} - \boldsymbol{H}^{(m)^T}\cdot\mathcal{F}^B\right\}\;\mathrm{d}V^{(m)}\Big)\Big] = \\
&\delta\hat{\mathfrak{u}}^T\Big[\sum_{(m)}\underbrace{\Big(\int\limits_{-1}^{+1}\int\limits_{-1}^{+1}\int\limits_{-1}^{+1}\left\{\boldsymbol{B}^{(m)^T}(r,s,t)\cdot\mathcal{S}^{(m)} - \boldsymbol{H}^{(m)^T}(r,s,t)\cdot\mathcal{F}^B\right\}\det(\boldsymbol{J}(r,s,t))\;\mathrm{d}r\,\mathrm{d}s\,\mathrm{d}t\Big)\Big]}_{=:\mathcal{F}^{(m)}_{int}-\mathcal{F}^{(m)}_{ext}}.
\end{aligned} \tag{3.19}$$

The elemental contributions of the internal forces $\mathcal{F}_{int}^{(m)}$ and the external forces $\mathcal{F}_{ext}^{(m)}$ are summarized into the internal and external force vectors $\mathcal{F}_{int}$ and $\mathcal{F}_{ext}$, respectively

$$\delta\Pi_{\eta}^{\tau}(\mathfrak{u}) = \delta\hat{\mathfrak{u}}^T \sum_{(m)} (\mathcal{F}_{int}^{(m)} - \mathcal{F}_{ext}^{(m)}) = \delta\hat{\mathfrak{u}}^T \left(\mathcal{F}_{int} - \mathcal{F}_{ext}\right). \tag{3.20}$$

The increment of the variation of the potential of the internal forces yields

$$\begin{aligned}
&\Delta(\delta\Pi_{\eta}^{\tau}(\mathfrak{u})) = \\
&\int_{\mathbb{B}} \left\{\delta\mathfrak{c}^T : \mathbb{C}_{\eta}^{\tau} : \Delta\mathfrak{c}\right\} \,\mathrm{d}V \approx \\
&\delta\hat{\mathfrak{u}}^T \Big[\sum_{(m)} \Big(\int_{\mathbb{B}^{(m)}} \left\{\boldsymbol{B}^{(m)^T} \!: \mathbb{C}_{\eta}^{\tau(m)} : \boldsymbol{B}^{(m)}\right\} \,\mathrm{d}V^{(m)}\Big)\Big] \Delta\hat{\mathfrak{u}} = \\
&\delta\hat{\mathfrak{u}}^T \underbrace{\Big[\sum_{(m)} \Big(\underbrace{\int_{-1}^{+1}\int_{-1}^{+1}\int_{-1}^{+1} \left\{\boldsymbol{B}^{(m)^T}(r,s,t) \!: \mathbb{C}_{\eta}^{\tau(m)} : \boldsymbol{B}^{(m)}(r,s,t)\right\} \,\det(\boldsymbol{J}(r,s,t)) \,\mathrm{d}r\,\mathrm{d}s\,\mathrm{d}t}_{=:\mathcal{K}^{(m)}}\Big)\Big]}_{=:\mathcal{K}} \Delta\hat{\mathfrak{u}}\,.
\end{aligned} \tag{3.21}$$

The elemental stiffness matrices $\mathcal{K}^{(m)}$ are summarized into a global stiffness matrix $\mathcal{K}$. With the residual and the stiffness matrix defined, the nonlinear equation system is eastablished as

$$\mathcal{K} \cdot \Delta\hat{\mathfrak{u}} = \mathcal{F}_{ext} - \mathcal{F}_{int} \quad \text{with} \quad \Delta\hat{\mathfrak{u}} = \hat{\mathfrak{u}}(t_{n+1}) - \hat{\mathfrak{u}}(t_n)\,. \tag{3.22}$$

and subjected to the the Newton-Raphson method, to solve for the incremental solution vector $\Delta\hat{\mathfrak{u}}$.

3.2.5. Numerical Integration

Up until this point the nature of the integrals over the single finite elements volume and surface have not been subjected to further analysis. These integrals, generaly non-linear in nature, are approximated by calculating the values of the function in a set of n so called quadrature points x_i and multiplying the results with n weights α_i

$$\int_a^b F(x)\,\mathrm{d}x = \sum_{i=1}^{n} \alpha_i F(x_i) + R_n\,. \tag{3.23}$$

When dealing with natural coordinates $r \in [-1;+1]$ a linear mapping procedure for the global coordinates $x \in [a;b]$, as well as for the weights α_i is introduced

$$x = \frac{b+a}{2} + \frac{b-a}{2} \cdot r \quad \text{and} \quad \overline{\alpha}_i = \frac{b-a}{2}\alpha_i\,. \tag{3.24}$$

Substituting the integration variable, results in the natural occurence of the determinant of the Jacobian

$$\frac{\mathrm{d}x}{\mathrm{d}r} = \frac{b-a}{2} \quad \Rightarrow \quad \mathrm{d}x = \frac{b-a}{2}\mathrm{d}r = \frac{\mathrm{d}x}{\mathrm{d}r}\mathrm{d}r = \det(\boldsymbol{J})\mathrm{d}r\,. \tag{3.25}$$

The integral of a function $F(x(r)) = \overline{F}(r)$ transformed from natural coordinates into a global coordinate system is approximated via

$$\begin{aligned} \int\limits_a^b F(x)\,\mathrm{d}x &= \int\limits_{-1}^{+1} \overline{F}(r)\det(\boldsymbol{J})\,\mathrm{d}r \\ &= \sum_{i=1}^{n} \overline{\alpha}_i \overline{F}(r_i)\det(\boldsymbol{J})\,. \end{aligned} \tag{3.26}$$

For the elemental internal and external forces put forth in equation (3.19) this translates to

$$\begin{aligned} \mathcal{F}_{int}^{(m)} - \mathcal{F}_{ext}^{(m)} = \sum_{i=1}^{l}\sum_{j=1}^{m}\sum_{k=1}^{n} \; \overline{\alpha}_i\,\overline{\alpha}_j\,\overline{\alpha}_k \Big\{ & \boldsymbol{B}^{(m)^T}(r_i,s_j,t_k)\cdot\boldsymbol{\mathcal{S}}^{(m)}(r_i,s_j,t_k) - \\ & -\boldsymbol{H}^{(m)^T}(r_i,s_j,t_k)\cdot\mathcal{F}^B \Big\} \det(\boldsymbol{J}(r_i,s_j,t_k)) \end{aligned} \tag{3.27}$$

while for the elemental stiffness matrix introduced in equation (3.21) above statement is formulated to yield

$$\begin{aligned} \mathcal{K}^{(m)} = \sum_{i=1}^{l}\sum_{j=1}^{m}\sum_{k=1}^{n} \overline{\alpha}_i\,\overline{\alpha}_j\,\overline{\alpha}_k \Big\{ & \boldsymbol{B}^{(m)^T}(r_i,s_j,t_k) \boldsymbol{:} \, \mathbb{C}_{\eta}^{\tau(m)}(r_i,s_j,t_k) \boldsymbol{:} \\ & \boldsymbol{:} \boldsymbol{B}^{(m)}(r_i,s_j,t_k) \Big\} \det(\boldsymbol{J}(r_i,s_j,t_k))\,. \end{aligned} \tag{3.28}$$

For the hexahedral finite element used in the numerical implementation a $(2\times2\times2)$ Gauß quadrature was used.

4. Numerical Implementation. The Single Edge Notched Tension Test

In this chapter a numerical implementation of the rate-dependent model with treshold function is put forth, using the open source finite element library *deal.II*. The geometry, parameters, constants and boundary conditions are specified for the *single edge notched tension test*. The finite element library deal.II offers an environment for the solution of partial differential equations, with the possibility of using adaptive finite elements. It was developed by Wolfgang Bangerth et al. [1], [12]. The basic finite elements implemented in the code are quadrilaterals and hexahedrons, without the possibility of using triangles or tetrahedras.

4.1. Geometry, Mesh and Basic Setup

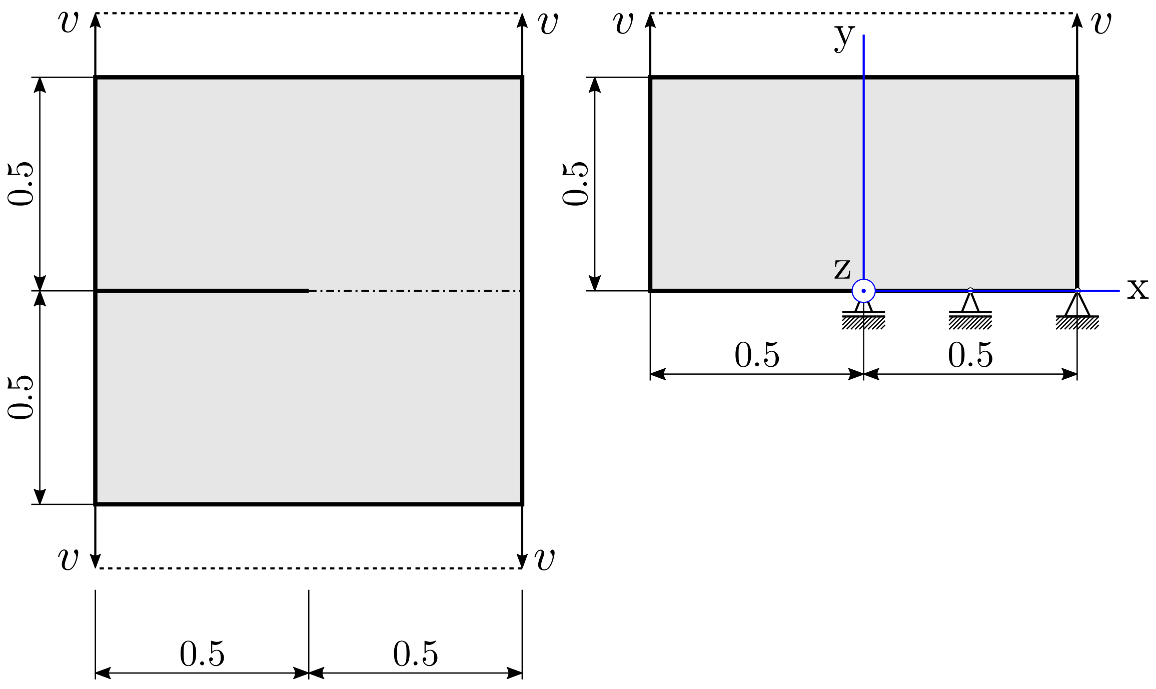


Figure 4.1. Single edge notched tension test

The single edge notched tension test is a displacement driven test, of which the geometry is distinguished by a quadratic plate, with a 0.5 mm long incision along the plane of symmetry, as depicted in Figure 4.1. The lower edge of the plate is mounted with floating bearings, with a fixed bearing positioned at the lower right corner of the plate. Making use of the symmetry condition, only the upper halve of the plate has to be considered, which reduces computation time significantly. The displacement step size is set to be $\Delta v = 1.0 \times 10^{-5}\, mm$, with a time step size of $\tau = 1.0 \times 10^{-5}\, s$. The two Lamé constants for a steel plate, with a Young's modulus of $E = 210\, kN/mm^2$ and a Poisson's ratio of $\nu = 0.3$, are $\lambda = 121.15\, kN/mm^2$ and $\mu = 80.77\, kN/mm^2$. The a critical energy release rate is chosen to be $g_c = 2.7 \times 10^{-3}\, kN/mm$. The computation was executed for 5000 cubic elements, with an aspect ratio of 1 and a side length of $h = 5.0 \times 10^{-3}\, mm$, as depicted in Figure 4.2. The calculation is carried out for a length scale parameter of $l = 0.0375\, mm$, whereby the condition $h < l/2$ is satisfied.

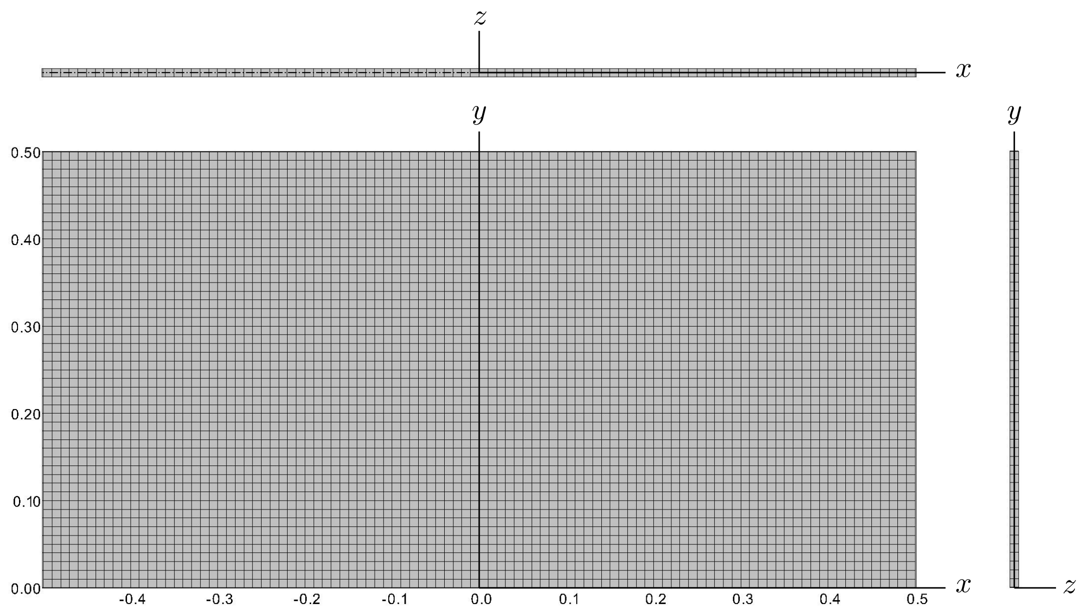


Figure 4.2. Mesh for the single edge notched tension test, stemming from 100 horizontal and 50 vertical subdivisions, resulting in 5000 equally sized cubic elements.

The shape functions are chosen to be identical for the three field variables, and are standard Lagrange interpolation polynomials of order one, i.e. linear functions. The numerical integration used is the Gauß-Legendre quadrature, with the quadrature of order two. The tolerances for the residual and the Newton update are set to be

$$\left\| {}^{t+\Delta t}R^{(i)} \right\| < 10^{-9} \quad \text{and} \quad \left\| \Delta \mathbf{u}^{(i+1)} \right\| < 10^{-6} \tag{4.1}$$

4.2. Boundary Conditions

The finite elements used in the implementation are cubic hexahedrons, as depicted in Figure 3.2. In order to represent the two-dimensional steel plate as seen in Figure 4.1, a plane strain state has to be induced by constraining the displacements w in the z-coordinate direction to zero. Thus, every strain component in z-direction is zero, a property which translated to Hooke's law reads

$$\begin{aligned}
\text{I. } \varepsilon_{xx} &= \frac{\partial u}{\partial x} = \frac{1}{E}(+\sigma_{xx} - \nu\sigma_{yy} - \nu\sigma_{zz}) \\
\text{II. } \varepsilon_{yy} &= \frac{\partial v}{\partial y} = \frac{1}{E}(-\nu\sigma_{xx} + \sigma_{yy} - \nu\sigma_{zz}) \\
\text{III. } \varepsilon_{zz} &= \frac{\partial w}{\partial z} = \frac{1}{E}(-\nu\sigma_{xx} - \nu\sigma_{yy} + \sigma_{zz}) \overset{!}{=} 0 \\
\text{IV. } \gamma_{xy} &= \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} = \frac{1}{G} = \sigma_{xy} \\
\text{V. } \gamma_{yz} &= \frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \overset{!}{=} 0 \quad \Rightarrow \quad \sigma_{yz} = 0 \\
\text{VI. } \gamma_{zx} &= \frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} \overset{!}{=} 0 \quad \Rightarrow \quad \sigma_{zx} = 0 \\
\text{with } \varepsilon_{ij} &= \frac{1}{2}\gamma_{ij}
\end{aligned} \tag{4.2}$$

The normal stresses are formulated with respect to the normal strains

$$\begin{aligned}
\sigma_{xx} &= \frac{E}{(1+\nu)(1-2\nu)}\left[(1-\nu)\varepsilon_{xx} + \nu\varepsilon_{yy}\right] \\
\sigma_{yy} &= \frac{E}{(1+\nu)(1-2\nu)}\left[\nu\varepsilon_{xx} + (1-\nu)\varepsilon_{yy}\right] \\
\sigma_{zz} &= \frac{\nu E}{(1+\nu)(1-2\nu)}\left[\varepsilon_{xx} + \varepsilon_{yy}\right]
\end{aligned} \tag{4.3}$$

The resulting characteristic strain and stress tensors for the plane strain state read

$$\boldsymbol{\varepsilon}^{PSS} = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{xy} & 0 \\ \varepsilon_{yx} & \varepsilon_{yy} & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad \boldsymbol{\sigma}^{PSS} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & 0 \\ \sigma_{yx} & \sigma_{yy} & 0 \\ 0 & 0 & \sigma_{zz} \end{bmatrix} \tag{4.4}$$

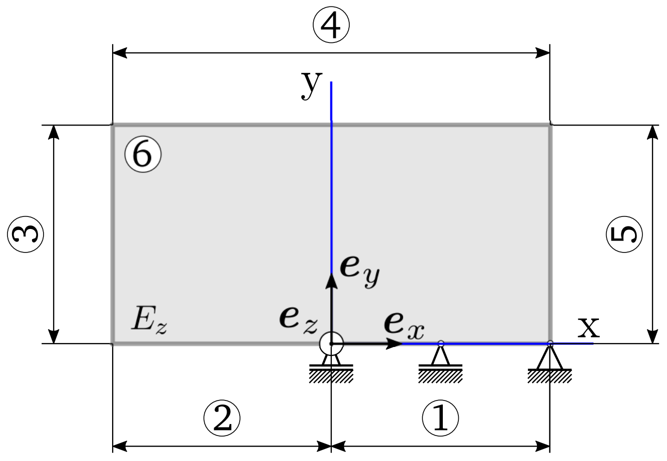


Figure 4.3. Assigned boundary identification numbers.

Figure 4.3 depicts the steel plate, with boundary identification numbers assigned to it. Implementationwise the plane strain state is achieved by locking all displacements in z-direction along the top and bottom surface (6), which are parallel to the surface E_z spanned by the unit vectors $\boldsymbol{e}_x$ and $\boldsymbol{e}_y$. The displacement in y-direction is locked along the surface (1) to take into account the floating bearings, while the x-displacement is only locked at the lower right corner, corresponding to the fixed bearing. No boundary conditions are assigned to the crack phase field d or the thermodynamic dissipative force β.

4.3. Results and Discussion.

Regarding the shape of the load function itself, linear elastic loading is observed, which migrates into nonlinear elastic loading and ultimately mimics plastic degradation, at which point the solution diverges for numerical reasons. The increase of the dynamic viscosity in steps of $\Delta\eta = 1.0 \times 10^{-6}\, kN\, s/mm^2$ is displayed graphically in Figure 4.4. The maximal admissible force, increases with the viscosity. Figure 4.5 shows the crack phase field for a length scale parameter of $l = 0.0375\, mm$ and a viscosity of $\eta = 1.0 \times 10^{-6}\, kN\, s/mm^2$ at varying values of the displacement. Since the solution is computed for only 5000 elements, it is not possible to adequately resolve the crack pattern itself.

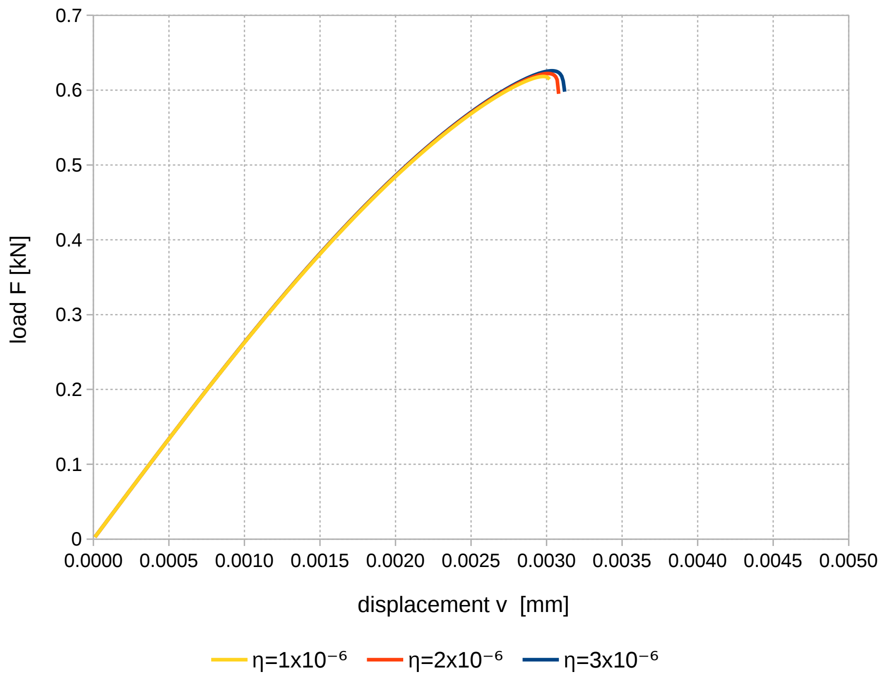


Figure 4.4. Plot of the calculated loads at the top boundary cells, for varying dynamic viscosity. The viscosity is increased in steps of $\Delta\eta = 1.0 \times 10^{-6}\, kN\, s/mm^2$. The maximum force $F = 618.301\, N$ for a viscosity of $\eta = 1.0 \times 10^{-6}\, kN\, s/mm^2$ is found at a displacement $v = 2.98 \cdot 10^{-3}\, mm$. The maximum force $F = 622.268\, N$ for a viscosity of $\eta = 2.0 \times 10^{-6}\, kN\, s/mm^2$ is found at a displacement of $v = 3.01 \cdot 10^{-3}\, mm$. The maximum force $F = 625.871\, N$ for a viscosity of $\eta = 3.0 \times 10^{-6}\, kN\, s/mm^2$ is found at a displacement of $v = 3.04 \cdot 10^{-3}\, mm$.

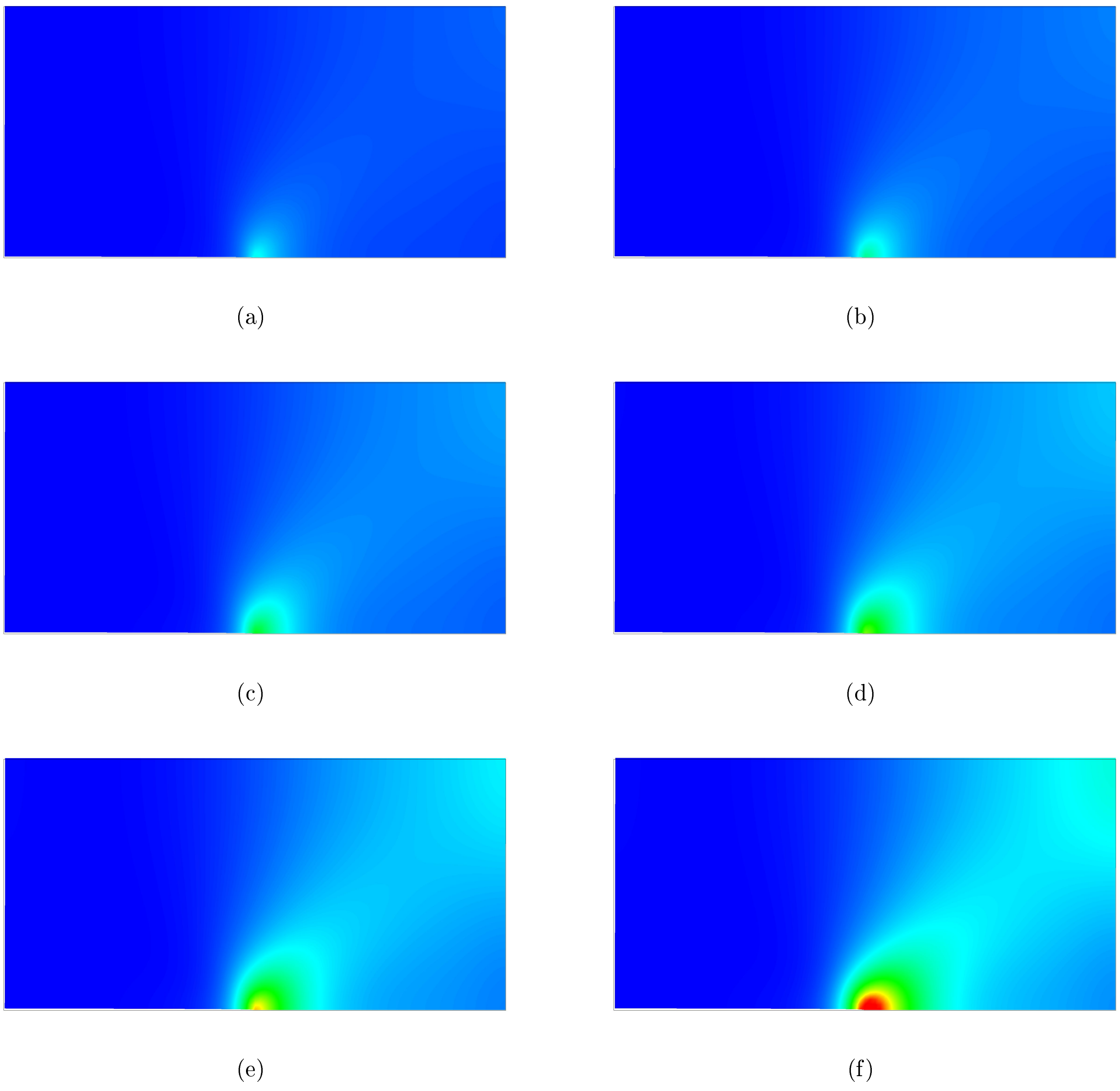


Figure 4.5. Results for the single edge notched tension test for $l = 0.0375\,mm$ and $\eta = 1.0 \times 10^{-6}\,kN\,s/mm^2$. The crack phase field d is displaced at a displacement of (a) $v = 1.75\times 10^{-3}\,mm$, (b) $v = 2.00 \times 10^{-3}\,mm$, (c) $v = 2.25 \times 10^{-3}\,mm$, (d) $v = 2.50 \times 10^{-3}\,mm$, (e) $v = 2.75 \times 10^{-3}\,mm$, (f) $v = 3.00 \times 10^{-3}\,mm$.

5. Concluding Remarks

Beginning with a short recapitulation of the first and second law of thermodynamics a framework is discussed, which is then used to deduce Griffith‘s facture criterion. The crack phase field and the dual thermodynamic force are introduced as microscopic field variables, whereby a destinction was made between sharp and difussive crack topology. Based on a thermodynamically consistent multi-field formulation, a stored energy and a dissipation function were introduced, expanding the macroscopic displacmenet field by the two aforementioned microscopic quantities. In line with the gradient-extended theory, the derivatives of the internal variables (“crack phase field”) were not limited to the temporal derivative, but extended to include the geometric gradient, thus considering nonlocalities. Isotropic degradation of the stored bulk energy was implemented, while a regularized threshold function was introduced, of which the attributes were the subject of in-depth discussion. Resting upon the balance of total power, a time-discretization was applied, which in turn was subjected to the variational principle, resulting in a stationary problem. The coupled Euler equations were derived from the incremental potential. A spatial discretization towards finite elements requires the field variables to be approximated by so called shape functions. For the special case of hexahedral elements the equation system was assembled, with in detail descriptions of the displacement interpolation matrix, the strain-displacement matrix, as well as the tangent and residual matrix.
The numerical implementation was carried out for the example of the single edge notched tension test, using the finite element library deal.II. The results were computed for varying kinematic viscosities.

Appendix A.

Code in C++

```
/*
 * Created in deal.II Library, 8.4.1
 * Numerical Implementation of Variational-Based Phase Field
 * Modelling of Fracture
 */

#include "deal.II/base/parameter_handler.h"
#include "deal.II/base/quadrature_lib.h"
#include "deal.II/base/symmetric_tensor.h"

#include <deal.II/dofs/dof_handler.h>
#include "deal.II/dofs/dof_renumbering.h"
#include "deal.II/dofs/dof_tools.h"
#include <deal.II/dofs/dof_accessor.h>

#include <deal.II/grid/tria_accessor.h>
#include <deal.II/grid/tria_iterator.h>
#include "deal.II/grid/grid_generator.h"
#include <deal.II/grid/grid_tools.h>
#include <deal.II/grid/manifold_lib.h>
#include "deal.II/grid/tria.h"
#include "deal.II/grid/grid_out.h"
#include <deal.II/grid/grid_in.h>

#include "deal.II/fe/fe_q.h"
#include "deal.II/fe/fe_system.h"
#include "deal.II/fe/fe_values.h"
#include "deal.II/fe/mapping_q_eulerian.h"

#include "deal.II/lac/block_vector.h"
#include "deal.II/lac/block_sparse_matrix.h"
#include "deal.II/lac/full_matrix.h"
#include "deal.II/lac/constraint_matrix.h"
#include "deal.II/lac/solver_gmres.h"
#include "deal.II/lac/sparse_direct.h"
#include "deal.II/lac/precondition.h"

#include "deal.II/numerics/vector_tools.h"
#include "deal.II/numerics/data_out.h"
```

```
#include "deal.II/numerics/data_postprocessor.h"

#include "libfest/output.h"
#include "libfest/teestream.h"

#include <iostream>
#include <fstream>
#include <iomanip>

#include <map>


namespace crackprop
{
  using namespace dealii;


  namespace StandardFunctions
  {
    double ramp_function(const double &arg) { return arg>=0 ? arg : 0; }
    double step_function(const double &arg) { return arg>=0 ? 1 : 0; }

  }


  namespace Parameters
  {


    struct BoundaryConditionControl
    {
      double     displacement_start;
      double     displacement_end;
      double     displacement_stepsize;
      double     time_start;
      double     velocity;

      static void declare_parameters(ParameterHandler &prm);
             void parse_parameters (ParameterHandler &prm);
    };

    void BoundaryConditionControl::declare_parameters(ParameterHandler
       &prm)
    {
      prm.enter_subsection("Boundary Condition Control");
      {
        prm.declare_entry("Displacement Start", "0.0",
                          Patterns::Double(0),
                         "Displacement Start in Millimeters");
        prm.declare_entry("Displacement End", "0.008",
                         Patterns::Double(0),
                        "Displacement End in Millimeters");
      prm.declare_entry("Displacement Stepsize", "1e-5",
                         Patterns::Double(1e-10),
                        "Displacement Stepsize in Millimeters");
      prm.declare_entry("Start Time", "0",
                        Patterns::Double(0),
                        "Starting Time in Seconds");
      prm.declare_entry("Velocity", "1e3",
                        Patterns::Double(1e-10),
                        "Velocity in mm/s");
   }
   prm.leave_subsection();
}

void BoundaryConditionControl::parse_parameters(ParameterHandler &prm)
{
   prm.enter_subsection("Boundary Condition Control");
   {
      displacement_start  = prm.get_double("Displacement Start");
      displacement_end    = prm.get_double("Displacement End");
      displacement_stepsize = prm.get_double("Displacement Stepsize");
      time_start          = prm.get_double("Start Time");
      velocity            = prm.get_double("Velocity");
   }
   prm.leave_subsection();
}


struct Geometry
{
   double      sidelength;
   unsigned int subdivisions_horizontally;
   unsigned int subdivisions_vertically;

   static void declare_parameters(ParameterHandler &prm);
          void parse_parameters (ParameterHandler &prm);
};

void Geometry::declare_parameters(ParameterHandler &prm)
{
   prm.enter_subsection("Geometry");
   {
      prm.declare_entry("Sidelength", "1",
                        Patterns::Double(0.0),
                        "Square Sidelength in mm");
      prm.declare_entry("Subdivisions horizontally", "5",
                        Patterns::Integer(0),
                        "Number of Subdivisions in the horizontal
                            direction");
      prm.declare_entry("Subdivisions vertically", "10",
                        Patterns::Integer(0),
                        "Number of Subdivisions in the vertical
                            direction");
   }
   prm.leave_subsection();
}

void Geometry::parse_parameters(ParameterHandler &prm)
{
   prm.enter_subsection("Geometry");
   {
      sidelength        = prm.get_double("Sidelength");
      subdivisions_horizontally = prm.get_integer("Subdivisions
          horizontally");
      subdivisions_vertically = prm.get_integer("Subdivisions
          vertically");
   }
   prm.leave_subsection();
}


struct Material
{
 double eta;
   double lambda;
   double mu;
   double gc;
   double l;
   double k;

   static void declare_parameters(ParameterHandler &prm);
          void parse_parameters (ParameterHandler &prm);
};

void Material::declare_parameters(ParameterHandler &prm)
{
   prm.enter_subsection("Material");
   {
      prm.declare_entry("Eta", "1e-6",
                         Patterns::Double(0.0),
                        "Viskosity Eta in kN s/mm^2");
      prm.declare_entry("Lambda", "121.15",
                        Patterns::Double(0.0),
                        "Bulk modulus in kN/mm^2");
      prm.declare_entry("Mu", "80.77",
                        Patterns::Double(0.0),
                        "Shear modulus in kN/mm^2");
      prm.declare_entry("Gc", "2.7e-3",
                        Patterns::Double(0.0),
                        "Critical energy release rate in kN/mm");
      prm.declare_entry("L", "0.0075",
                        Patterns::Double(0.0),
                        "Length scale parameter in mm");
      prm.declare_entry("K", "1e-10",
                        Patterns::Double(0.0),
                        "Degradation parameter k");
   }
   prm.leave_subsection();
}

void Material::parse_parameters(ParameterHandler &prm)
{
   prm.enter_subsection("Material");
   {
    eta  = prm.get_double("Eta");
      lambda = prm.get_double("Lambda");
      mu     = prm.get_double("Mu");
      gc     = prm.get_double("Gc");
      l    = prm.get_double("L");
      k    = prm.get_double("K");
   }
   prm.leave_subsection();
}


struct NonlinearSolver
{
   unsigned int max_newton_iterations;
   double      tol_residual;
   double      tol_update;

   static void declare_parameters(ParameterHandler &prm);
          void parse_parameters(ParameterHandler &prm);
};

void NonlinearSolver::declare_parameters(ParameterHandler &prm)
{
   prm.enter_subsection("Nonlinear Solver");
   {
      prm.declare_entry("Max Newton Iterations", "10",
                        Patterns::Integer(0),
                        "Number of Newton-Raphson iterations allowed");
      prm.declare_entry("Tolerance Residual", "1.0e-9",
                        Patterns::Double(0.0),
                        "Force residual tolerance");
      prm.declare_entry("Tolerance Displacement", "1.0e-6",
                        Patterns::Double(0.0),
                        "Displacement error tolerance");
   }
   prm.leave_subsection();
}

void NonlinearSolver::parse_parameters(ParameterHandler &prm)
{
   prm.enter_subsection("Nonlinear Solver");
   {
      max_newton_iterations = prm.get_integer("Max Newton
          Iterations");
      tol_residual       = prm.get_double ("Tolerance Residual");
      tol_update         = prm.get_double ("Tolerance Displacement");
   }
   prm.leave_subsection();
}


struct FESystem
{
   unsigned int poly_degree;
   unsigned int quad_order;

   static void declare_parameters(ParameterHandler &prm);
          void parse_parameters(ParameterHandler &prm);
};

void FESystem::declare_parameters(ParameterHandler &prm)
{
   prm.enter_subsection("Finite Element System");
   {
      prm.declare_entry("Polynomial Degree", "1",
                        Patterns::Integer(0),
                        "Displacement system polynomial order");
      prm.declare_entry("Quadrature Order", "2",
                        Patterns::Integer(0),
                        "Gauss quadrature order");
   }
   prm.leave_subsection();
}

void FESystem::parse_parameters(ParameterHandler &prm)
{
   prm.enter_subsection("Finite Element System");
   {
      poly_degree = prm.get_integer("Polynomial Degree");
      quad_order = prm.get_integer("Quadrature Order");
   }
   prm.leave_subsection();
}
```

```
  struct AllParameters :
    public BoundaryConditionControl,
    public Geometry,
    public Material,
    public NonlinearSolver,
    public FESystem
    {
      AllParameters(const std::string &input_file);
      static void declare_parameters(ParameterHandler &prm);
           void parse_parameters(ParameterHandler &prm);
    };

  AllParameters::AllParameters(const std::string &input_file)
  {
    ParameterHandler prm;
    declare_parameters(prm);
    prm.read_input(input_file);
    parse_parameters(prm);
    std::ofstream out("parameters.log");
    prm.print_parameters(out, ParameterHandler::Text);
  }

  void AllParameters::declare_parameters(ParameterHandler &prm)
  {
    BoundaryConditionControl::declare_parameters(prm);
    Geometry::declare_parameters(prm);
    Material::declare_parameters(prm);
    NonlinearSolver::declare_parameters(prm);
    FESystem::declare_parameters(prm);
  }

  void AllParameters::parse_parameters(ParameterHandler &prm)
  {
    BoundaryConditionControl::parse_parameters(prm);
    Geometry::parse_parameters(prm);
    Material::parse_parameters(prm);
    NonlinearSolver::parse_parameters(prm);
    FESystem::parse_parameters(prm);
  }
}



class BoundaryConditionControl
{
public:
BoundaryConditionControl(const double displacement_start,
                      const double displacement_end,
                      const double displacement_stepsize,
                      const double velocity,
                        const double time_start);

double get_start_displacement() const { return displacement_start; }
double get_final_displacement() const { return displacement_end; }
double get_displacement_stepsize() const { return
    displacement_stepsize; }

double get_velocity() const { return velocity; }

double get_start_time()         const { return time_start; }
double get_time_stepsize()      const { return time_stepsize; }
double get_total_timestep_number() const { return
    timestep_number_total; }
double get_final_time()         const { return time_end; }

double get_current_displacement() const { return displacement_current; }
double get_current_time()         const { return time_current; }
double get_current_timestep_number() const { return
    timestep_number_current; }
void increment()
{

   time_current       += time_stepsize;
   displacement_current += displacement_stepsize;
   ++timestep_number_current;
}

private:
const double displacement_start;
const double displacement_end;
      double displacement_stepsize;

const double velocity;

const double time_start;
      double time_stepsize;

const unsigned int timestep_number_total;
const double      time_end;

double      displacement_current;
double      time_current;
unsigned int timestep_number_current;
};


BoundaryConditionControl::BoundaryConditionControl
   (const double displacement_start,
    const double displacement_end,
    const double displacement_stepsize,
    const double velocity,
    const double time_start)
:
displacement_start(displacement_start),
displacement_end(displacement_end),
displacement_stepsize(displacement_stepsize),
velocity(velocity),
time_start(time_start),
time_stepsize(displacement_stepsize/velocity),
timestep_number_total(
    (displacement_end-displacement_start)/displacement_stepsize ),
time_end( time_start+timestep_number_total*time_stepsize )
{
   displacement_current  = displacement_start;
   time_current          = time_start;
   timestep_number_current = 0;
}



template <int dim>
class IncrementalBoundaryValues : public Function<dim>
{
public:
   IncrementalBoundaryValues (const double displacement_stepsize);
   virtual ~IncrementalBoundaryValues() {}

   virtual void vector_value (const Point<dim> &p,
                                 Vector<double> &values) const;

   virtual void vector_value_list (const std::vector<Point<dim> >
       &points,
                                      std::vector<Vector<double> >
                                          &value_list) const;


private:
   const double displacement_stepsize;
};


template <int dim>
IncrementalBoundaryValues<dim>::
IncrementalBoundaryValues (const double displacement_stepsize)
:
Function<dim> (dim+2),
displacement_stepsize(displacement_stepsize)
{ }

template <int dim>
```

```
void IncrementalBoundaryValues<dim>::
     vector_value (const Point<dim>    &/*p*/,
                        Vector<double> &values) const
{
   Assert (values.size() == dim+2, ExcDimensionMismatch (values.size(),
       dim));

   values   = 0;
   values(1) = displacement_stepsize;
}

template <int dim>
void IncrementalBoundaryValues<dim>::
     vector_value_list (const std::vector<Point<dim> > &points,
                         std::vector<Vector<double> > &value_list)
                            const
{
   const unsigned int n_points = points.size();

   Assert (value_list.size() == n_points,
           ExcDimensionMismatch (value_list.size(), n_points));

   for (unsigned int p=0; p<n_points; ++p)
      IncrementalBoundaryValues<dim>::vector_value (points[p],
                                                   value_list[p]);
}


struct NewtonError
{
   NewtonError()
   :
   norm(1.0), u(1.0), dcpf(1.0), beta(1.0)
   {}
   void reset()
   {
      norm  = 1.0;
      u     = 1.0;
      dcpf  = 1.0;
      beta  = 1.0;
   }

   void normalise(const NewtonError &reference)
   {
      if (reference.norm != 0.0)
         norm /= reference.norm;
      if (reference.u != 0.0)
         u /= reference.u;
      if (reference.dcpf != 0.0)
         dcpf /= reference.dcpf;
      if (reference.beta != 0.0)
         beta /= reference.beta;
   }
   double norm, u, dcpf, beta;
};

//   Alternative class for elasticity tensor:
//   template <int dim>
//   class StandardTensors
//   {
//   public:
//
//     static const SymmetricTensor<2, dim> I;
//     static const SymmetricTensor<4, dim> IxI;
//     static const SymmetricTensor<4, dim> II;
////     static const SymmetricTensor<4, dim> dev_P;
//   };
//
//   template <int dim>
//   const SymmetricTensor<2, dim>
//   StandardTensors<dim>::I = unit_symmetric_tensor<dim>();
//
//   template <int dim>
//   const SymmetricTensor<4, dim>
//   StandardTensors<dim>::IxI = outer_product(I, I);
//
//   template <int dim>
//   const SymmetricTensor<4, dim>
//   StandardTensors<dim>::II = identity_tensor<dim>();

template <int dim>
class ElasticityTensor
{
public:
   ElasticityTensor(const double lambda, const double mu)
   {
      for (unsigned int i=0; i<dim; ++i)
         for (unsigned int j=0; j<dim; ++j)
            for (unsigned int k=0; k<dim; ++k)
               for (unsigned int l=0; l<dim; ++l)
                  entries[i][j][k][l] = (((i==k) && (j==l) ? mu : 0.0) +
                                         ((i==l) && (j==k) ? mu : 0.0) +
                                         ((i==j) && (k==l) ? lambda : 0.0));
   }

   SymmetricTensor<4,dim> get_entries() const { return entries; }
private:
   SymmetricTensor<4,dim> entries;
};
```

```
template <int dim>
class Materialcrackprop
{
public:
   Materialcrackprop(const double eta,
              const double lambda,
                    const double mu,
                    const double gc,
                    const double l,
                    const double k)
   :
   eta         (eta),
   lambda      (lambda),
   mu       (mu),
   gc       (gc),
  l            (l),
   k           (k),
   epsilon_newton(SymmetricTensor<2,dim>()),
   dcpf_newton(0.0),
   grad_dcpf_newton(Tensor<1,dim>()),
   beta_newton(0.0),
   dcpf_n(0.0),

   CC(lambda,mu)
   { }

   void update_material_data_newton(const SymmetricTensor<2, dim>
       &epsilon_in,
                                  const double                &dcpf_in,
                                  const Tensor<1,dim>
                                      &grad_dcpf_in,
                                  const double                &beta_in)
   {
      epsilon_newton   = epsilon_in;
      dcpf_newton      = dcpf_in;
      grad_dcpf_newton = grad_dcpf_in;
      beta_newton      = beta_in;
   }

   void update_material_data_timestep(const double &dcpf_in)
   {
      dcpf_n = dcpf_in;
   }

   ElasticityTensor<dim> get_CC () const { return CC; }

   double get_eta      () const { return eta;  }
   double get_lambda     () const { return lambda; }
  double get_mu         () const { return mu;    }
  double get_gc         () const { return gc;    }
  double get_l          () const { return l;  }
  double get_k          () const { return k;     }


  SymmetricTensor<2,dim> get_epsilon_newton () const { return
     epsilon_newton; }
  double                 get_dcpf_newton     () const { return
     dcpf_newton;   }
  Tensor<1,dim>          get_grad_dcpf_newton () const { return
     grad_dcpf_newton; }
  double                 get_beta_newton     () const { return
     beta_newton;   }

  double get_dcpf_n() const { return dcpf_n; }

private:

  const double eta;
  const double lambda;
  const double mu;
  const double gc;
  const double l;
  const double k;

  SymmetricTensor<2, dim> epsilon_newton;
  double                  dcpf_newton;
  Tensor<1,dim>           grad_dcpf_newton;
  double                  beta_newton;

  double                  dcpf_n;

  const ElasticityTensor<dim> CC;

};

template <int dim>
class PointHistory
{
public:
  PointHistory()
  :
  material(nullptr)
  {}

  virtual ~PointHistory()
  {
     delete material;
     material = nullptr;
}

void setup_quadrature_point(const Parameters::AllParameters
    &parameters)
{
   material = new Materialcrackprop<dim> ( parameters.eta,
                              parameters.lambda,
                              parameters.mu,
                              parameters.gc,
                              parameters.l,
                              parameters.k);

   const SymmetricTensor<2,dim> epsilon_start;
   const double                dcpf_start = 0.0;
   const Tensor<1,dim>         grad_dcpf_start;
   const double                beta_start = 0.0;

   update_values_newton (epsilon_start, dcpf_start, grad_dcpf_start,
       beta_start);
   update_values_timestep (dcpf_start);
}

void update_values_newton(const SymmetricTensor<2,dim> epsilon_newton,
                          const double dcpf_newton,
                          const Tensor<1,dim> grad_dcpf_newton,
                          const double beta_newton)
{
   material->update_material_data_newton(epsilon_newton,
                                         dcpf_newton,
                                         grad_dcpf_newton,
                                         beta_newton);
}

void update_values_timestep(const double dcpf_n)
{
   material->update_material_data_timestep(dcpf_n);
}

SymmetricTensor<2,dim> get_epsilon_newton() const { return
    material->get_epsilon_newton(); }
double                 get_dcpf_newton()    const { return
    material->get_dcpf_newton(); }
Tensor<1,dim>          get_grad_dcpf_newton() const { return
    material->get_grad_dcpf_newton(); }
double                 get_beta_newton()   const { return
    material->get_beta_newton();  }

double                 get_dcpf_n()         const { return
    material->get_dcpf_n(); }
   double get_material_eta    () const { return material->get_eta(); }
   double get_material_lambda  () const { return material->get_lambda();
      }
   double get_material_mu     () const { return material->get_mu(); }
   double get_material_gc     () const { return material->get_gc(); }
   double get_material_l      () const { return material->get_l(); }
   double get_material_k         () const { return material->get_k(); }

   ElasticityTensor<dim> get_CC () const { return material->get_CC(); }

private:
   Materialcrackprop<dim> *material;
};



template <int dim>
class TopLevel
{
public:
   TopLevel(const std::string &input_file, const std::string &log_file);
   ~TopLevel();

   void run();

private:
   void create_grid       ();
   void setup_system      ();
   void do_timestep_zero  ();
   void display_timestep_info_head();
   void do_nonlinear_timestep(BlockVector<double> &solution_delta);
   void update_qph_timestep();
   void update_resulting_force_top ();
   void update_resulting_force_bottom ();
   void output_results () const;


   void setup_qph           ();
   void display_newton_info_head ();
   void assemble_system     ();
   void make_constraints    (const unsigned int& iteration_number);
   void display_newton_info_body (const unsigned int& iteration_number,
                            const NewtonError residual,
                            const NewtonError update);
   void solve_linear_system (BlockVector<double> &newton_update);
   void update_qph_newton   (const BlockVector<double> &solution_delta);

   BlockVector<double> get_total_solution (const BlockVector<double>
      &solution_delta) const;
```

```
Parameters::AllParameters parameters;
std::ofstream log;
teestream tee;


Triangulation<dim>           triangulation;

std::vector<PointHistory<dim> > quadrature_point_history;

BoundaryConditionControl     bc_control;

const unsigned int           fe_degree;
const FESystem<dim>          fe;

DoFHandler<dim>              dof_handler;
const unsigned int           dofs_per_cell;

const FEValuesExtractors::Vector u_fe;
const FEValuesExtractors::Scalar dcpf_fe;
const FEValuesExtractors::Scalar beta_fe;

static const unsigned int    n_blocks        = 3;
static const unsigned int    n_components    = dim + 2;
static const unsigned int    first_u_component = 0;
static const unsigned int    dcpf_component  = dim;
static const unsigned int    beta_component   = dim + 1;

enum
{
   u_dof    = 0,
   dcpf_dof = 1,
   beta_dof = 2
};

std::vector<types::global_dof_index> dofs_per_block;

const QGauss<dim> qf_cell;
const QGauss<dim-1> qf_face;

const unsigned int n_q_points;
const unsigned int n_q_points_face;

ConstraintMatrix        constraints;
BlockSparsityPattern    sparsity_pattern;
BlockSparseMatrix<double> tangent_matrix;
BlockVector<double>     system_rhs;
BlockVector<double>     solution_n;

NewtonError error_residual, error_residual_0, error_residual_norm;
NewtonError error_update, error_update_0, error_update_norm;
```

```
   void get_error_residual(NewtonError &residual);
   void get_error_update (const BlockVector<double> &newton_update,
                              NewtonError    &error_update);

   std::vector<double> resulting_displacement_top;
   std::vector<double> resulting_force_top;
   std::vector<double> resulting_force_bottom;

   class Postprocessor;
};



template <int dim>
TopLevel<dim>::TopLevel(const std::string &input_file, const
    std::string &log_file)
:
parameters(input_file),
log(log_file),
tee(std::cout, log),

bc_control(parameters.displacement_start,
          parameters.displacement_end,
          parameters.displacement_stepsize,
          parameters.velocity,
          parameters.time_start),

fe_degree(parameters.poly_degree),
fe(FE_Q<dim>(parameters.poly_degree), dim, //displacement
   FE_Q<dim>(parameters.poly_degree), 1, //dcpf
   FE_Q<dim>(parameters.poly_degree), 1), //beta

dof_handler  (triangulation),
dofs_per_cell (fe.dofs_per_cell),

u_fe  (first_u_component),
dcpf_fe(dcpf_component),
beta_fe(beta_component),

dofs_per_block(n_blocks),

qf_cell(parameters.quad_order),
qf_face(parameters.quad_order),

n_q_points    (qf_cell.size()),
n_q_points_face(qf_face.size())
{

}
```

```
template <int dim>
TopLevel<dim>::~TopLevel()
{
   dof_handler.clear();
}



template <int dim>
void TopLevel<dim>::run()
{
   create_grid ();
   setup_system ();
   output_results();

   display_timestep_info_head();

   BlockVector<double> solution_delta(dofs_per_block);
   while (bc_control.get_current_displacement() <
       bc_control.get_final_displacement())
   {
      bc_control.increment();
      display_timestep_info_head();

      solution_delta = 0.0;
      do_nonlinear_timestep(solution_delta);
      solution_n += solution_delta;

      update_qph_timestep();

      update_resulting_force_top();
      update_resulting_force_bottom();
      output_results();


      libfest::make_2D_matlab_plot_file(resulting_displacement_top,
                                        resulting_force_top,
                                        "output/kraft_weg_top.m");
      libfest::make_2D_matlab_plot_file(resulting_displacement_top,
                                        resulting_force_bottom,
                                        "output/kraft_weg_bottom.m");
   }

   libfest::make_2D_matlab_plot_file(resulting_displacement_top,
                                     resulting_force_top,
                                     "output/kraft_weg_top.m");
   libfest::make_2D_matlab_plot_file(resulting_displacement_top,
                                resulting_force_bottom,
                                "output/kraft_weg_bottom.m");
}



template <int dim>
void TopLevel<dim>::create_grid()
{
   const double left = -parameters.sidelength/2;
   const double right = parameters.sidelength/2;
   const unsigned int n_elements_h =
      parameters.subdivisions_horizontally;
   const unsigned int n_elements_v = parameters.subdivisions_vertically;
   const double element_length =
      parameters.sidelength/parameters.subdivisions_vertically;

  Point<dim> p1,p2;
  Point<dim> global_center;
  std::vector<unsigned int> repetitions = {n_elements_v,n_elements_h};


   if(dim==2)
   {
       p1           = {left,0.0};
       p2           = {right,right};
       global_center = {0.0, 0.0};
   }
   else if(dim==3)
   {
      p1           = {left , 0 ,-element_length/2};
      p2           = {right,right, element_length/2};
      global_center = {0.0, 0.0, 0.0};

      repetitions.push_back(1);
  }

  GridGenerator::subdivided_hyper_rectangle(triangulation, repetitions,
      p1, p2);

  for (typename Triangulation<dim>::active_cell_iterator
  cell=triangulation.begin_active();
  cell!=triangulation.end(); ++cell)
  {
    for (unsigned int f=0; f<GeometryInfo<dim>::faces_per_cell; ++f)
       if (cell->face(f)->at_boundary())
       {

         if (cell->face(f)->center()[0] >= 0.0
            &&
              cell->face(f)->center()[1] == 0.0)
                cell->face(f)->set_boundary_id(1); // right, bottom edge
            if (
              cell->face(f)->center()[0] < 0.0
              &&
              cell->face(f)->center()[1] == 0.0)
                cell->face(f)->set_boundary_id(2); // left, bottom edge
////           if (cell->face(f)->center()[1] == 0.0)
////               cell->face(f)->set_boundary_id(7); // bottom edge
            if (cell->face(f)->center()[0] == right)
              cell->face(f)->set_boundary_id(5); // right edge
            if (cell->face(f)->center()[0] == left)
              cell->face(f)->set_boundary_id(6); // left edge
            if (cell->face(f)->center()[1] == right)
              cell->face(f)->set_boundary_id(3); // top edge
            if (dim==3)
               if (cell->face(f)->center()[2] == -element_length/2 ||
                cell->face(f)->center()[2] == element_length/2)
                cell->face(f)->set_boundary_id(4); // in the 3-D case, top
                    and bottom surface


          }
    }


    std::ofstream out ("grid_initial.eps");
    GridOut grid_out;
    grid_out.write_eps (triangulation, out);
  }



  template <int dim>
  void TopLevel<dim>::setup_system()
  {
     std::vector<unsigned int> block_component(n_components, u_dof); //
         displacement
     block_component[dcpf_component] = dcpf_dof;               // dcpf
     block_component[beta_component] = beta_dof;               // beta

     dof_handler.distribute_dofs(fe);

     DoFRenumbering::Cuthill_McKee(dof_handler);
     DoFRenumbering::component_wise(dof_handler, block_component);
     DoFTools::count_dofs_per_block(dof_handler, dofs_per_block,
                                block_component);


     tee << "Triangulation:"
```

```
           << "\n\t Number of Subdivisions in the horizontal
               direction:\t" << parameters.subdivisions_horizontally
          << "\n\t Number of Subdivisions in the vertical direction:\t"
              << parameters.subdivisions_vertically
              << "\n\t Number of active cells:\t" <<
                  triangulation.n_active_cells()
              << "\n\t Number of degrees of freedom:\t" <<
                  dof_handler.n_dofs()
              << std::endl;

tangent_matrix.clear();
{
   const types::global_dof_index n_dofs_u = dofs_per_block[u_dof];
   const types::global_dof_index n_dofs_dcpf =
      dofs_per_block[dcpf_dof];
   const types::global_dof_index n_dofs_beta =
      dofs_per_block[beta_dof];

   BlockDynamicSparsityPattern dsp(n_blocks, n_blocks);

   dsp.block(u_dof, u_dof) .reinit(n_dofs_u, n_dofs_u);
   dsp.block(u_dof, dcpf_dof).reinit(n_dofs_u, n_dofs_dcpf);
   dsp.block(u_dof, beta_dof).reinit(n_dofs_u, n_dofs_beta);

   dsp.block(dcpf_dof, u_dof) .reinit(n_dofs_dcpf, n_dofs_u);
   dsp.block(dcpf_dof, dcpf_dof).reinit(n_dofs_dcpf, n_dofs_dcpf);
   dsp.block(dcpf_dof, beta_dof).reinit(n_dofs_dcpf, n_dofs_beta);

   dsp.block(beta_dof, u_dof) .reinit(n_dofs_beta, n_dofs_u);
   dsp.block(beta_dof, dcpf_dof).reinit(n_dofs_beta, n_dofs_dcpf);
   dsp.block(beta_dof, beta_dof).reinit(n_dofs_beta, n_dofs_beta);
   dsp.collect_sizes();

   Table<2, DoFTools::Coupling> coupling(n_components, n_components);
   for (unsigned int ii = 0; ii < n_components; ++ii)
      for (unsigned int jj = 0; jj < n_components; ++jj)
         if (((ii < dcpf_component) && (jj == beta_component)) ||
             ((ii == beta_component) && (jj < dcpf_component)))
            coupling[ii][jj] = DoFTools::none;
         else
            coupling[ii][jj] = DoFTools::always;

   DoFTools::make_sparsity_pattern(dof_handler,
                                   coupling,
                                   dsp,
                                   constraints,
                                   false);

   sparsity_pattern.copy_from(dsp);
```

```
      std::ofstream out ("sparsity_pattern.gnu");
      sparsity_pattern.print_gnuplot (out);
   }
   tangent_matrix.reinit(sparsity_pattern);

   system_rhs.reinit(dofs_per_block);
   system_rhs.collect_sizes();

   solution_n.reinit(dofs_per_block);
   solution_n.collect_sizes();

   setup_qph();

   resulting_displacement_top.clear();
   resulting_force_top. clear();
   resulting_force_bottom. clear();
}



template <int dim>
void TopLevel<dim>::do_timestep_zero()
{
  const double initial_displacement =
      bc_control.get_start_displacement();
  const double initial_force     = 0.0;

   resulting_displacement_top.at(0) = initial_displacement;
   resulting_force_top   .at(0) = initial_force;
   resulting_force_bottom  .at(0) = initial_force;
}



template <int dim>
void TopLevel<dim>::display_timestep_info_head()
{
   const unsigned int total_width = 103;
   const double progress =
       bc_control.get_current_displacement()/bc_control
   .get_final_displacement()*100;

   tee << "\n\n\n Timestep Number " <<
       bc_control.get_current_timestep_number() <<
               "/" << bc_control.get_total_timestep_number() << " | ";

   std::streamsize ss = tee.precision();
   tee << std::fixed << std::setprecision(2);

   tee << progress << "%" << std::endl;
   tee << std::setprecision(ss);
   tee.unsetf(std::ios::fixed);

   tee << std::setfill('-') << std::setw(total_width) << "-" <<
       std::endl;
   tee << std::setfill(' ');

   tee << " Current Time:   " << bc_control.get_current_time() << "s" <<
       std::endl;
   tee << " Top Displacement: " << bc_control.get_current_displacement()
       << "mm" << std::endl;
}



template <int dim>
void TopLevel<dim>::do_nonlinear_timestep(BlockVector<double>
    &solution_delta)
{
   BlockVector<double> newton_update (dofs_per_block);

   error_residual.reset();
   error_residual_0.reset();
   error_residual_norm.reset();
   error_update.reset();
   error_update_0.reset();
   error_update_norm.reset();

   display_newton_info_head ();

   unsigned int newton_iteration = 0;
   for (; newton_iteration <= parameters.max_newton_iterations;
       ++newton_iteration)
   {
      tangent_matrix = 0.0;
      system_rhs     = 0.0;

      assemble_system();
      make_constraints(newton_iteration);
      constraints.condense(tangent_matrix, system_rhs);

      solve_linear_system(newton_update);

      get_error_residual(error_residual);
      if (newton_iteration == 0)
         error_residual_0 = error_residual;

      error_residual_norm = error_residual;
      error_residual_norm.normalise(error_residual_0);
      get_error_update(newton_update, error_update);
      if (newton_iteration == 0)
        error_update_0 = error_update;

      error_update_norm = error_update;
      error_update_norm.normalise(error_update_0);

      display_newton_info_body(newton_iteration, error_residual_norm,
          error_update_norm);

      if (newton_iteration > 0 && error_update_norm.norm <=
          parameters.tol_update
          && error_residual_norm.norm <= parameters.tol_residual)
      {
         tee << " Newton solver converged." << std::endl;
         break;
      }

      solution_delta += newton_update;

      update_qph_newton(solution_delta);
   }

   AssertThrow (newton_iteration < parameters.max_newton_iterations,
                ExcMessage("No convergence in nonlinear solver!"));
}



template <int dim>
void TopLevel<dim>::update_qph_timestep()
{
   FEValues<dim> fe_values (fe, qf_cell,
                            update_values);

    std::vector<double>       dcpfs            (n_q_points, 0.0);

    for(typename DoFHandler<dim>::active_cell_iterator
        cell = dof_handler.begin_active();
        cell != dof_handler.end(); ++cell)
    {
      PointHistory<dim> *qph =
          reinterpret_cast<PointHistory<dim>*>(cell->user_pointer());

      Assert (qph >= &quadrature_point_history.front(),
          ExcInternalError());
      Assert (qph < &quadrature_point_history.back() ,
          ExcInternalError());
      fe_values.reinit(cell);

      fe_values[dcpf_fe].get_function_values (solution_n, dcpfs);

      for (unsigned int q = 0; q < n_q_points; ++q)
        qph[q].update_values_timestep(dcpfs[q]);
    }
}



template<int dim>
class TopLevel<dim>::Postprocessor : public DataPostprocessor<dim>
{
public:
   Postprocessor () {}
   virtual ~Postprocessor() {}

   virtual void compute_derived_quantities_vector
   (const std::vector<Vector<double> >          &uh,
    const std::vector<std::vector<Tensor<1,dim> > > &duh,
    const std::vector<std::vector<Tensor<2,dim> > > &dduh,
    const std::vector<Point<dim> >               &normals,
    const std::vector<Point<dim> >               &evaluation_points,
    std::vector<Vector<double> >           &computed_quantities) const;

   virtual std::vector<std::string> get_names() const;

   virtual
   std::vector<DataComponentInterpretation::DataComponentInterpretation>
   get_data_component_interpretation () const;

   virtual UpdateFlags get_needed_update_flags () const;
};


template <int dim>
std::vector<std::string> TopLevel<dim>::Postprocessor::get_names() const
{
   std::vector<std::string> solution_names (dim, "displacement");
   solution_names.push_back ("dcpf");
   solution_names.push_back ("beta");


   return solution_names;
}

template <int dim>
std::vector<DataComponentInterpretation::DataComponentInterpretation>
TopLevel<dim>::Postprocessor::get_data_component_interpretation () const
```

```
{
   std::vector<DataComponentInterpretation::DataComponentInterpretation>
   interpretation (dim,
       DataComponentInterpretation::component_is_part_of_vector);

   interpretation.push_back
       (DataComponentInterpretation::component_is_scalar);
   interpretation.push_back
       (DataComponentInterpretation::component_is_scalar);

   return interpretation;
}

template <int dim>
UpdateFlags
TopLevel<dim>::Postprocessor::get_needed_update_flags() const
{
   return update_values | update_gradients | update_q_points;
}

template <int dim>
void
TopLevel<dim>::Postprocessor::
compute_derived_quantities_vector
(const std::vector<Vector<double> >         &uh,
 const std::vector<std::vector<Tensor<1,dim> > > &/* duh */,
 const std::vector<std::vector<Tensor<2,dim> > > &/* dduh */,
 const std::vector<Point<dim> >             &/* normals */,
 const std::vector<Point<dim> >             &/* evaluation_points */,
       std::vector<Vector<double> >         &computed_quantities) const
{
   const unsigned int n_quadrature_points = uh.size();
   Assert (computed_quantities.size() == n_quadrature_points,
       ExcInternalError());
   Assert (uh[0].size() == dim+2,
       ExcInternalError());

   for (unsigned int q=0; q<n_quadrature_points; ++q)
     {
       for (unsigned int d=0; d<dim; ++d)
         computed_quantities[q](d) = uh[q](d);

       const double dcpf = uh[q](dim);
       computed_quantities[q](dim) = dcpf;

       const double beta = uh[q](dim+1);
       computed_quantities[q](dim+1) = beta;

     }
}
```

```
   template <int dim>
   void TopLevel<dim>::output_results() const
   {
      Postprocessor postprocessor;

      DataOut<dim> data_out;

      data_out.attach_dof_handler (dof_handler);
      data_out.add_data_vector (solution_n, postprocessor);
      data_out.add_data_vector (system_rhs, "system_rhs");

      Vector<double> soln(solution_n.size());
      for (unsigned int i = 0; i < soln.size(); ++i)
         soln(i) = solution_n(i);

      MappingQEulerian<dim> q_mapping(fe_degree, dof_handler, soln);

      data_out.build_patches(q_mapping, fe_degree);

      data_out.build_patches();

      std::ostringstream filename;
      filename << "output/vtk_files/solution-" <<
          bc_control.get_current_timestep_number() << ".vtk";
      std::ofstream output(filename.str().c_str());

      data_out.write_vtk(output);

// For text output:
//      std::ostringstream filename;
//      filename << "output/text/rhs_vector-" <<
    bc_control.get_current_timestep_number() << ".txt";
//      std::ofstream output(filename.str().c_str());

      data_out.write_vtk(output);
   }


   template <int dim>
   void TopLevel<dim>::update_resulting_force_top()
   {
      if(bc_control.get_current_timestep_number() == 0)
      {
         resulting_displacement_top.push_back(bc_control
         .get_start_displacement());
         resulting_force_top. push_back(0.0);
         return;
      }
```

```
double resulting_force_t = 0;

FEFaceValues<dim> fe_face_values(fe, qf_face,
                                 update_values |update_gradients |
                                 update_quadrature_points |
                                    update_JxW_values);

std::vector<Tensor<2,dim> > displacement_grads (n_q_points_face,
   Tensor<2,dim>() );
std::vector<double>      dcpfs           (n_q_points_face, 0.0);

const ElasticityTensor<dim> ela_tensor(parameters.lambda,
   parameters.mu);
const SymmetricTensor<4,dim> CC = ela_tensor.get_entries();

typename DoFHandler<dim>::active_cell_iterator
cell = dof_handler.begin_active(),
endc = dof_handler.end();
for (; cell!=endc; ++cell)
  for(unsigned int f=0; f<GeometryInfo<dim>::faces_per_cell; ++f)
    if(cell->face(f)->at_boundary()
    &&
      cell->face(f)->boundary_id()==3)
    {
      fe_face_values.reinit(cell, f);

      fe_face_values[u_fe]. get_function_gradients(solution_n,
         displacement_grads);
      fe_face_values[dcpf_fe].get_function_values (solution_n,
         dcpfs);

      SymmetricTensor<2,dim> averaged_face_stress;

      for(unsigned int q=0; q<n_q_points_face; ++q)
      {
        const SymmetricTensor<2,dim> epsilon =
           symmetrize(displacement_grads[q]);
        const double              dcpf   = dcpfs[q];


        averaged_face_stress += (pow(1-dcpf,2) + parameters.k) *
           CC * epsilon/n_q_points_face;
      }

      double F_face_x = averaged_face_stress[1][1] *
         cell->face(f)->measure();

      resulting_force_t += F_face_x;
    }
   const double element_width    =
      parameters.sidelength/parameters.subdivisions_vertically;
   const double force_per_thickness_t = resulting_force_t/element_width;

   resulting_displacement_top
   .push_back(bc_control.get_current_displacement());
   resulting_force_top  .push_back(force_per_thickness_t);

   tee << " Resulting force at the top [kN]: " << force_per_thickness_t
      << std::endl;
}

template <int dim>
void TopLevel<dim>::update_resulting_force_bottom()
{
   if(bc_control.get_current_timestep_number() == 0)
   {
      resulting_force_bottom. push_back(0.0);
      return;
   }

   double resulting_force_b = 0;

   FEFaceValues<dim> fe_face_values(fe,qf_face,
                                 update_values |update_gradients |
                                 update_quadrature_points |
                                    update_JxW_values);

   std::vector<Tensor<2,dim> > displacement_grads (n_q_points_face,
      Tensor<2,dim>() );
   std::vector<double>      dcpfs           (n_q_points_face, 0.0);

   const ElasticityTensor<dim> ela_tensor(parameters.lambda,
      parameters.mu);
   const SymmetricTensor<4,dim> CC = ela_tensor.get_entries();

   typename DoFHandler<dim>::active_cell_iterator
   cell = dof_handler.begin_active(),
   endc = dof_handler.end();
   for (; cell!=endc; ++cell)
      for(unsigned int f=0; f<GeometryInfo<dim>::faces_per_cell; ++f)
         if(cell->face(f)->at_boundary()
            &&
         cell->face(f)->center()[1]==0)
         {
            fe_face_values.reinit(cell, f);

            fe_face_values[u_fe]. get_function_gradients(solution_n,
               displacement_grads);
            fe_face_values[dcpf_fe].get_function_values (solution_n,
                dcpfs);;

            SymmetricTensor<2,dim> averaged_face_stress;

            for(unsigned int q=0; q<n_q_points_face; ++q)
            {
               const SymmetricTensor<2,dim> epsilon =
                   symmetrize(displacement_grads[q]);
               const double                dcpf   = dcpfs[q];

               averaged_face_stress += (pow(1-dcpf,2) + parameters.k) *
                   CC * epsilon/n_q_points_face;

            }

            double F_face_x = averaged_face_stress[1][1] *
                cell->face(f)->measure();

            resulting_force_b += F_face_x;
         }

   const double element_width    =
       parameters.sidelength/parameters.subdivisions_vertically;
   const double force_per_thickness_b = resulting_force_b/element_width;

   resulting_force_bottom     .push_back(force_per_thickness_b);

   tee << " Resulting force at the bottom [kN]: " <<
       force_per_thickness_b << std::endl;
}




template <int dim>
void TopLevel<dim>::setup_qph()
{
   tee << "   Setting up quadrature point data...\n\n" << std::endl;
  {
     triangulation.clear_user_data();
     {
       std::vector<PointHistory<dim> > tmp;
       tmp.swap(quadrature_point_history);
     }

     quadrature_point_history.resize(triangulation.n_active_cells() *
         n_q_points);
     unsigned int history_index = 0;
     for(typename Triangulation<dim>::active_cell_iterator
         cell = triangulation.begin_active();
         cell != triangulation.end(); ++cell)
     {
        cell->set_user_pointer(&quadrature_point_history[history_index]);
        history_index += n_q_points;
     }
     Assert(history_index == quadrature_point_history.size(),
           ExcInternalError());
  }

   for (typename Triangulation<dim>::active_cell_iterator
      cell = triangulation.begin_active();
      cell != triangulation.end(); ++cell)
  {
     PointHistory<dim> *qp_history =
        reinterpret_cast<PointHistory<dim>*>(cell->user_pointer());

     Assert(qp_history >= &quadrature_point_history.front(),
        ExcInternalError());
     Assert(qp_history <= &quadrature_point_history.back(),
        ExcInternalError());

     for (unsigned int q_point = 0; q_point < n_q_points; ++q_point)
        qp_history[q_point].setup_quadrature_point(parameters);
    }
}



template <int dim>
void TopLevel<dim>::display_newton_info_head ()
{
   const unsigned int column_width = 16;
   tee << std::left;
   tee << "\n Newton Solver: " << std::setw(column_width) << "
      Iteration No. " << "|" <<
                                    std::setw(column_width) << " Residual
                                       Norm" << "|" <<
                                    std::setw(column_width) << " Update
                                       Norm" << "|" <<
                                    std::setw(column_width) << " Residual
                                       u" << "|" <<
                                    std::setw(column_width) << " Update u"
                                          <<
                                    std::endl;
   tee << std::setfill('-');
   tee <<  "                  " << std::setw(column_width) << "-" << "+"
      <<
```

```
                                std::setw(column_width) << "-" << "+"
                                    <<
                                std::setw(column_width) << "-" << "+"
                                    <<
                                std::setw(column_width) << "-" << "+"
                                    <<
                                std::setw(column_width) << "-" <<
                                       std::endl;
    tee << std::setfill(' ');
}



template <int dim>
void TopLevel<dim>::assemble_system()
{
   FEValues<dim> fe_values (fe, qf_cell,
                            update_values     | update_gradients |
                            update_JxW_values);

   FullMatrix<double> local_tangent(dofs_per_cell, dofs_per_cell);
   Vector   <double> local_rhs  (dofs_per_cell);

   std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);

   for(typename DoFHandler<dim>::active_cell_iterator
     cell = dof_handler.begin_active();
     cell != dof_handler.end(); ++cell)
   {
      fe_values.reinit(cell);
      local_tangent = 0.0;
      local_rhs    = 0.0;

      const PointHistory<dim> *local_quadrature_points_data
        = reinterpret_cast<PointHistory<dim>*>(cell->user_pointer());

      for(unsigned int q=0; q<n_q_points; ++q)
      {
         const SymmetricTensor<2,dim> epsilon =
            local_quadrature_points_data[q].get_epsilon_newton();
         const double               dcpf     =
            local_quadrature_points_data[q].get_dcpf_newton();
         const Tensor<1,dim>        grad_dcpf =
            local_quadrature_points_data[q].get_grad_dcpf_newton();
         const double               beta     =
            local_quadrature_points_data[q].get_beta_newton();


         const double dcpf_n =
            local_quadrature_points_data[q].get_dcpf_n();
```

```
          const double eta   =
             local_quadrature_points_data[q].get_material_eta();
          const double lambda  =
             local_quadrature_points_data[q].get_material_lambda();
          const double mu      =
             local_quadrature_points_data[q].get_material_mu();
          const double gc      =
             local_quadrature_points_data[q].get_material_gc();
          const double l     =
             local_quadrature_points_data[q].get_material_l();
          const double k     =
             local_quadrature_points_data[q].get_material_k();

          const ElasticityTensor<dim> ela_tensor =
             local_quadrature_points_data[q].get_CC();
          const SymmetricTensor<4,dim> CC     = ela_tensor.get_entries();

          const double tau = bc_control.get_time_stepsize();

//  alternative:  const SymmetricTensor<4,dim> C_eps_eps = (pow(1 - dcpf,2)
    + k) * (lambda * StandardTensors<dim>::IxI + 2 * mu *
    StandardTensors<dim>::II);
          const SymmetricTensor<4,dim> C_eps_eps = (pow(1 - dcpf,2) + k)
             * CC;

//  alternative:       const SymmetricTensor<2,dim> C_eps_dcpf = -2 * (1 -
    dcpf) * (lambda * trace(epsilon) * unit_symmetric_tensor<dim>() +
//                           2 * mu * epsilon);
          const SymmetricTensor<2,dim> C_eps_dcpf  = -2 * (1 - dcpf) * CC
             * epsilon;

          const SymmetricTensor<2,dim> C_dcpf_eps  = C_eps_dcpf;
          const double                C_dcpf_dcpf    = 2 * (0.5 * lambda *
             pow(trace(epsilon),2) +
                            mu * (pow(trace(epsilon),2) - 2 *
                               second_invariant(epsilon)));
          const double                C_dcpf_beta    = 1.0;

          const SymmetricTensor<2,dim> C_grad_dcpf_grad_dcpf = gc * l *
             unit_symmetric_tensor<dim>();

          const double                C_beta_dcpf    = C_dcpf_beta;
      const double     C_beta_beta     = -tau/eta *
         StandardFunctions::step_function(beta - gc/l * dcpf_n);

          for(unsigned int i=0; i<dofs_per_cell; ++i)
          {
             const SymmetricTensor <2,dim> phi_i_eps  =
                fe_values[u_fe].symmetric_gradient(i,q);
```

```
              const double              phi_i_dcpf    =
                 fe_values[dcpf_fe].value(i,q);
              const Tensor <1,dim>      phi_i_grad_dcpf =
                 fe_values[dcpf_fe].gradient(i,q);
              const double              phi_i_beta    = fe_values[beta_fe
                 ].value(i,q);

              for(unsigned int j=0; j<dofs_per_cell; ++j)
              {
                 const SymmetricTensor <2,dim> phi_j_eps  =
                    fe_values[u_fe].symmetric_gradient(j,q);
                 const double              phi_j_dcpf   =
                    fe_values[dcpf_fe].value(j,q);
                 const Tensor <1,dim>      phi_j_grad_dcpf =
                    fe_values[dcpf_fe].gradient(j,q);
                 const double              phi_j_beta   =
                    fe_values[beta_fe ].value(j,q);

                 local_tangent(i,j) += (phi_i_eps * C_eps_eps * phi_j_eps +
                                     phi_i_eps * C_eps_dcpf * phi_j_dcpf +

                                     phi_i_dcpf * C_dcpf_eps * phi_j_eps +
                                     phi_i_dcpf * C_dcpf_dcpf *
                                        phi_j_dcpf +
                                     phi_i_dcpf * C_dcpf_beta *
                                        phi_j_beta +

                                     phi_i_grad_dcpf *
                                        (C_grad_dcpf_grad_dcpf *
                                        phi_j_grad_dcpf) +

                                     phi_i_beta * C_beta_dcpf *
                                        phi_j_dcpf +
                                     phi_i_beta * C_beta_beta *
                                        phi_j_beta) * fe_values.JxW(q);

              }

//  alternative: const SymmetricTensor<2,dim> S_eps = (pow(1 - dcpf,2) +
    k) * (lambda * trace(epsilon) * unit_symmetric_tensor<dim>() + 2 * mu
    * epsilon);
              const SymmetricTensor<2,dim> S_eps = (pow(1 - dcpf,2) + k) *
                 CC * epsilon;
              const double              S_dcpf    = beta - 2 * (1 - dcpf) *
                 (0.5 * lambda * pow(trace(epsilon),2) +
                   mu * (pow(trace(epsilon),2) - 2 *
                      second_invariant(epsilon)));
              const Tensor<1,dim>       S_grad_dcpf = gc * l * grad_dcpf;
              const double              S_beta     = dcpf - dcpf_n
```

```
                - tau/eta * StandardFunctions::ramp_function(beta -
                  gc/l * dcpf_n);

            local_rhs(i) -= (S_eps   * phi_i_eps +
                             S_dcpf   * phi_i_dcpf +
                             S_grad_dcpf* phi_i_grad_dcpf +
                             S_beta   * phi_i_beta) * fe_values.JxW(q);

         }
      }

      cell->get_dof_indices(local_dof_indices);

      for (unsigned int i=0; i<dofs_per_cell; ++i)
      {
         system_rhs(local_dof_indices[i]) += local_rhs(i);

         for (unsigned int j=0; j<dofs_per_cell; ++j)
            tangent_matrix.add (local_dof_indices[i],
                                local_dof_indices[j],
                                local_tangent(i,j));
      }
   }
}



template <int dim>
void TopLevel<dim>::make_constraints(const unsigned int
    &iteration_number)
{
   if(iteration_number>1)
      return;
   constraints.clear();

   const bool apply_Dirichlet_bc = (iteration_number == 0);

   const FEValuesExtractors::Scalar x_displacement(0);
   const FEValuesExtractors::Scalar y_displacement(1);
   const FEValuesExtractors::Scalar z_displacement(2);

   const std::string dim_error_msg("Dimension must be 2 or 3");



   {
   const int boundary_id = 1;

   ComponentMask cmp_mask;
   if(dim==2)
```

```
 cmp_mask = (fe.component_mask(y_displacement));
else if(dim == 3)
 cmp_mask =
 (fe.component_mask(y_displacement)|fe.component_mask(z_displacement));
else
 Assert(false, ExcMessage(dim_error_msg));

if (apply_Dirichlet_bc == true)
  VectorTools::interpolate_boundary_values(dof_handler,
                                          boundary_id,
                                          ZeroFunction<dim>(n_components),
                                          constraints,
                                          cmp_mask);
else
 VectorTools::interpolate_boundary_values(dof_handler,
                                         boundary_id,
                                         ZeroFunction<dim>(n_components),
                                         constraints,
                                         cmp_mask);
}

{
const int boundary_id = 2;

const ComponentMask cmp_mask = (fe.component_mask(z_displacement));

if (apply_Dirichlet_bc == true)
 VectorTools::interpolate_boundary_values(dof_handler,
                                         boundary_id,
                                         ZeroFunction<dim>(n_components),
                                         constraints,
                                         cmp_mask);
else
 VectorTools::interpolate_boundary_values(dof_handler,
                                         boundary_id,
                                         ZeroFunction<dim>(n_components),
                                         constraints,
                                         cmp_mask);
}

{
const int boundary_id = 5;

const ComponentMask cmp_mask = (fe.component_mask(z_displacement));

if (apply_Dirichlet_bc == true)
 VectorTools::interpolate_boundary_values(dof_handler,
                                         boundary_id,
                                         ZeroFunction<dim>(n_components),
                                         constraints,
                                          cmp_mask);
else
 VectorTools::interpolate_boundary_values(dof_handler,
                                          boundary_id,
                                          ZeroFunction<dim>(n_components),
                                          constraints,
                                          cmp_mask);
}

{
const int boundary_id = 6;

const ComponentMask cmp_mask = (fe.component_mask(z_displacement));

if (apply_Dirichlet_bc == true)
 VectorTools::interpolate_boundary_values(dof_handler,
                                          boundary_id,
                                          ZeroFunction<dim>(n_components),
                                          constraints,
                                          cmp_mask);
else
 VectorTools::interpolate_boundary_values(dof_handler,
                                          boundary_id,
                                          ZeroFunction<dim>(n_components),
                                          constraints,
                                          cmp_mask);
}

{
const int boundary_id = 3;

const ComponentMask cmp_mask = fe.component_mask(y_displacement);

if (apply_Dirichlet_bc == true)
 VectorTools::interpolate_boundary_values(dof_handler,
                                          boundary_id,
                                          IncrementalBoundaryValues<dim>
                                          (bc_control.get_displacement_stepsize()),
                                          constraints,
                                          cmp_mask);
else
 VectorTools::interpolate_boundary_values(dof_handler,
                                          boundary_id,
                                          ZeroFunction<dim>(n_components),
                                          constraints,
                                          cmp_mask);
}

{
const int boundary_id = 3;
const ComponentMask cmp_mask = (fe.component_mask(z_displacement));

if (apply_Dirichlet_bc == true)
 VectorTools::interpolate_boundary_values(dof_handler,
                                          boundary_id,
                                          ZeroFunction<dim>(n_components),
                                          constraints,
                                          cmp_mask);
else
 VectorTools::interpolate_boundary_values(dof_handler,
                                          boundary_id,
                                          ZeroFunction<dim>(n_components),
                                          constraints,
                                          cmp_mask);
}



if(dim == 3)
{
const int boundary_id = 4;

const ComponentMask cmp_mask = (fe.component_mask(z_displacement));

if (apply_Dirichlet_bc == true)
 VectorTools::interpolate_boundary_values(dof_handler,
                                          boundary_id,
                                          ZeroFunction<dim>(n_components),
                                          constraints,
                                          cmp_mask);
else
 VectorTools::interpolate_boundary_values(dof_handler,
                                          boundary_id,
                                          ZeroFunction<dim>(n_components),
                                          constraints,
                                          cmp_mask);
}


const double tol_boundary = 1e-6;


typename DoFHandler<dim>::active_cell_iterator cell =
  dof_handler.begin_active(), endc = dof_handler.end();

for (; cell != endc; ++cell)
  for (unsigned int v = 0; v < GeometryInfo<dim>::vertices_per_cell;
      ++v)
if (
    (std::abs(cell->vertex(v)[2] - 0.05) < tol_boundary)
     &&
   (std::abs(cell->vertex(v)[1] - 0.0) < tol_boundary)
    &&
   (std::abs(cell->vertex(v)[0] - 0.5)< tol_boundary))
     {
  types::global_dof_index
            idx = cell->vertex_dof_index(v, 2);// 2=z displacement
            constraints.add_line(idx);
         idx = cell->vertex_dof_index(v, 1);// 1=y displacement
                        constraints.add_line(idx);
                        idx = cell->vertex_dof_index(v, 0);// 0=x
                            displacement
                        constraints.add_line(idx);
     }



{
const double tol_boundary = 1e-6;


typename DoFHandler<dim>::active_cell_iterator cell =
  dof_handler.begin_active(), endc = dof_handler.end();

for (; cell != endc; ++cell)
  for (unsigned int w = 0; w < GeometryInfo<dim>::vertices_per_cell;
      ++w)
if (
   (std::abs(cell->vertex(w)[2] + 0.05) < tol_boundary)
    &&
  (std::abs(cell->vertex(w)[1] - 0.0) < tol_boundary)
   &&
  (std::abs(cell->vertex(w)[0] - 0.5)< tol_boundary))
    {
  types::global_dof_index
            idxt = cell->vertex_dof_index(w, 2);// 2=z displacement
            constraints.add_line(idxt);
            idxt = cell->vertex_dof_index(w, 1);// 1=y displacement
                        constraints.add_line(idxt);
                        idxt = cell->vertex_dof_index(w, 0);// 0=x
                            displacement
                        constraints.add_line(idxt);
    }
}

constraints.close();
}
```

```
template <int dim>
void TopLevel<dim>::display_newton_info_body(const unsigned int&
    iteration_number,
                                        const NewtonError residual,
                                        const NewtonError update)
{
   const unsigned int column_width = 16;

   tee << "                        " << iteration_number << "    | " <<
                                    std::scientific <<
                                    std::setw(column_width-1) <<
                                       residual.norm << "| "<<
                                    std::setw(column_width-1) <<
                                       update.norm << "| "<<
                                    std::setw(column_width-1) <<
                                       residual.u << "| "<<
                                    std::setw(column_width-1) <<
                                       update.u  <<
                                    std::endl;
   tee.unsetf(std::ios::scientific);
}



template <int dim>
void TopLevel<dim>::solve_linear_system(BlockVector<double>
    &newton_update)
{
   SparseDirectUMFPACK A_direct;
   A_direct.initialize(tangent_matrix);
   A_direct.vmult(newton_update, system_rhs);


   constraints.distribute(newton_update);
}



template <int dim>
void TopLevel<dim>::update_qph_newton(const BlockVector<double>
    &solution_delta)
{
   const BlockVector<double>
      solution_total(get_total_solution(solution_delta));

   FEValues<dim> fe_values (fe, qf_cell,
                         update_values | update_gradients);
   std::vector<Tensor<2,dim> > displacement_grads(n_q_points,
       Tensor<2,dim>());
   std::vector<double>       dcpfs           (n_q_points, 0.0);
   std::vector<Tensor<1,dim> > dcpf_grads     (n_q_points,
       Tensor<1,dim>());
   std::vector<double>       betas           (n_q_points, 0.0);


   for(typename DoFHandler<dim>::active_cell_iterator
       cell = dof_handler.begin_active();
       cell != dof_handler.end(); ++cell)
   {
      PointHistory<dim> *qph =
         reinterpret_cast<PointHistory<dim>*>(cell->user_pointer());

      Assert (qph >= &quadrature_point_history.front(),
         ExcInternalError());
      Assert (qph < &quadrature_point_history.back() ,
         ExcInternalError());

      fe_values.reinit(cell);

      fe_values[u_fe]  .get_function_gradients(solution_total,
         displacement_grads);
      fe_values[dcpf_fe].get_function_values (solution_total, dcpfs);
     fe_values[dcpf_fe].get_function_gradients(solution_total,
        dcpf_grads);
    fe_values[beta_fe].get_function_values (solution_total, betas);

      for (unsigned int q = 0; q < n_q_points; ++q)
      {
         const SymmetricTensor<2,dim> epsilon =
            symmetrize(displacement_grads[q]);
         const double              dcpf     = dcpfs[q];
         const Tensor<1,dim>       grad_dcpf = dcpf_grads[q];
         const double              beta     = betas[q];

         qph[q].update_values_newton(epsilon,
                                  dcpf,
            grad_dcpf,
            beta);
      }
   }
}



template <int dim>
void TopLevel<dim>::get_error_residual(NewtonError &error_residual)
{
    BlockVector<double> error_res(dofs_per_block);
    for (unsigned int i = 0; i < dof_handler.n_dofs(); ++i)
      if (!constraints.is_constrained(i))
        error_res(i) = system_rhs(i);

    error_residual.norm = error_res.l2_norm();
    error_residual.u    = error_res.block(u_dof).l2_norm();
    error_residual.dcpf = error_res.block(dcpf_dof).l2_norm();
    error_residual.beta = error_res.block(beta_dof).l2_norm();
  }



  template <int dim>
  void TopLevel<dim>::get_error_update(const BlockVector<double>
     &newton_update,
                                                   NewtonError       &error_update)
  {
    BlockVector<double> error_ud(dofs_per_block);
    for (unsigned int i = 0; i < dof_handler.n_dofs(); ++i)
      if (!constraints.is_constrained(i))
        error_ud(i) = newton_update(i);

    error_update.norm = error_ud.l2_norm();
    error_update.u    = error_ud.block(u_dof).l2_norm();
    error_update.dcpf = error_ud.block(dcpf_dof).l2_norm();
    error_update.beta = error_ud.block(beta_dof).l2_norm();
  }



  template <int dim>
  BlockVector<double> TopLevel<dim>::get_total_solution(const
     BlockVector<double> &solution_delta) const
  {
    BlockVector<double> solution_total(solution_n);
    solution_total += solution_delta;
    return solution_total;
  }
}

int main()
{
  try
  {
    using namespace dealii;

    using namespace crackprop;
    TopLevel<3> crack_propagation("parameters.prm", "output/output.log");
    crack_propagation.run();
```

```
   }
   catch (std::exception &exc)
   {
      std::cerr << std::endl << std::endl
               << "----------------------------------------------------"
               << std::endl;
      std::cerr << "Exception on processing: " << std::endl
               << exc.what() << std::endl
               << "Aborting!" << std::endl
               << "----------------------------------------------------"
               << std::endl;
      return 1;
   }
   catch (...)
   {
      std::cerr << std::endl << std::endl
               << "----------------------------------------------------"
               << std::endl;
      std::cerr << "Unknown exception!" << std::endl
               << "Aborting!" << std::endl
               << "----------------------------------------------------"
               << std::endl;
      return 1;
   }

   return 0;
}
```

Bibliography

[1] W. Bangerth, G. Kanschat, R. Hartmann. *deal.II – a general purpose object oriented finite element library.* ACM Trans. Math. Softw. 33, Issue 4, 24/1-24/27, 2007.

[2] K.-J. Bathe. *Finite Element Procedures.* Prentice-Hall, Englewood Cliffs, 1996.

[3] T. Belytschko, B. Moran, W.K. Liu. *Nonlinear Finite Elements for Continua and Structures.* John Wiley and Sons, Chichester, first edition, 2000.

[4] J. Bonet, R.D. Wood. *Nonlinear Continuum Mechanics for Finite Element Analysis.* The Press Syndicate of the University of Cambridge, Cambridge, first edition, 1997.

[5] R. de Borst, P. Nithiarasu, T.E. Tezduyar, G. Yagawa, T. Zohdi. *Non-Linear Finite Element Analysis of Solids and Structures.* John Wiley and Sons, Chichester, second edition, 2012.

[6] G. Capriz. *Continua with latent microstructure.* Archive for Rational Mechanics and Analysis 90, Issue 1, 43-56, 1985.

[7] G. Capriz. *Continua with Microstructure.* Springer Tracts in Natural Philosophy 35. Springer-Verlag New York, 1989.

[8] G. Capriz, P.M Mariano. *Advances in Multifield Theories for Continua with Substructure.* Springer Science + Business Media, New York, 2004.

[9] B.D. Coleman, M.E.Gurtin. *Thermodynamics with internal state variables.* The Journal of Chemical Physics 47, 597-613, 1967.

[10] C. Celigoj. *Methode der Finiten Elemente.* Institute for Strength of Materials, Faculty for Mechanical Engineering, Graz University of Technology, Graz, 1998.

[11] C. Celigoj. *Festigkeitslehre.* Institute for Strength of Materials, Faculty for Mechanical Engineering, Graz University of Technology, Graz, 2004.

[12] D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B.Turcksin, D. Wells, D. Arndt, W. Bangerth. *The deal.II library, version 8.4.1.* Journal of Numerical Mathematics 25, Issue 3, 137-146, 2017.

[13] Y.I. Dimitrienko *Nonlinear Continuum Mechanics and Large Inelastic Deformations.* Springer Netherlands, Berlin, 2011.

[14] B.J. Dimitrijevic, K. Hackl. *A method for gradient enhancement of continuum damage models.* Technische Mechanik 28, Issue 1, 43-52, 2008.

[15] A.A. Griffith. *The phenomena of rupture and flow in solids.* Philosophical Transactions of the Royal Society of London. Series A, Containing Papers of a Math. or Phys. Character 221, 163–198, 1921.

[16] D. Gross, T. Seelig. *Bruchmechanik: Mit einer Einführung in die Mikromechanik.* Springer Vieweg, sixth edition, 2016.

[17] M. Hammer. *The Finite Element Method, Linear and Non-Linear Structural Analysis.* Institute for Strength of Materials, Faculty for Mechanical Engineering, Graz University of Technology, Graz, 2013.

[18] P. Haupt. *Continuum Mechanics and Theory of Materials.* Springer-Verlag Berlin Heidelberg, Berlin, second edition, 2002.

[19] M. Hofacker, C. Miehe , F.R. Welschinger. *A variational–based formulation of regularized brittle fracture.* Proceedings in Applied Mathematics and Mechanics 9, 207-208, 2009.

[20] H.K. Iben. *Tensorrechnung.* B. G. Teubner, Stuttgart, second edition, 1999.

[21] G.A. Maugin. *The method of virtual power in continuum mechanics: Application to coupled fields.* Acta Mechanica 35, 1-70, 1980.

[22] G.A. Maugin, A. Morro. *Viscoelastic materials with internal variables and dissipation functions.* Acta Physica Hungarica 66, Issue 1-4, 69-78, 1989.

[23] G.A. Maugin; A. Morro. *Constitutive equations in viscoelasticity through the Legendre-Fenchel transformation.* Rheologica Acta 28, Issue 2, 190-192, 1989.

[24] G.A. Maugin. *Internal variables and dissipative structures.* Journal of Non-Equilibrium Thermodynamics 15, 173-192, 1990.

[25] G.A. Maugin, H. Altenbach and V. Erofeev (Editors). *Mechanics of Generalized Continua.* Springer-Verlag Berlin Heidelberg, Heidelberg, 2011.

[26] C. Miehe, F.R. Welschinger, M. Hofacker. *A phase field model for rate-independent crack propagation: Robust algorithmic implementation based on operator splits.* Computer Methods in Applied Mechanics and Engineering, 199, 2765–2778, 2010.

[27] C. Miehe, M. Hofacker, F. Welschinger. *A phase field model of electromechanical fracture.* Journal of the Mechanics and Physics of Solids 58, Issue 10, 1716-1740, 2010.

[28] C. Miehe, M. Hofacker, F. Welschinger. *Thermodynamically consistent phase-field models of fracture: Variational principles and multi-field FE implementations.* Journal for Numerical Methods in Engineering 83, 1273-1311, 2010.

[29] C. Miehe. *A multi-field incremental variational framework for gradient-extended standard dissipative solids.* Journal of the Mechanics and Physics of Solids 59, Issue 4, 898–923, 2011.

[30] C. Miehe. *Mixed variational principles for the evolution problem of gradient–extended dissipative solids.* GAMM-Mitteilungen, 35, Issue 1, 8-25, 2012.

[31] A. Mielke, T. Roubiček. *Rate-independent damage processes in nonlinear elasticity.* Mathematical Models and Methods in Applied Sciences, 16, Issue 2, 1–33, 2005.

[32] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P Flannery. *Numerical Recipes, The Art of Scientific Computing.* Cambridge University Press, Cambridge, third edition, 2007.

[33] C. Truesdell, W. Noll. *The non-linear field theories of mechanics.* Encyclopedia of Physics, Voume III/3, 1965.

[34] T. Waffenschmidt, S. Blanco, A. Menzel, C. Polindara. *A gradient-enhanced large-deformation continuum damage model for fibre-reinforced materials.* Computer Methods in Applied Mechanics and Engineering 268, 801-842, 2014.

[35] F.R. Welschinger, C. Miehe, D. Zimmermann. *A variational formulation of nonlocal materials with microstructure based on dual macro- and micro-balances.* Proceedings in Applied Mathematics and Mechanics 7, Issue 1, 4080015 - 4080016, 2007.

[36] F.R. Welschinger, C. Miehe. *Variational formulations and FE active-set strategies for rate-independent nonlocal material response.* Proceedings in Applied Mathematics and Mechanics 8, 10475 - 10476, 2008.

[37] F.R. Welschinger. *A Variational Framework for Gradient-Extended Dissipative Continua. Application to Damage Mechanics, Fracture, and Plasticity.* Bericht Nr.: I-24, Institut für Mechanik (Bauwesen), Stuttgart, 2011.

[38] O.C. Zienkiewicz, R.L. Taylor. *The Finite Element Method.* McGraw-Hill Book Company, London u.a., fourth edition, 1989.