Department of Mechanical Engineering
Division of Mechanics

ISRN LUTFD2/(TFME-20)/5042-SE(1-136)

Evaluation of Peridynamics and Its Application in Harmonic-Structured Material Simulations

Master's Dissertation by

Theodor de Sousa

Supervisors:
Prof. Aylin Ahadi, Division of Mechanics
Prof. Dmytro Orlov, Division of Materials Engineering

Examiner:
Prof. Solveig Melin, Division of Mechanics

For information address:
Division of Mechanics, Lund University, Box 118, SE-221 00 Lund, Sweden
Homepage: http://www.mek.lth.se/

Abstrakt

Materialegenskaper som hållfasthet och duktilitet har historiskt förbättrats genom att materialet modifieras homogent, men även kompositer har vuxit fram som ett vanligt verktyg inom industrin som sätt att förbättra materialegenskaper. En alternativ metod som bygger på att materialets mikrostruktur modifieras heterogent, är *harmoniskt strukturerade material*. Det består av två olika moder av materialet där det ena innehar hög hållfasthet, och det andra hög duktilitet, deras topologi i materialet är dessutom kontrollerat så att ett repeterande mönster bildas. I experiment har harmoniskt strukturerade material har uppvisat gynnsamma materialegenskaper, men en djupgående förståelse för deformationsbeteendet saknas fortfarande. En del i ledet av att lära sig mer om materialbeteendet är beräkningsbaserad materialmodellering. Dock kantas traditionella simuleringsmetoder av bland annat två problem, dels att på ett resurseffektivt sätt modellera längdskalorna, och även att på ett simpelt sätt modellera brottmekanik. Genom en alternativ formulering av kontinuumsmekaniken kan båda problemen lösas, nämligen *peridynamiken*.

Det här examensarbetet, som är utfört vid avdelningen för mekanik på Lunds Tekniska Högskola, tillämpar peridynamik på harmoniskt strukturerat nickel med hjälp av programvaran *Peridigm* för att utvärdera dess potential för forskning inom harmoniskt strukturerade material, med fokus på brottmekanik. I arbetet utsätts de ingående material-moderna för enaxiella dragprover för att kalibrera en linjär elastoplastisk materialmodell mot experimentell data. Prototypsimuleringar av harmoniskt strukturerat nickel har också genomförts.

Simuleringarna lyckas inte generera kalibrerade materialmodeller, då de inte uppvisar frakturer eller midjebildning på ett tillfredställande sätt. Orsaken till avsaknaden av frakturer tillskrivs den numeriska lösningsmetoden, som antar statiska förhållanden för varje inkrementellt steg. Orsaken till att midjebildningen inte uppkommer tros bero delvis på den linjära deformationshärdningskonstanten. Prototypsimuleringen av harmoniskt strukturerat nickel uppvisar uppkomst, spridning, och reflektion av vibrationer, samt sprick-bildning och fortplantning. Beträffande Peridigm bedöms potentialen vara låg jämfört med kommersiella programvaror inom den klassiska kontinuumsmekaniken när det kommer till effektiv materialmodellering. Detta på grund av att programvaran fortfarande är i ett utvecklingsstadie, samt avsaknad av dokumentation, support, och tillgängliga verktyg. Däremot har Peridigm, i och med dess öppna källkoden och därmed den höga modifikationsnivån, stor potential för att vara ett kraftfullt verktyg för forskning på peridynamik.

Abstract

Materials are constantly sought to be improved in different manners. Two of the most common characteristics to improve are strength, and ductility. Historically, materials are modified homogeneously to achieve improvement, while the past century has also seen composite materials becoming standard-issue within various industries. Just recently the concept of heterogeneously modifying material micro-structures has gained prominence. One such concept is the *harmonic-structured material*. It consists of the same material, but topologically controlled to yield two different modes organised in a repeated manner, possessing high strength and high ductility respectively. Harmonic structured materials exhibit favourable characteristics, but the precise behaviour of the deformation of the material is yet to be understood in-depth. Computational modelling is a great asset in this behaviour evaluation, but as the length-scales of the modes and the topology are relatively large, modelling using molecular dynamics is not suitable. Neither are regular classical continuum mechanics simulations, which are incapable of modelling fracture in materials. Both of these issues are addressed by a certain formulation of continuum mechanics called *peridynamics*.

This master thesis, written at the division of mechanics at the faculty of engineering at Lund University, has applied the peridynamic theory to harmonic-structured nickel using the *Peridigm* software in order to evaluate the utility of peridynamics in the field of harmonic-structured materials, mainly focusing on fracture behaviour. In the thesis, the modal component materials are subjected to uniaxial tension in order to calibrate a linear elastic-plastic material model of experimental data. A proof-of-concept simulations of harmonic-structured nickel are also performed.

The calibration simulations fail to properly model the experimental data regarding stress localisation and fracture. The cause for absence of fracture is attributed to the use of a numerical solver which disregards velocities, and the cause for the lack of stress localisation is believed to originate partly from the linear behaviour of the strain-hardening modulus. The proof-of-concept simulation of harmonic-structured nickel is capable of modelling both wave initiation, propagation, and reflection, as well as fracture initiation and propagation. Peridigm, as a tool for material modelling, is found to be subpar regarding efficiency when compared to commercial classical continuum mechanics software. This is mainly due to its development status, lack of documentation, support, and implemented utilities. However, Peridigm is an open-source initiative and therefore could be customised extensively. Overall, it is most certainly a useful tool for research on peridynamics.

Preface and Acknowledgements

This degree project was authored in cooperation with the Division of Mechanics and Division of Materials Engineering at Lund University.

It was supervised by Professor Aylin Ahadi at the Division of mechanics, co-supervised by Professor Dmytro Orlov at the Division of materials engineering, and examined by Professor Solveig Melin at the Division of mechanics.

I first and foremost wish to express my gratitude to Aylin and Orlov for assisting me in various ways during this project, as well as Solveig for taking on the role as my examiner.

I would also like to direct several thanks to the teachers spanning from elementary school to upper secondary school and university. This project is the culmination of their effort to guide me through my youth, and they will always have my gratitude.

A large thanks to Elin Önnestam, Filip Sjövall, Karlo Kasljevic, Lea Christierson, Minh Tang, and Rania Torabi Aysf for carrying me through the magic world of computational mechanics.

For the endless discussions, as well as enduring my company for several years, I want to express a special thank-you to Martin Montelius and Robin Emanuelsson, my flatmates at Skytte C.

Finally, this report is dedicated to my parents Rose and Frederik de Sousa, for their constant encouragement, and for being the rock where I can rest on my stride through the universe. I am looking forward to doing the same for them.

List of Abbreviations

CCM	Classical Continuum Mechanics
CG	Coarse-Grained
HS	Harmonic-Structured
MD	Molecular Dynamics
NS	Node Set
PD	Peridynamics
UFG	Ultra Fine-Grained
UTS	Ultimate Tensile Strength

Contents

1 Introduction

This report applies the continuum mechanics theory of peridynamics on a novel type of materials called harmonic-structured materials. Simulations of uniaxial tension tests, on the material components as well as on a proof-of-concept harmonic-structured domain are conducted in the peridynamic software Peridigm. The topological and macro-scale stress-strain behaviour are then used to analyse the utility of peridynamics in the field of harmonic-structured materials.

1.1 Background

As society endeavours into new frontiers of possibility, the material industry and scientific community maintain pace by finding ways to improve the ductility, strength, deformation behaviour, weight, micro-structures, etc. of materials. Historically, improvements have been sought in both homogeneous materials and composite materials [1], but relatively recently a paradigm shift has occurred, resulting in an increase of research into materials with heterogeneous micro-structures. Strength is one of two key characteristics of materials [2]. It can be increased in a multitude of ways, but is often correlated with a decrease of ductility [3], the other key material characteristic. One of the methods for increasing strength is a decrease in grain size of the material, from *coarse-grained* (CG, $d_{CG} \geq 10\,\mu\mathrm{m}$) [4] down to *ultra-fine-grained* (UFG, $d_{UFG} \leq 1.0\,\mu\mathrm{m}$) [4] microstructures, but this method also decreases ductility [2].

A possible solution proposed is to modify the structure heterogeneously in a certain topology, creating what is denoted as a *Harmonic-structured Material* (HS) [4, 5, 6, 1]; The harmonic structure consists of evenly distributed cores of CG material inside a shell of UFG material, as seen in Figure 2 on page 5. The grain sizes should differ at least by one order of magnitude, and the elastic behaviour and chemical composition must be the same for both UFG and CG materials. Given that the material is relatively new, and the fabrication process involves five separate stages, some requiring specific and expensive hardware, finite element simulations have previously been utilised to better understand the deformation behaviour of HS materials [7, 8, 9].

There are several techniques for computational mechanics existing today. One such technique is the finite element method mentioned in the earlier section, which is a local classical continuum mechanics (CCM) theory [10]. In addition, there are atomistic methods for treating molecular dynamics (MD) [11]. Each of these approaches have their respective benefits and drawbacks. Within CCM [10], the material is assumed to be continuous on every scale, meaning that there is no discrete smallest form (compared to atoms, for example), instead, the material is divisible to infinity. This divisibility is represented by continuum particles possessing the micro-scale characteristics of the material. For these characteristics to remain valid, there must exist a length l which is significantly larger than the discrete phase of the material, yet small enough to omit large differences in a material on the macro-scale. This length is problem dependent, and care must be taken to ensure that l is neither too small nor too large. If the problem possesses length-scales in this interval, continuum mechanics provides a resource-efficient approximation of reality suitable for computational mechanics. One of the reasons for being resource-efficient is the fact that it is a *local* theory. A local theory of mechanics regards only immediate neighbours to the point in question when calculating properties, reducing the amount of point-wise calculations. The local theories neglect the effects of for example wave dispersion and deformation history of the complete body [12]. A local theory also fails to model the long-wavelength impact of short-wavelength cohesive forces present on atomic scales [13], like Van der Waals bonds [3].

Regarding molecular dynamics, the usefulness of length-scales is in practice limited. This is partly due to a significant increase in computational resources required as the domain size grows, but also due to a demand growth stemming from large horizons resulting in a cubic increase of computations per node. The aforementioned issues in CCM and molecular dynamics have been addressed by the creation of *Peridynamics* (PD), which was developed in 2000 [14]. It is a non-local continuum mechanics theory based on an integral formulation of the equations of motion, being valid even at discontinuities in the domain.

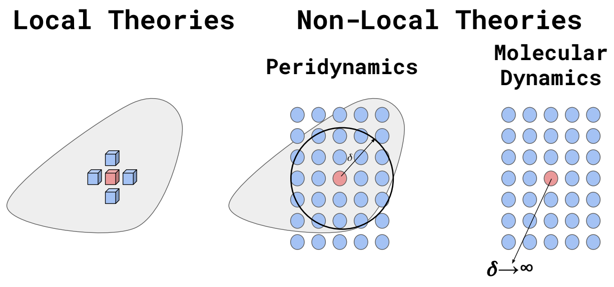


Figure 1: An illustration of the interaction between elements/nodes in different locality theories.

The peridynamic domain, being a continuum theory, consists of an infinite amount of material points/particles $\mathbf{x}_{(k)}$, which are individually identified using their reference/initial coordinates $\mathbf{x}_{(k)}$. The material points possess a volume $V_{(k)}$, and a mass density $\rho(\mathbf{x}_{(k)})$, as well as a strain energy density $W_{(k)}$. The points can be prescribed or subjected to body loads $\mathbf{b}_{(k)}(\mathbf{x}_{(k)}, t)$, displacements $\mathbf{u}_{(k)}(\mathbf{x}_{(k)}, t)$, or velocities $\dot{\mathbf{u}}_{(k)}(\mathbf{x}_{(k)}, t)$. The specific material points to be included in the calculations are determined by the horizon δ, and it is assumed that influence beyond this region is negligible [14]. The material points within the horizon are said to belong to the *family* $H_{\mathbf{x}_{(k)}}$ of $\mathbf{x}_{(k)}$and contribute to the net force acting on $\mathbf{x}_{(k)}$.

Three different peridynamic theories exists. The so-called *bond-based* peridynamics is the first of the three. This theory describes how a material point interact bond-wise with the other material points within the horizon δ. When calculating bond-wise interaction, the influence of the other material points within $H_{\mathbf{x}_{(k)}}$ on $\mathbf{x}_{(j)}$ are neglected when calculating the effect of $\mathbf{y}_{(k)}$on $\mathbf{x}_{(k)}$. Later, the concepts of state-based peridynamics emerged, due to issues with the bond-based approach [15]. They resulted in the peridynamic theory becoming more compatible with the classical continuum theory. The two state-based formulations are called *ordinary* and *non-ordinary* state based peridynamics. The ordinary state-based formulation allows peridynamic equivalents of CCM formulation, and the non-ordinary state-based formulation directly implements classical continuum descriptions of material behaviour.

1.2 Aim and Purpose

This master thesis aims to investigate the suitability of using peridynamics for fracture simulations on HS materials, and developing a workflow for using the Peridigm software, including pre- and post-processing, as well as an evaluation of the current status of Peridigm.

More specifically this thesis will document the installation and use of Peridigm. Also, methods for topology domain creation as well as data extraction and post-processing functionality are ascertained. An assessment of the Peridigm capabilities for fracture simulations in general, and HS materials specifically, is a key purpose of this thesis.

1.3 Disposition of Report

The report will first explain the theory behind HS materials and the basics of peridynamics, as well as the related Peridigm implementations. This is followed by a run-down on the methodology, which presents the experimental procedures of the thesis, the software used, domains and behaviour models used in the simulations, as well as post-processing of the results. After the methodology has been presented, the results are then presented in tables which contain the relevant settings for the simulations, a short comment, and reference to the corresponding figures. The bulk of the analysis is found in the discussion, where the simulation results of the material behaviour of necking and fracture are dissected, reasoned and related back to the theory as results of the damage and material models, as well as the numerical solvers. It also contains a discussion on the Peridigm simulation software. The most important remarks are summarised in the conclusion together with a compilation of the suggestions on further work.

2 Theory

This section is structured as an initial overview of the current understanding of harmonic-structured materials, followed by relevant peridynamic theory, and finally the corresponding Peridigm-specific implementations and workflows. The method section will more closely deal with some Peridigm-related topics more closely.

2.1 Harmonic-Structured Materials

Strength is one of two key characteristics of materials [2]. It can be increased in a multitude of ways, but is often correlated with a decrease of ductility [3], the other key characteristic of materials. One of the methods for increasing strength is a decrease in grain size of the material, from *coarse-grained* (CG, $d_{CG} \geq 10\,\mu\text{m}$) [4] down to *ultra-fine-grained* (UFG, $d_{UFG} \geq 1.0\,\mu\text{m}$) [4] microstructures, but this method does not increase ductility [2]. A proposed solution is to modify the structure heterogeneously, by featuring both CG and UFG microstructures in a *bimodal* fashion [16]. The increased ductility will originate from the coarse grains containing the lattice defects, while the UFG will provide increased strength [1].

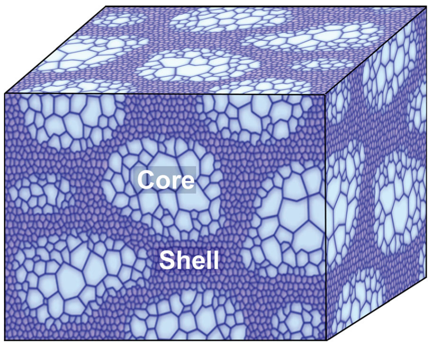


Figure 2: A conceptual illustration of the topology of the harmonic-structured material fabricated in [5]. The CG cores are surrounded by a UFG shell. Image adopted from [5].

Fabrication of such materials exists, but [1] identifies an issue in common for the methods, namely *topological distribution of coarse and fine grains*. The topology of the materials is not easily reproduced and does not take full advantage of the respective grain size characteristics. For this issue, different topology controlling methods exist, whereof one is the method being the subject of this paper: *Harmonic-Structured* (HS) materials [5]. The structure consists of regularly distributed CG cores inside a UFG shell as seen in Figure 2. Furthermore, the grain sizes differ at least by one order of magnitude, and the elastic behaviour and chemical composition are the same for both UFG and CG [1].

The fabrication process to achieve the harmonic structured material in a reproducible fashion is as follows [1]:

1. Powder creation made using plasma rotating electrode processing [17] or similar methods.
2. Deformation hardening using mechanical milling or jet milling, which decreases the grain size on the surface of the particles.
3. Combination of individual particles into full specimens by sintering using spark-plasma sintering, hot isostatic pressing, or hot-roll sintering, in a vacuum to avoid pores.
4. The previous step is repeated iteratively until the desired quality is achieved.

Current understanding of harmonic-structured metals [1] suggest that due to the heterogeneous topology, some HS material will exhibit both a ductility increase as well as an increase in toughness and strength, compared to the random-distributed bimodal material which only exhibits increased strength. Some other materials only exhibit increased toughness and strength, with the pure UFG material still having higher ductility than the harmonic structured ditto. A material which shows mutual increase in strength and ductility is harmonic-structured nickel [4], the behaviour being shown in the curves in Figure 3. Here, the stress-strain curves for randomly distributed bimodal nickel, bimodal HS nickel, and CG nickel are shown in a comparison.

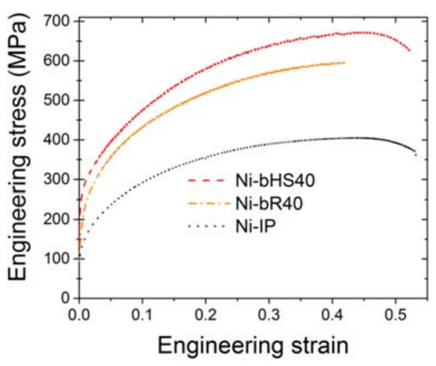


Figure 3: The stress strain curves measured in [4]. Ni-bHS40 and Ni-bR40 denotes the bimodal harmonic and random structured topology with a 40 % fraction of UFG material. The Ni-IP is the corresponding CG nickel. Image adopted from [4].

The deformation process, according to [1], consists of an initial homogeneous elastic deformation (elastic properties of the UFG and CG materials in the HS material are the same), followed by a plasticity in the CG regions. This initial plasticity affects the stress strain-curve moderately, changing the slope inclination but maintaining a nearly linear behaviour during much of the stress increase. Knowledge of the behaviour following yielding in the CG region is as of yet not established fully, but is believed to be attributed to dislocation movement [4].

2.2 Peridynamics

Being a continuum theory, the PD domain consists of an infinite amount of material points/particles. Using the notation given in [13], these particles are individually identified using their reference/initial coordinates $\mathbf{x}_{(k)}$. This is also known as a *Lagrangian* description, as opposed to the *Eulerian* description, in which references to set spatial points are used for calculations [10]. The material points possess a volume $V_{(k)}$, a mass density $\rho(\mathbf{x}_{(k)})$, and a strain energy density $W_{(k)}$ which is the sum of micro-potentials $w_{(k)(j)}$ between two material points. The points can be prescribed or subjected to body loads $\mathbf{b}_{(k)}(\mathbf{x}_{(k)}, t)$, displacements $\mathbf{u}_{(k)}(\mathbf{x}_{(k)}, t)$, or velocities $\dot{\mathbf{u}}_{(k)}(\mathbf{x}_{(k)}, t)$. Which material points to include in the calculations are defined by the horizon δ, and assumes that influence beyond this region is negligible [14]. The material points within the horizon are said to belong to the *family* $H_{\mathbf{x}_{(k)}}$ of $\mathbf{x}_{(k)}$and contribute to the net force acting on $\mathbf{x}_{(k)}$. The size of the horizon acts as a scale parameter, as well as allowing various long-range forces to be modelled. When modelling continuous and homogeneous materials for the purpose of damage and fracture, choosing the horizon as three times the grid spacing $\delta = 3\Delta x$ is found to be optimal [18].

Bond-based peridynamics, originally introduced in 2000 [14], describes bond-wise interaction between material points within the horizon δ. When calculating bond-wise interaction of $\mathbf{y}_{(k)}$on $\mathbf{x}_{(k)}$, the influence of the other material points within $H_{\mathbf{x}_{(k)}}$ on $\mathbf{x}_{(j)}$ are neglected. Several issues with the bond-based approach exist, prompting the creation of a new PD formulation known as state-based peridynamics [15]. These issues are:

- The simplification resulting from the assumption that material points within the family only interact with the central particle and not each other, resulting in, amongst other things, a limitation on the Poisson's ratio to 0.25.

- An inability to formulate a stress tensor, making it difficult to use existing continuum constitutive models, reducing the de facto usability of peridynamics.
- Inappropriate plasticity modelling, a main issue being failure to model plastic incompressibility in metals.

State-based peridynamics redefines the constitutive aspect of peridynamics, making it easier to apply and analyse CCM concepts such as elastic-plastic stress, and the Cauchy and Piola-Kirchhoff stress tensors [13]. The theory behind the peridynamic formulations is further expanded in the sections below.

2.2.1 Vector States

A state [15, 13] is generated by considering a continuous (or discontinuous) function $\underline{g}$ using an infinite amount of discrete values. The array made from these values is the state of the attribute

$$\underline{g} = \begin{bmatrix} g(x_1) \\ g(x_2) \\ g(x_3) \\ \vdots \\ g(x_i) \end{bmatrix} \tag{1}$$

where $i = 1, \ldots, \infty$.

States can be related to other states using tensors, explained in detail in [10], such as by *expansion* of a vector state $\underline{\mathbf{X}}$ to $\underline{\mathbf{Y}}$ by operating the tensor $\mathbf{A}$ on state $\underline{\mathbf{X}}$:

$$\underline{\mathbf{Y}} = \mathbf{A}\underline{\mathbf{X}} \tag{2}$$

The *extraction* of information regarding a certain discrete value from a state is denoted by

$$\underline{\mathbf{X}}\langle \mathbf{x}_1 \rangle = \mathbf{x}_1 \tag{3}$$

$$\underline{\mathbf{Y}}\langle \mathbf{x}_1 \rangle = \mathbf{y}_1 \tag{4}$$

The purpose of the state is to provide information about an attribute of a material point and how it is related to the other material points. Another key feature of the state concept is *reduction*, which is the transformation from a vector state to a second order tensor. It is defined as

$$\mathbf{F} = \Re\{\underline{\mathbf{Y}}\} = (\underline{\mathbf{Y}} * \underline{\mathbf{X}})\mathbf{K}^{-1} \tag{5}$$

Here, $\mathbf{K} = \underline{\mathbf{X}} * \underline{\mathbf{X}}$ is defined as the shape tensor, and * is the convolution of the vector states.

2.2.2 Deformation:

The deformation of the family surrounding $\mathbf{x}_{(k)}$influences its deformation. In the same manner, $\mathbf{x}_{(j)}$ is influenced by its own family. The displacement of these are denoted as $\mathbf{u}_{(k)}$ and $\mathbf{u}_{(j)}$. Introducing the relative position vector $(\mathbf{x}_{(j)} - \mathbf{x}_{(k)})$ and applying the corresponding deformations yields the relative deformed-configuration position vector

$$(\mathbf{y}_{(j)} - \mathbf{y}_{(k)}) = \big((\mathbf{x}_{(j)} + \mathbf{u}_{(j)}) - (\mathbf{x}_{(k)} + \mathbf{u}_{(k)})\big) \tag{6}$$

The stretch between $\mathbf{x}_{(k)}$and $\mathbf{x}_{(j)}$, which is a property needed for modelling damage and cracks, can now be defined as

$$s_{(k)(j)} = \frac{\big(\left|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}\right| - \left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right|\big)}{\left|\mathbf{x}_{(j)} - \mathbf{x}_{(k)}\right|} \tag{7}$$

Having yielded the deformed-configuration position vector associated with a bond, the resulting deformed-configuration position vector state is constructed from each bond within the family $H_{\mathbf{x}_{(k)}}$ giving

$$\underline{\mathbf{Y}}(\mathbf{x}_{(k)}, t) = \begin{bmatrix} (\mathbf{y}_{(1)} - \mathbf{y}_{(k)}) \\ \vdots \\ (\mathbf{y}_{(j)} - \mathbf{y}_{(k)}) \\ \vdots \\ (\mathbf{y}_{(\infty)} - \mathbf{y}_{(k)}) \end{bmatrix} \tag{8}$$

2.2.3 Force Density:

The total deformation of the material points within the family results in a force density vector $\mathbf{t}_{(k)(j)}$ which is the net force exerted on $\mathbf{x}_{(k)}$by $\mathbf{y}_{(k)}$. This force density vector is not necessarily equal to the net force exerted on $\mathbf{y}_{(k)}$by $\mathbf{x}_{(k)}$, since the family $H_{\mathbf{x}_{(j)}}$ differs from $H_{\mathbf{x}_{(k)}}$. The force density vector state for $\mathbf{x}_{(k)}$can then be defined from all interactions within the family by

$$\underline{\mathbf{T}}(\mathbf{x}_{(k)}, t) = \begin{bmatrix} \mathbf{t}_{(k)(1)} \\ \vdots \\ \mathbf{t}_{(k)(j)} \\ \vdots \\ \mathbf{t}_{(k)(\infty)} \end{bmatrix} \tag{9}$$

But, as the force density depends on the displacement, the force vector state also depends on the deformation vector state as

$$\underline{\mathbf{T}}(\mathbf{x}_{(k)}, t) = \underline{\mathbf{T}}\left(\underline{\mathbf{Y}}(\mathbf{x}_{(k)}, t)\right) \tag{10}$$

2.2.4 Strain Energy Density

The strain energy density is the result of interaction between $\mathbf{x}_{(k)}$and $\mathbf{x}_{(j)}$. It is a scalar micro-potential, depending on material properties and stretches between the material particles in the family, and constitutes a function of the interaction between the deformed-configuration position vector of the point $\mathbf{y}_{(k)}$, and the deformed-configuration position vectors of the interacting material points $\mathbf{y}_{(i^k)}$ where $i = 1, 2, \dots, \infty$:

$$\begin{aligned} w_{(k)(j)} + w_{(j)(k)} = w_{(k)(j)} &\left(\mathbf{y}_{(1^k)} - \mathbf{y}_{(k)}, \dots, \mathbf{y}_{(\infty^k)} - \mathbf{y}_{(k)}\right) \dots \\ &+ w_{(j)(k)} \left(\mathbf{y}_{(1^j)} - \mathbf{y}_{(j)}, \dots, \mathbf{y}_{(\infty^j)} - \mathbf{y}_{(j)}\right) \end{aligned} \tag{11}$$

These micro-potentials form the total strain energy density $W_{(k)}$, which can be expressed using the strain energy function

$$W_{(k)} = \frac{1}{2} \sum_{j=1}^{\infty} \left(w_{(k)(j)} + w_{(j)(k)}\right) V_{(j)} \tag{12}$$

2.2.5 Balance Laws

The PD formulation, like the CCM formulation, should provide a physically sound model of reality. This is enforced by applying balance laws, which enforces conservation of energy, as well as linear and angular momentum, on the equations of motion. The equations of motion are not only governed by the balance laws, but by displacement and constitutive relations as well. In the derivation of the equations of motion, the linear momentum and energy conservation laws are met, leaving angular momentum to be defined separately. This is done differently depending on whether bond-based, ordinary state-based or non-ordinary state-based peridynamics is used.

Equations of Motion In order to describe the PD equations of motion, the principle of virtual work is applied on the body as

$$\delta \int_{t_0}^{t_1} (L)dt = \delta \int_{t_0}^{t_1} (T-U)dt = 0 \tag{13}$$

where L is defined as the Lagrangian

$$L = T - U \tag{14}$$

Here, T is the total kinetic energy of the body and U is the total potential energy of the body, given by the summation over all material points as

$$T = \sum_{i=1}^{\infty} \frac{1}{2} \rho_{(i)} \dot{\mathbf{u}}_{(i)} \cdot \dot{\mathbf{u}}_{(i)} V_{(i)} \tag{15}$$

and

$$U = \sum_{i=1}^{\infty} W_{(i)} V_{(i)} - \sum_{i=1}^{\infty} (\mathbf{b}_{(i)} \cdot \mathbf{u}_{(i)}) V_{(i)} \tag{16}$$

respectively. Alternatively, the principle of virtual work can be formulated in words as [19]

The virtual work done by external active forces on an ideal mechanical system in equilibrium is zero for any and all virtual displacements consistent with the constraints.

The principle of virtual work is satisfied by the solution to Lagrange's equation

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\mathbf{u}}_{(k)}}\right) - \frac{\partial L}{\partial \mathbf{u}_{(k)}} = 0 \tag{17}$$

By expanding (16) using (12) and (11), inserting (16) and (15) into Lagrange's equation, and only performing calculations related to $\mathbf{x}_{(k)}$, (14) is rewritten as

$$\rho_{(k)}\ddot{\mathbf{u}}_{(k)}V_{(k)} + \left[\sum_{j=1}^{\infty}\left(\sum_{i=1}^{\infty}\frac{\partial w_{(k)(j)}}{\partial(\mathbf{y}_{(j)}-\mathbf{y}_{(k)})}V_{(i)}\right)\frac{\partial(\mathbf{y}_{(j)}-\mathbf{y}_{(k)})}{\partial\mathbf{u}_{(k)}}\right.$$
$$\left. + \sum_{j=1}^{\infty}\left(\sum_{i=1}^{\infty}\frac{\partial w_{(j)(k)}}{\partial(\mathbf{y}_{(k)}-\mathbf{y}_{(j)})}V_{(i)}\right)\frac{\partial(\mathbf{y}_{(k)}-\mathbf{y}_{(j)})}{\partial\mathbf{u}_{(k)}} - \mathbf{b}_{(k)}\right]V_{(k)} = 0 \quad (18)$$

As only the kinetic energy depends on the velocity, and both terms in the potential energy depends only on displacement, the first term is equivalent to $\frac{d}{dt}\left(\frac{\partial T}{\partial\dot{\mathbf{u}}_{(k)}}\right)$ and the bracketed term is equivalent to $\frac{\partial U}{\partial\mathbf{u}_{(k)}}$.

Knowing the dimensions of the strain energy density and the force density as

$$w_{(k)(j)} = \left[\frac{N}{m^2}\right] = \left[ML^{-1}T^{-2}\right] \quad (19)$$

$$\mathbf{t}_{(k)(j)} = [\rho\ddot{\mathbf{u}}] = \left[ML^{-2}T^{-2}\right] \quad (20)$$

a dimensional analysis of (18) is performed yielding

$$\begin{aligned}
&ML^{-3}\times LT^{-2}\times L^3\\
&+\left[\left(ML^{-1}T^{-2}\times L^{-1}\times L^3\right)\times L\times L^{-1}\right.\\
&\left.+\left(ML^{-1}T^{-2}\times L^{-1}\times L^3\right)\times L\times L^{-1} - ML^{-2}T^{-2}\right]\times L^3\\
&= MLT^{-2} + \left[\left(ML^{-2}T^{-2}\times L^3\right) + \left(ML^{-2}T^{-2}\times L^3\right) - ML^{-2}T^{-2}\right]\times L^3
\end{aligned} \quad (21)$$

where it is noted that the dimensions of the force density appear in

$$\sum_{i=1}^{\infty}\frac{\partial w_{(k)(j)}}{\partial(\mathbf{y}_{(j)}-\mathbf{y}_{(k)})}V_{(i)} \quad (22)$$

The force densities can thus be expressed as

$$\mathbf{t}_{(k)(j)} = \frac{1}{2}\frac{1}{V_{(j)}}\left(\sum_{i=1}^{\infty}\frac{\partial w_{(k)(i)}}{\partial(\mathbf{y}_{(i)}-\mathbf{y}_{(k)})}V_{(i)}\right) \quad (23)$$

$$\mathbf{t}_{(j)(k)} = \frac{1}{2}\frac{1}{V_{(j)}}\left(\sum_{i=1}^{\infty}\frac{\partial w_{(k)(i)}}{\partial(\mathbf{y}_{(i)}-\mathbf{y}_{(k)})}V_{(i)}\right) \quad (24)$$

Rearranging the expanded Lagrange's equation and inserting the force density results in

$$\rho_{(k)}\ddot{\mathbf{u}}_{(k)} = \sum_{j=1}^{\infty} \left[\mathbf{t}_{(k)(j)} - \mathbf{t}_{(j)(k)}\right] V_{(j)} + \mathbf{b}_{(k)} \tag{25}$$

Applying the vector state information extraction notation from (4) on the force density vector state (9), and inserting this in (25) yields

$$\rho_{(k)}\ddot{\mathbf{u}}_{(k)} = \sum_{j=1}^{\infty} \left[\underline{\mathbf{T}}\langle \mathbf{x}_{(k)} - \mathbf{x}_{(j)}\rangle - \underline{\mathbf{T}}\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle\right] V_{(i)} + \mathbf{b}_{(k)} \tag{26}$$

As the amount of material points described in the vector state approaches infinity, the resulting volume of each material point approaches an infinitesimally small value. This results in the infinite sum being expressed as an integral. Since the total volume only involves the summation of the volumes of the material points influencing $\mathbf{x}_{(k)}$, the volumetric domain of the integral will be equal to the family H. The equations of motion can now be formulated as

$$\rho_{(k)}\ddot{\mathbf{u}}_{(k)} = \int_{H_{(k)}} \left(\underline{\mathbf{T}}\langle \mathbf{x}_{(k)} - \mathbf{x}_{(j)}\rangle - \underline{\mathbf{T}}\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle\right) dH_{(k)} + \mathbf{b}_{(k)} \tag{27}$$

or, using matrix notation,

$$\mathbf{M}\ddot{\mathbf{u}} = \mathbf{f}(\mathbf{u}) + \mathbf{b}(\mathbf{u}) \tag{28}$$

where $\mathbf{M}$ is the lumped (diagonal) mass matrix with nodal densities along the diagonal, allowing for a simple inversion to $\mathbf{M}^{-1}$.

The integration formulation in (27) is one of the fundamental benefits of peridynamics. The integral domain is easily modified by removing a material point from the states when a bond is considered broken, allowing for native fracture modelling. Since there are no spatial derivatives in the equation, physical singularities or discontinuations present no issue.

Angular Momentum As the balance of energy and balance of linear momentum are already accounted for by deriving the equations of motion from the principle of virtual work, the angular momentum is derived separately. However, to show conservation of angular momentum, the conservation of linear momentum is applied.

The linear momentum exerted on a set of particles within a volume around a coordinate at a specific time is given by

$$\mathbf{L} = \int_{V_{(k)}} \rho(\mathbf{x}_{(k)})\dot{\mathbf{u}}(\mathbf{x}_{(k)}, t) dV_{(k)} \tag{29}$$

and the total force acting on the same set by

$$\mathbf{F} = \int_{V_{(k)}} \mathbf{b}(\mathbf{x}_{(k)}, t) dV_{(k)} + \int_{V_{(k)}} \int_{H_{(k)}} \underline{\mathbf{T}}(\mathbf{x}_{(k)}, t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)} \rangle dH_{(k)} dV_{(k)} \\ - \int_{V_{(k)}} \int_{H_{(k)}} \underline{\mathbf{T}}(\mathbf{x}_{(j)}, t)\langle \mathbf{x}_{(k)} - \mathbf{x}_{(j)} \rangle dH_{(k)} dV_{(k)} \qquad (30)$$

where $H_{(k)}$, as mentioned before, is the family containing the set of particles interacting with $\mathbf{x}_{(k)}$. If there is balance of linear momentum, all change in momentum must come from the total force acting upon the volume as $\dot{\mathbf{L}} = \mathbf{F}$. Per definition, any material point not part of the family will not affect the material point in question, allowing the integral over the family to be rewritten to contain all the particles within the volume of the contributing material point. Knowing this, the change in linear momentum can be written as

$$\int_{V_{(k)}} \rho(\mathbf{x}_{(k)}) \ddot{\mathbf{u}}(\mathbf{x}_{(k)}, t) dV_{(k)} = \int_{V_{(k)}} \mathbf{b}(\mathbf{x}_{(k)}, t) dV_{(k)} \\ + \int_{V_{(k)}} \int_{V_{(k)}} \underline{\mathbf{T}}(\mathbf{x}_{(k)}, t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)} \rangle dV_{(j)} dV_{(k)} \\ - \int_{V_{(k)}} \int_{V_{(k)}} \underline{\mathbf{T}}(\mathbf{x}_{(j)}, t)\langle \mathbf{x} - \mathbf{x}_{(j)} \rangle dV_{(j)} dV_{(k)} \qquad (31)$$

According to Newton's second law, the forces in the material points described in the force density vector state are equal in magnitude and opposite in direction of each other, implying that

$$\int_{V_{(k)}} \int_{V_{(k)}} \underline{\mathbf{T}}(\mathbf{x}_{(j)}, t)\langle \mathbf{x}_{(k)} - \mathbf{x}_{(j)} \rangle dV_{(j)} dV_{(k)} = \int_{V_{(k)}} \int_{V_{(k)}} \underline{\mathbf{T}}(\mathbf{x}_{(k)}, t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)} \rangle dV_{(k)} dV_{(j)} \qquad (32)$$

Replacing the last double integral on the right-hand side in (31) using (32), and rearranging the remaining terms results in

$$\int_{V_{(k)}} \big(\rho(\mathbf{x}_{(k)}) \ddot{\mathbf{u}}(\mathbf{x}_{(k)}, t) - \mathbf{b}(\mathbf{x}_{(k)}, t)\big) \, dV_{(k)} = 0 \qquad (33)$$

which is valid for arbitrary force density vectors.

The angular momentum conservation condition is reached in a similar manner, by taking the cross product of the deformed coordinate vector and the terms in the linear momentum calculations as

$$\mathbf{H}_0 = \int_{V_{(k)}} \mathbf{y}(\mathbf{x}_{(k)}, t) \times \rho(\mathbf{x}_{(k)}) \dot{\mathbf{u}}(\mathbf{x}_{(k)}, t) dV_{(k)} \qquad (34)$$

where $\mathbf{y} = \mathbf{x}_{(k)} + \mathbf{u}_{(k)}$ is the deformed configuration material point vector. The torque acting on the volume is given by

$$\begin{aligned}\mathbf{\Pi}_0 = &\int_{V_{(k)}} \mathbf{y}(\mathbf{x}_{(k)},t) \times \mathbf{b}(\mathbf{x}_{(k)},t)dV_{(k)} \\ &+ \int_{V_{(k)}}\int_{H_{(k)}} \mathbf{y}(\mathbf{x}_{(k)},t) \times \underline{\mathbf{T}}(\mathbf{x}_{(k)},t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle dH_{(k)}dV_{(k)} \\ &- \int_{V_{(k)}}\int_{H_{(k)}} \mathbf{y}(\mathbf{x}_{(k)},t) \times \underline{\mathbf{T}}(\mathbf{x}_{(j)},t)\langle \mathbf{x}_{(k)} - \mathbf{x}_{(j)}\rangle dH_{(k)}dV_{(k)} \end{aligned} \tag{35}$$

The change of angular momentum must come solely from the total forces, i.e. the torque, expressed as $\dot{\mathbf{H}}_0 = \mathbf{\Pi}_0$, which is given by

$$\begin{aligned}\int_{V_{(k)}} \mathbf{y}(\mathbf{x}_{(k)},t) \times \rho(\mathbf{x}_{(k)})\ddot{\mathbf{u}}(\mathbf{x}_{(k)},t)dV_{(k)} = &\int_{V_{(k)}} \mathbf{y}(\mathbf{x}_{(k)},t) \times \mathbf{b}(\mathbf{x}_{(k)},t)dV_{(k)} \\ &+ \int_{V_{(k)}}\int_{H_{(k)}} \mathbf{y}(\mathbf{x}_{(k)},t) \times \underline{\mathbf{T}}(\mathbf{x}_{(k)},t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle dH_{(k)}dV_{(k)} \\ &- \int_{V_{(k)}}\int_{H_{(k)}} \mathbf{y}(\mathbf{x}_{(k)},t) \times \underline{\mathbf{T}}(\mathbf{x}_{(j)},t)\langle \mathbf{x}_{(k)} - \mathbf{x}_{(j)}\rangle dH_{(k)}dV_{(k)} \end{aligned} \tag{36}$$

Once again it is possible to replace the force density integrals by integrating over the volume instead of the family, and replacing the material point of interest as

$$\begin{aligned}&\int_{V_{(k)}}\int_{V_{(k)}} \mathbf{y}(\mathbf{x}_{(k)},t) \times \underline{\mathbf{T}}(\mathbf{x}_{(j)},t)\langle \mathbf{x}_{(k)} - \mathbf{x}_{(j)}\rangle dV_{(j)}dV_{(k)} \\ =&\int_{V_{(k)}}\int_{V_{(k)}} \mathbf{y}(\mathbf{x}_{(j)},t) \times \underline{\mathbf{T}}(\mathbf{x}_{(k)},t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle dV_{(k)}dV_{(j)} \end{aligned} \tag{37}$$

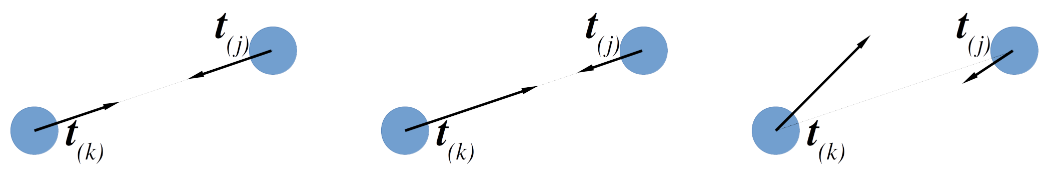
Unlike (33), the resulting equation is not valid for all force density vectors at a glance. Instead the following is yielded

$$\begin{aligned}&\int_{V_{(k)}} \mathbf{y}(\mathbf{x}_{(k)},t) \times \big(\rho(\mathbf{x}_{(k)})\ddot{\mathbf{u}}(\mathbf{x}_{(k)},t) - \mathbf{b}(\mathbf{x}_{(k)},t)\big)\, dV_{(k)} \\ =&- \int_{V_{(k)}}\int_{V_{(k)}} \big(\mathbf{y}(\mathbf{x}_{(j)},t) - \mathbf{y}(\mathbf{x}_{(k)},t)\big) \times \underline{\mathbf{T}}(\mathbf{x}_{(k)},t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle dV_{(j)}dV_{(k)} \end{aligned} \tag{38}$$

The terms $\mathbf{y}(\mathbf{x}_{(j)},t) - \mathbf{y}(\mathbf{x}_{(k)},t)$ can be expressed using the deformed-configuration position vector state in (8) as $(\mathbf{y}_{(j)} - \mathbf{y}_{(k)}) = \underline{\mathbf{Y}}(\mathbf{x}_{(k)},t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle$. If the linear momentum is conserved, the right-hand side must equal zero. Simplified, disregarding the outer integral,

$$\int_{H_{(k)}} \underline{\mathbf{Y}}(\mathbf{x}_{(k)}, t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle \times \underline{\mathbf{T}}(\mathbf{x}_{(k)}, t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle dH_{(k)} = 0 \tag{39}$$

Here, integration is once again performed over the family, since the deformed-state position state vector is defined for the horizon. In the first PD theory [14], (39) was achieved by yielding a cross-product of zero, which occurs if the force vectors are parallel with the position vectors. Equal and opposite force density vectors are the main characteristics of *bond-based* peridynamics, illustrated in Figure 4a. As it is a very limiting case of peridynamics, bond-based peridynamics is not covered in this project. Two other formulations exist as well [15, 13]. They are the *ordinary state-based peridynamics* pictured in Figure 4b and the *non-ordinary state-based peridynamics* seen in Figure 4c. Both are used in the project and will be examined further.



(a) In bond-based peridynamics, the force density vector pair solves (39) being equal in magnitude and parallel in direction.

(b) In ordinary state-based peridynamics, the force density vector pair solves (39) being parallel in direction, but magnitudes are allowed to differ.

(c) In non-ordinary state-based peridynamics, the force density vector pair solves (39) with arbitrary magnitude and direction.

Figure 4: The different force density vectors allowed using the different PD formulations.

Satisfying Angular Momentum using Ordinary State-Based Peridynamics: The first modification away from bond-based peridynamics is allowing the force density vectors different magnitudes, i.e. they will differ by a scalar. The cross-product will still yield zero if they are parallel, satisfying the balance of angular momentum. Knowing this, the force density vector is extracted from the force vector state (10) for material points $\mathbf{x}_{(k)}$ and $\mathbf{x}_{(j)}$, and defined as

$$\mathbf{t}_{(k)}(\mathbf{u}_{(j)} - \mathbf{u}_{(k)}, \mathbf{x}_{(j)} - \mathbf{x}_{(k)}, t) = \underline{\mathbf{T}}(\mathbf{x}_{(k)}, t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle = \frac{1}{2} A \frac{\mathbf{y}_{(j)} - \mathbf{y}_{(k)}}{|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}|} \tag{40}$$

$$\mathbf{t}_{(j)}(\mathbf{u}_{(k)} - \mathbf{u}_{(j)}, \mathbf{x}_{(k)} - \mathbf{x}_{(j)}, t) = \underline{\mathbf{T}}(\mathbf{x}_{(j)}, t)\langle \mathbf{x}_{(k)} - \mathbf{x}_{(j)}\rangle = \frac{1}{2} B \frac{\mathbf{y}_{(k)} - \mathbf{y}_{(j)}}{|\mathbf{y}_{(k)} - \mathbf{y}_{(j)}|} \tag{41}$$

where the values of A and B depend on their material constants, the horizon, and the deformation field. The force density vectors were related to the strain energy densities in (23) and (24), whose magnitude in the direction of deformation is given by the strain energy density function in (12) by

$$\frac{\partial W(\mathbf{x}_{(k)})}{\partial(|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}|)} \tag{42}$$

The values of A and B can now be found explicitly for different types of constitutive models knowing that

$$\begin{aligned}\mathbf{t}_{(k)}(\mathbf{u}_{(j)} - \mathbf{u}_{(k)}, \mathbf{x}_{(j)} - \mathbf{x}_{(k)}, t) &= \underline{\mathbf{T}}(\mathbf{x}_{(k)}, t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle \\ &\propto \frac{\partial W(\mathbf{x}_{(k)})}{\partial(|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}|)} \frac{\mathbf{y}_{(j)} - \mathbf{y}_{(k)}}{|\mathbf{y}_{(j)} - \mathbf{y}_{(k)}|} \qquad (43) \\ \mathbf{t}_{(j)}(\mathbf{u}_{(k)} - \mathbf{u}_{(j)}, \mathbf{x}_{(k)} - \mathbf{x}_{(j)}, t) &= \underline{\mathbf{T}}(\mathbf{x}_{(j)}, t)\langle \mathbf{x}_{(k)} - \mathbf{x}_{(j)}\rangle \\ &\propto \frac{\partial W(\mathbf{x}_{(k)})}{\partial(|\mathbf{y}_{(k)} - \mathbf{y}_{(j)}|)} \frac{\mathbf{y}_{(k)} - \mathbf{y}_{(j)}}{|\mathbf{y}_{(k)} - \mathbf{y}_{(j)}|} \qquad (44)\end{aligned}$$

Satisfying Angular Momentum using Non-ordinary State-Based Peridynamics: The PD formulation allowing arbitrary force vectors is called the non-ordinary state-based peridynamics. It can be derived in several ways, [15] and [13] has two different methods to reach an equivalent expression. As [13] integrates the Piola-Kirchhoff stress tensor in the derivation, further illustrating the connection to CCM, it is the one which will be used here.

A virtual displacement increment $\Delta\mathbf{u}$ is introduced to (27). The material point replacement used in (32) is applied, and the whole expression is integrated across the body. Remembering that the force density relative to material points outside the family is zero, the following is yielded:

$$\begin{aligned}&\int_{V_{(k)}} \left(\rho\ddot{\mathbf{u}_{(k)}} - \mathbf{b}_{(k)}\right) \Delta\mathbf{u}_{(k)} dV_{(k)} \\ &= -\int_{V_{(k)}} \int_{V_{(k)}} \left(\underline{\mathbf{T}}\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle(\Delta\mathbf{u}_{(k)} - \Delta\mathbf{u}_{(\mathbf{j})})\right) dV_{(j)} dV_{(k)}\end{aligned} \tag{45}$$

Here it is noted that the displacement increment $\Delta\mathbf{u}_{(k)} - \Delta\mathbf{u}_{(j)}$ is the equivalent to the increment of the deformed configuration vector state $\underline{\mathbf{Y}}\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle$. As force times distance is dimensionally equivalent to virtual work, the integral on the right-hand side in (45) can be reformulated to the virtual work acting on $\mathbf{x}_{(k)}$ due to internal forces,

$$\Delta W_I = \int_{H_{(k)}} \left(\underline{\mathbf{T}}\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle^T (\Delta\underline{\mathbf{Y}}\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle)\right) dH_{(k)} \tag{46}$$

This allows (45) to be reformulated to

$$\int_{V_{(k)}} \left(\rho \ddot{\mathbf{u}}_{(k)} - \mathbf{b}_{(k)}\right) \Delta \mathbf{u}_{(k)} dV_{(k)} = - \int_{V_{(k)}} \Delta W_I dV_{(k)} \tag{47}$$

The ability to incorporate classical continuum constitutive formulations into peridynamics originates from the above reformulation of virtual work. Comparing the virtual work formulated in peridynamics (48) and in CCM (49), using matrix notation:

$$\Delta W_I = \int_{H_{(k)}} \left(\underline{\mathbf{T}}\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle^T (\Delta \underline{\mathbf{Y}}\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle\right) dH_{(k)} \tag{48}$$

$$\Delta \hat{W}_I = \mathrm{tr}(\mathbf{S}^T \Delta \mathbf{E}) \tag{49}$$

In the CCM formulation, $\mathbf{S}$ is the second Piola-Kirchhoff stress tensor, and $\mathbf{E} = \frac{1}{2}(\mathbf{F}^T\mathbf{F} - \mathbf{I})$ is the Green-Lagrange strain tensor. Here the deformation gradient tensor is denoted with $\mathbf{F}$. Knowing the first Piola-Kirchhoff stress tensor $\mathbf{P} = \mathbf{S}^T\mathbf{F}^T$, (49) is reformulated to

$$\Delta \hat{W}_I = \mathrm{tr}(\mathbf{P} \Delta \mathbf{F}) \tag{50}$$

Applying reduction using (5), and performing a significant reformulation, (50) is expressed as

$$\Delta \hat{W}_I = \int_{H_{(k)}} \left(\underline{w}\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)}\rangle \mathbf{P}\mathbf{K}^{-1}(\mathbf{x}_{(j)} - \mathbf{x}_{(k)})\right)^T \left(\Delta \underline{\mathbf{Y}}(\mathbf{x}_{(k)}, t)\right) dH_{(k)} \tag{51}$$

where $\underline{w}$ is a weight function, not to be confused with the scalar micro-potential $w_{(k)(j)}$, and $\mathbf{K}$, as mentioned before, is the shape tensor.

With these final steps, balance of angular momentum has been shown to be satisfied for arbitrary force vectors, using virtual displacement, if the internal virtual work can be formulated using (51), which is possible if the first Piola-Kirchhoff stress tensor can be derived for the material. The non-ordinary state-based PD force vector can now be expressed as

$$\begin{aligned}\mathbf{t}(\mathbf{u}_{(j)} - \mathbf{u}_{(k)}, \mathbf{x}_{(j)} - \mathbf{x}_{(k)}, t) &= \underline{\mathbf{T}}(\mathbf{x}_{(k)}, t)\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)} \rangle \\ &= \underline{w}\langle \mathbf{x}_{(j)} - \mathbf{x}_{(k)} \rangle \mathbf{P}\mathbf{K}^{-1}(\mathbf{x}_{(j)} - \mathbf{x}_{(k)}) \qquad (52)\end{aligned}$$

and thus, the necessary balance laws have been formulated.

2.2.6 Boundary Conditions

The PD equations of motions only necessitates initial displacement and velocity conditions for time-integration. But in order to model more complex systems, time and spatial constraints, as well as external loads, can also be applied. How to apply them differs compared to CCM formulations, as different conditions have to be applied to different kinds of volumetric regions of nodes, shown for 2D in Figure 5 on the next page. Since this project did not make use of external loads, it will not be covered in the theory section. For further reading on external loads in PD, see [13].

Initial conditions are straight-forward. Every material point on the domain R is given an initial displacement $\mathbf{u}^*(\mathbf{x})$ and velocity $\mathbf{v}^*(\mathbf{x})$, their initial spatial gradients can also, if needed, be specified as the spatial displacement gradient $\mathbf{H}^*$ and spatial velocity gradient $\mathbf{L}^*$.

Constraint conditions, as mentioned before, are not necessary, but useful to prevent rigid body motion, as well as impose displacements. Since peridynamics is a non-local theory, the constraints cannot simply be applied on surfaces. Instead a virtual boundary domain R_c is introduced, onto which the constraint is applied. Its thickness should be the same as the horizon δ [20], to make sure that the true boundary fully experiences the value from that part of the family.

2.2.7 Damage Modelling

In peridynamics, the architecture of the state vectors are used to introduce damage to the material, by removing any information on the micro-potential regarding another node from the state vector of the node in question. Removing the micro-potential between material points breaks the bond irreversibly. When to break the bond can be decided in two ways [21]: *Critical Energy* and *Critical Stretch*. As the critical

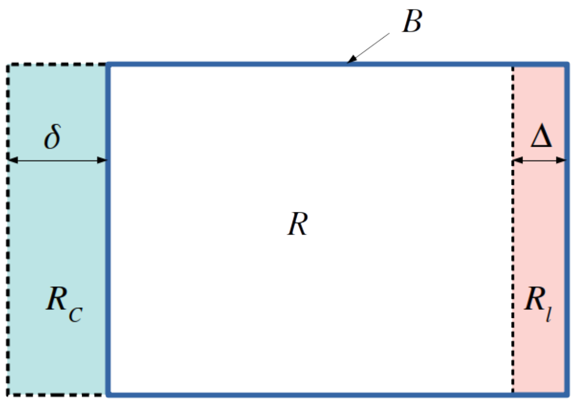


Figure 5: An illustration of the different condition regions of the PD domain. R denotes the full domain, R_l is the sub-region of R where an external load is applied, B denotes the boundary of R, and R_c denotes a fictitious boundary layer for constraint conditions.

energy criterion implemented in Peridigm is only valid for certain elastic materials only, it is not of any use to the project, and will not be covered in this report.

Using the critical stretch, the bond between two material points is broken when the stretch between them reaches a predefined value. This value can be either static, or time dependent [22] when dealing with for example viscous material or creep. The formulation of the critical stretch is part material model dependent, and part computation specific due to horizon dependence originating from the micro-potentials [13]. This means it can be derived for every material, but the values of its dependencies might be hard to find analytically.

For example, the micro-brittle 3D PD material model has a critical stretch which depends on the horizon, bulk modulus, shear modulus, and the critical energy release rate as [21]

$$s_c = \sqrt{\frac{G_{0c}}{(3G + (3/4)^4)\,(K - 5G/3)\,\delta}} \tag{53}$$

Finding the correct value of s_c using a similar analytical method for other, even more complex material models would be a very time-consuming exercise. Instead, an iterative method of finding the correct value can be used, which boils down to running repeated simulations, varying only the critical stretch, until the point of fracture corresponds to experimental data.

As a bond is broken, the stress is distributed to the other material points still bonded to the material point in question. This increase in stress promotes further bond breakage, leading to a progressively growing fracture [13]. As an indicator of the amount of interactions eliminated from a material point, relative to the interactions present at initiation, the local damage [18] can be used. Local damage ranges from 0 to 1, where a 0 indicates that all interactions are intact, and a 1 is equivalent to all initial interactions being severed. Introducing the individual interaction indicator μ, or failure parameter for short, and attributing the following boolean expression

$$\mu = \begin{cases} 1 \text{ if bond stretch below critical stretch} \\ 0 \text{ else} \end{cases} \tag{54}$$

allows the local damage to be formulated as

$$\varphi(\mathbf{x}_{(k)}, t) = 1 - \frac{\int_{H_{(k)}} \mu(\mathbf{x}_{(j)} - \mathbf{x}_{(k)}, \mathbf{t}) dV_{(j)}}{\int_{H_{(k)}} dV_{(j)}} \tag{55}$$

2.3 Peridigm

Peridigm is an open-source peridynamic simulation program developed at Sandia National Laboratories [23]. It was first published in 2011, and has subsequently been developed continuously by Sandia and a small community of private contributors. The latest source code is available on GitHub [24]. Some general Peridigm theory is presented below, summarising some of the important aspects regarding usage. A thorough explanation of the software has been made by Deutsches Zentrum für Luft- und Raumfahrt (DLR) [21], they also provide an installation guide, both of which are available in the PeriDoX package on GitHub [25].

2.3.1 Input - Output

Input is given by providing an *input deck* written in either `XML` [26] or `YAML` [27] format, and a discretization containing information on the material points. The available option sections [21] and their respective purpose are shown in Table 1 on the following page. An example of a `YAML` input deck is found in Appendix B. Code is written using significant whitespace, meaning that the indentation guides the code interpreter. Tabs and spaces are not equal, and either one should be used consequently in the whole file. Input which is supposed to yield a value can be parsed using either functions or the value specified as an integer or double. Writing functions in the input deck is necessary when velocities are used, as the time variable t can be used, or when imposing displacements related to coordinates x, y, or z.

Table 1: The various options sections available in Peridigm, and their purpose.

Verbosity	The amount of information printed in the Peridigm output.
Discretization	Specifies the file type and file name of the domain to be used.
Materials Models	Specifies the different material models and their corresponding variables. Different material models are referenced by naming.
Damage Models	[Optional] Specifies the different damage models and their corresponding variables. Referenced by naming
Contact Models	[Optional] Specifies how regions not part of each others' families behave when in contact. Not used in this project
Blocks	Specifies the material, damage, and contact models to be applied to the different blocks.
Boundary Conditions	Specifies the different boundary and initial conditions applied during the simulation.
Solver	Specifies time interval and time integration schemes.
Compute Class Parameters	[Optional] Specifies eventual extra computations wished to be performed during a load step.
Output	Specifies which variables to save, how often, and to what file in which format.

2.3.2 Discretization

Two inputs are required for the discretization: *which* file to read, and what *file-type* it is. The two currently available file types [21] are *Text* files and *Exodus II* files [28].

Text file input is easily written by hand but requires a certain format. Both a *domain discretization* text file as well as auxiliary *node set* text files are needed for this. The domain discretization file contains information on coordinates, block ID and the volume of the node, arranged as below, each row being the corresponding node ID.

	Column 1	**Column 2**	**Column 3**	**Column 4**	**Column 5**
Node ID	X	Y	Z	Block ID	Volume

The node set text file simply contains the node ID of the nodes belonging to the set, on individual rows as below:

	Column 1
Row	Node ID

Peridigm I/O is streamlined to work with the Exodus II file format, and the text files generated should be converted to this format using the provided `text_to_genesis.py` script available in the source code on GitHub [29].

Blocks are used to define the type of material, damage, and contact properties the material points will exhibit, as well as specify nodes for the compute class parameters. Node sets are used by boundary conditions and compute class parameters in the same way.

Bond filters can be implemented to create pre-cracks. They prevent bonds from being created during initialization and will result in crack initiation during simulation. It can be defined using different 2D shapes: rectangular plane, disk, or Exodus mesh [30].

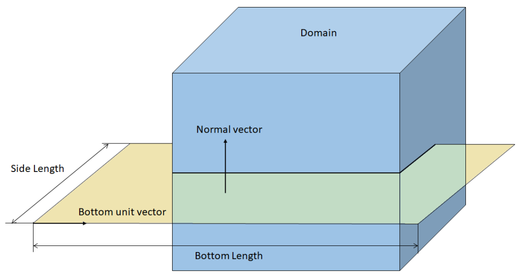


Figure 6: A bond-filter used for pre-cracking, defined by a rectangular slice.

2.3.3 Material Models in Peridigm

Several material models exist, they are listed in [21], and in the source code materials folder of Peridigm. Bond-based as well as ordinary and non-ordinary state-based materials exist. The latter two are denoted as non-correspondence and correspondence material models respectively in Peridigm. The available material models as of the beginning of spring 2020 are:

- Elastic (bond-based, non-correspondence, and correspondence)
- Position-aware elastic [Pals] (non-correspondence)
- Partial-volume solid (volumetric elasticity in nodes. Non-correspondence)
- Elastic-plastic (Perfect plasticity. Non-correspondence, correspondence)
- Elastic-Plastic Hardening (non-correspondence, correspondence)
- Viscoelastic (non-correspondence)
- Needleman viscoelasto-plastic (correspondence)

- Several non-documented material models in the source code.

This project has made use two material models: The *Isotropic Hardening Correspondence* model [21], and the *Elastic Plastic Hardening* model [31], which both exhibit the behaviour shown in Figure 7. This is due to plasticity being a necessary component in the modelling of the material, disregarding all non-plasticity models. Since perfect plasticity does not increase stress levels following yielding, it is not suitable either. The Needleman viscoplastic material model is not well documented in [21], and requires many parameter inputs, increasing calibration time and is as such not deemed a useful model for this project.

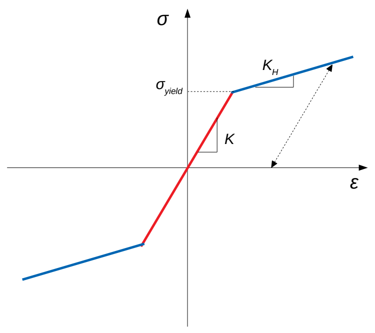


Figure 7: The stress-strain behaviour of the Peridigm elastic-plastic hardening models. Adapted from [21].

The elastic plastic hardening models feature linear elasticity as well as linear plasticity, the inclinations being the bulk modulus and hardening modulus, respectively. The non-correspondence elastic plastic hardening material model is governed by seven variables:

- Density: ρ
- Bulk modulus: K
- Shear modulus: G
- Elasticity modulus: E
- Poisson's ratio: ν
- Yield stress: σ_{yield}
- Hardening modulus: K_H

When using the correspondence model, an extra variable is introduced for calculation of hourglass force densities, which are added to the regular force density to minimise low- and zero-energy modes [32].

- Hourglass coefficient: C_{hk}

Peridigm code uses K and G for calculations, but engineering constants (G, E, ν) can be used instead as they are recalculated in Peridigm into K and G using the relations

$$K = \frac{E}{3(1-2\nu)} \tag{56}$$

and

$$G = \frac{E}{2(1+\nu)} \tag{57}$$

It makes no difference to the result, so the choice is purely dependent on which data is closest at hand. If used, only two engineering constants are needed as the third constant can be derived from (57). C_{hk} is used for calculation of hourglass force densities which are added to the regular force density to minimize low- and zero-energy modes [32].

The PD yield criterion used in Peridigm for plasticity [31], is equivalent to the CCM Von Mises plasticity model [33], which assumes that the stress is decomposable into an elastic and a plastic part, that the elastic region is confined within a deviatoric-state-dependent yield surface, and that only deviatoric stresses (non-hydrostatic stresses) will induce yield. The yield function (yield surface) is given by [31]

$$f = \psi - \psi_0 = 0 \tag{58}$$

where ψ_0 is the yield point of the material and $\psi = \frac{\|\mathbf{t}^d\|^2}{2}$ is a material flow function where $\mathbf{t}^d$ is the deviatoric force state. It is illustrated in Figure 8, as seen from the deviatoric plane in the stress space.

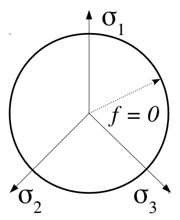


Figure 8: The von Mises yield surface, illustrating (58). $f < 0$ denotes the inner area of the circle, while outside the circle $f > 0$. On the boundary between them $f = 0$.

Together with the consistency parameter λ, f and λ must satisfy the Karush-Kuhn-Tucker complementary conditions [33]:

$$\lambda \geq 0 \tag{59}$$

$$f \leq 0 \tag{60}$$

$$\lambda f = 0. \tag{61}$$

The values of f and λ under these restrictions indicates whether the increment is plastic or elastic. A value of $f < 0$ implies an elastic increment if $\lambda = 0$, while $f = 0$ in combination with $\lambda > 0$ indicate that the stress has reached the yield surface.

The three-dimensional yield point (the distance from zero loading to the yield surface) is related to the yield stress. In Peridigm it is given by [34]

$$\psi_0 = \frac{25.0\sigma_{yield}}{8\pi\delta^5} \tag{62}$$

This is used in a *backward Euler* (*fully implicit*) scheme [33] to derive the value of the increment of λ resulting in

$$\Delta\lambda = \frac{1}{\alpha}\left[\frac{||\mathbf{t}^d_{iter}||}{\sqrt{2\psi_0}} - 1\right] \tag{63}$$

if the step is plastic. Otherwise $\Delta\lambda = 0$, and the force state will be calculated elastically. In order to prohibit $f > 0$, the yield surface must increase further to ensure that f stays at zero, ensuring that that $\dot{f} = 0$. This increase makes the increment a plastic loading increment. It is done by, using the Von Mises isotropic hardening [33], defining a hardening parameter $K = K(K_H)$ which is not to be confused with the bulk modulus, as well as a new yield function F, and using these to redefine the initial yield function as:

$$f = F - K = 0 \tag{64}$$

Specifics on the correspondence model: The *correspondence* materials are the Peridigm equivalents of the PD *non-ordinary state-based* materials. According to [21], they allow classical continuum output such as Von Mises and Cauchy stress tensor outputs, as well as proper surface treatment. Surface calculations is an issue for non-correspondence materials, as the calculations are made based on the assumption that the material point is completely surrounded by its family, i.e. in the bulk. On the boundaries where the material point is missing bonds, calculations will be erroneous. This can be corrected using surface correction algorithms, but the correspondence does not require any correction. Instead, they require that every material point has at least three non-parallel bonds, as these are needed for the computation of a proper deformation gradient.

The material can also feature spatial flaws [35], a region of the material where the yield strength is reduced using the formula

$$\sigma_{f,yield} = \sigma_{yield}\left(1 - C_{f,m}\exp\left\{\frac{-\left((x-x_f)^2+(y-y_f)^2+(z-z_f)^2\right)}{C_{f,s}^2}\right\}\right) \tag{65}$$

where $\sigma_{f,yield}$ is the resulting yield stress from the flaw, $C_{f,m}$ is the flaw magnitude expressed as part of total yield stress, $C_{f,s}$ is the physical flaw size, and x_f, y_f, and z_f are the coordinates of the flaw centre.

Conversion of engineering stress-strain to true stress-strain The engineering strain is defined as [3]

$$\varepsilon_e = \frac{\Delta L}{L_0} \tag{66}$$

where ΔL is the elongation, and L_0 is the initial length of the domain. The true strain is derived from (66) using the relation

$$\varepsilon_t = \ln\left(\frac{L}{L_0}\right) = \ln\left(\frac{\Delta L + L_0}{L_0}\right) = \ln(\varepsilon_e + 1) \tag{67}$$

The engineering stress is defined as [3]

$$\sigma_e = \frac{F}{A_0} \tag{68}$$

where F is the force and A_0 is the initial cross-sectional area. The true stress is calculated for the instantaneous area A as

$$\sigma_t = \frac{F}{A} \tag{69}$$

Assuming that the total volume is preserved, the instantaneous area A can be found knowing the elongation as well as the initial area and length

$$V = AL = A_0L_0 \implies A = \frac{V}{L} = \frac{A_0L_0}{L} \tag{70}$$

which gives

$$\sigma_t = \frac{F}{\frac{A_0L_0}{L0+\Delta L}} = \sigma_e(1+\varepsilon_e) \tag{71}$$

2.3.4 Damage Models in Peridigm

Peridigm implements the critical stretch model in three different ways. *Standard*, *time dependent*, or *interface aware*. The standard and time dependent critical stretch models can only be utilised for bond-based and non-correspondence state-based materials. For correspondence materials, an issue arises with bond breaking. A material point might end up with less than three non-parallel bonds, which will break the simulation as mentioned previously. The interface aware damage model contains a check, which makes sure that no *rank deficient* node will be part of the deformation gradient calculation. Instead, they are put in a certain node set and removed from the computation. This is done by invoking the command

```
    Create Node Set For Rank Deficient Nodes: true
```

shown in the Boundary Conditions section in Appendix B. Without this, the simulation will break. This information is not available in the documentation, but is understood from the source code of the interface aware damage model [36].

Apart from the node check, interface aware and standard critical stretch behave the same. As illustrated in Figure 9, a certain amount of work w_0, defined using the critical stretch s_c will result in the bond breaking. Until that point is reached, the bond can be stretched and contracted infinitely. When the bond breaks, further stretching will not contribute any force to the material point bond.

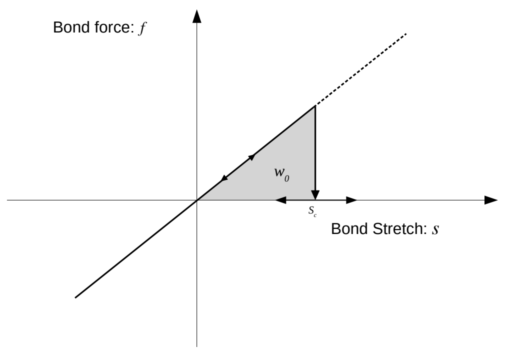


Figure 9: The schematic force-stretch behaviour of the Peridigm critical stretch model. Adapted from [21].

2.3.5 Boundary Conditions

Peridigm implements both constraint and initial conditions, but does not as of yet implement periodic boundary conditions. Body forces and displacements are enough for most applications, as the coordinate and time variables x, y, z, and t allow for rates of change to be expressed using the function parser. They can be specified for either blocks or node sets. Each boundary condition is given a unique name and a node list or block which identifies the nodes the boundary condition applies to. It also defines what kind of boundary condition it is. Several ones can be found in the source code, but some of them have not been implemented yet [21]. Depending on what type it is, the relevant attributes are specified as well. For a displacement boundary condition, a magnitude value and direction coordinate are given, one for each Cartesian coordinate. As mentioned before on page 28, the setting for rank deficient nodes has to be specified in this section if the interface aware damage model is used.

2.3.6 Solvers

Several numerical solvers exist in Peridigm, utilising different time integration schemes as well as quasi-static schemes. For dynamic simulations, *explicit* and *implicit* solvers exist. An explicit time integration scheme uses only values from the current time step to calculate the result of the next time step, while an implicit time integration scheme uses data from both the current as well as the next time step to calculate the next time step.

The explicit time integration scheme is recommended for damage and fracture modelling [21], and consists of a *Velocity Verlet* algorithm [37]. Given the discretised equation of motion (28) at time step n

$$\mathbf{M}\ddot{\mathbf{u}}^n = \mathbf{f}^n(\mathbf{u}^n) + \mathbf{b}^n(\mathbf{u}^n) \tag{72}$$

and the explicitly stored velocity $\mathbf{v}^n$, the velocities, external and internal forces, as well as displacements for the next load steps are calculated using the algorithm below [38].

1. Calculation of the half step velocity: $\mathbf{v}^{n+1/2} = \mathbf{v}^n + \frac{\Delta t}{2}\mathbf{M}^{-1}(\mathbf{f}^n + \mathbf{b}^n)$
2. Calculation of the new displacement: $\mathbf{u}^{n+1} = \mathbf{u}^n + \Delta t\mathbf{v}^{n+1/2}$
3. Re-evaluation of new force densities: $\mathbf{f}^{n+1}$, $\mathbf{b}^{n+1}$

4. Calculation of the new velocity: $\mathbf{v}^{n+1} = \mathbf{v}^{n+1/2} + \frac{\Delta t}{2}\mathbf{M}^{-1}(\mathbf{f}^{n+1} + \mathbf{b}^{n+1})$

Explicit time integrations are subjected to conditional stability [33], meaning that in order to prevent errors from diverging as time moves forward, a critical time step can be found, above which the solution is unstable. The time step allowed is related to information flow speed in the discretisation via a spring constant. It is calculated by finding the lowest critical time step among all material points as [39]

$$\Delta t_{crit} = \min \sqrt{\frac{2\rho}{t_d^k}}\ , k \in R \tag{73}$$

where the time step denominator for a material point $\mathbf{x}_{(k)}$is

$$t_d^k = \int_H \frac{V_j k_j}{(\mathbf{x}_{(k)} - \mathbf{x}_{(j)})} dH \tag{74}$$

where k_j is the spring constant. The same spring constant is used for every material in Peridigm, but actually based on the derivation of the bond constant for isotropic compression loading, through equation of the PD and CCM strain energy densities.

$$W_{PD} = \frac{1}{2}\int_H w dH \tag{75}$$

$$W_{CCM} = \frac{1}{2}\sigma_{ij}\varepsilon_{ij} \tag{76}$$

into

$$\frac{1}{2}\int_H w dH = \frac{1}{2}\sigma_{ij}\varepsilon_{ij} \tag{77}$$

The PD formulation, expressed in spherical coordinates since the family H is a sphere, is

$$W_{PD} = \int_{\xi=0}^{\delta}\int_{\varphi=0}^{2\pi}\int_{\theta=0}^{2\pi} \frac{ks^2}{4}\xi^3 \sin\theta d\xi d\theta d\varphi = \frac{ks^2\pi\delta^4}{4} \tag{78}$$

where s is the stretch and δ is the horizon. The classical continuum formulation, using Hooke's law for the strain energy density

$$\sigma_{ij} = 3(K-\lambda)\varepsilon_{ij} + \lambda\varepsilon_{kk}\delta_{ij} \tag{79}$$

and

$$\lambda = \frac{3K}{5} \tag{80}$$

results in

$$W_{CCM} = \sum_i \left(\sum_j \frac{6K}{10}\varepsilon_{ij}^2 + \frac{3K}{10}\sum_k \varepsilon_{kk}\varepsilon_{ii} \right) \tag{81}$$

Uniaxial loading allows the strain tensor to be written as

$$\varepsilon_{ii} = s \tag{82}$$

yielding

$$W_{CCM} = \frac{9}{2}Ks^2 \tag{83}$$

Solving (77) using (78) and (83) for k gives

$$k = \frac{18K}{\pi\delta^4} \tag{84}$$

which is denoted as k_j in (74), and is a semi-material-parameter, as it depends on both the bulk modulus as well as the horizon. The spring constant can also be used to calculate resonance frequencies ω using the relation

$$\omega = \sqrt{\frac{k}{m}} \tag{85}$$

where m is the mass of the node.

The critical time step is usually very low, presenting an issue when dealing with low strain rates. The explicit time integration scheme therefore is mostly suitable for short simulations of time intervals. When the time step criteria can be met however, the explicit solution solves the equations of motion without the use of iterations [33].

If the time step criteria cannot be met, an implicit solver can be used instead. They are less straight forward, requiring more calculations per load step as well as the usage of iterations, but allows for larger time steps. Depending on whether the number of iterations or the number of time steps required is the highest, the corresponding numerical solver should be used. The implicit solver implemented in Peridigm is the *Newmark-β* method [37]. It is not used in this project and therefore not explained further. Note that the fully implicit backward Euler method used for calculation of $\Delta\lambda$ in (63) is not governed by the choice of time-integration scheme.

If the left-hand side of (72) can be neglected, it is discretised into virtual time steps as

$$-\mathbf{f}^n(\mathbf{u}^n) = \mathbf{b}^n(\mathbf{u}^n) \tag{86}$$

This corresponds to the Quasi-static solver schemes. Peridigm features two different kinds: the regular *QuasiStatic*, and a scheme from one of the dependencies of Peridigm [40]. It is the Trilinos-developed *NOXQuasiStatic* solver. The NOXQuasiStatic method requires a general preconditioner, load step direction method, jacobian operator and preconditioner, as well as manually specifying the polynomial line search method [21]. The regular QuasiStatic method does not require these, but do require more numerical variables to be specified. As a result, it requires more adjustment of values for every new simulation and this is assumed to be less favourable for the project. Neither method is well documented in either the source code or the user manuals [38, 21, 37].

2.3.7 Running Peridigm

To run Peridigm, the input deck must be written and the domain discretisation must be provided. If the discretisation is specified as a text file, the file name is simply written. If the discretisation is an Exodus II file, it must be decomposed to the number of CPU cores, denoted as `<np>`, sought to be used. This is done by running the decomp script available in the Trilinos source code [41]. For an arbitrary decomposition, the command is

```
python decomp.py <filename>.g -p <np>
```

The discretisation file name is specified without the decomposition indicator "`.<total cores>.<np>`" in the input deck.

Peridigm is invoked in a Linux shell terminal either in serial, or in parallel. In serial, the command is simply:

```
Peridigm <filename>.yaml
```

Since the discretisation is declared in the yaml file, no further input is required for Peridigm. Preferably, the output is stored in a log file, storing error and regular output separately is done as:

```
Peridigm <filename>.yaml >PeridigmOut.log 2>PeridigmErr.log
```

or combined:

```
Peridigm <filename>.yaml >PeridigmOut.log 2>&1
```

As time is often a limited resource, parallel Peridigm is more suitable for large computations. It is made by calling the MPI interface [42], which is used to communicate data between the processors. For an arbitrary number of cores, the shell command is

```
mpirun -np <np> Peridigm <filename>.yaml >PeridigmOut.log 2>&1
```

The data is saved in the Exodus II output format, for every decomposed file separately. To combine these files into one single data file, they are merged using the script available in the Peridigm source code:

```
python MergeFiles.py <filename>.e 20
```

This can now be read using Paraview [43], where data can be presented in a comprehensible manner, and exported to other formats.

3 Method

In order to investigate if fracture of HS materials can be sufficiently modelled using peridynamics in Peridigm, realistic behaviour for the UFG and CG materials must be formulated and combined in a domain mirroring the topology of experimental HS materials. The steps required to achieve this are: generating a domain, finding proper constitutive models for both deformation and damage, and finally combining the models in bi-modal simulations an HS domain. The different steps and their subsequent goals are listed below for clarity.

The general procedure:

1. Domain generation:
    - Scalable resolution for mesh-independence studies.
    - Reasonable resolution interval for the available computational resources.
    - Domain for calibration
    - Ideal HS domain.
2. Material calibration: UFG and CG calibrated separately
    - Material model - Fit general experimental data on stress-strain curves.
    - Damage model - Show proper necking and fracture.
3. Harmonic structure simulations: UFG and CG combined
    - Evenly distributed domain - Comparable with current experimental data.

3.1 Software

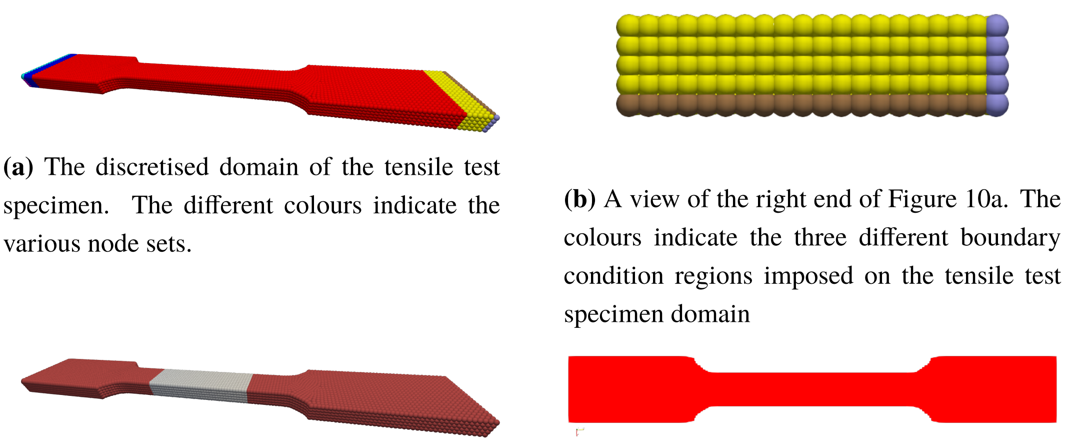
For domain generation, Matlab [44] was used. The open-source finite element mesh generation program Gmsh [45] was used initially but abandoned due to issues with the scalability and random generation of coarse-grained centres. Data points based on the experimental data in Figure 13a on page 39 and Figure 15a on page 41 was measured using GIMP [46] and subsequently loaded and analysed in Matlab. The simulations are carried out using Peridigm [23] and its dependencies, with parallelisation made possible using OpenMPI [42]. The data files are loaded into Paraview [43], where images and data are exported. The data is then read in Matlab for stress and strain calculations.

3.2 Domain and Discretisation

Two different domains are used:

- A standard homogeneous tensile test specimen pictured in Figure 10a, which is available as part of a tensile test example in the Peridigm source code. In this example, the material model is purely elastic without any damage model. Therefore, it has been verified neither for plastic deformation nor fracture. The domain is used for calibrating the two materials separately.
- A rectangular heterogeneous three-dimensional block pictured in Figure 11a on the next page, similar to the domain used in [7]. This domain is used for the harmonic structured simulations, as well as calibration of materials.

3.2.1 Standard Tensile Test Specimen


(a) The discretised domain of the tensile test specimen. The different colours indicate the various node sets.

(b) A view of the right end of Figure 10a. The colours indicate the three different boundary condition regions imposed on the tensile test specimen domain

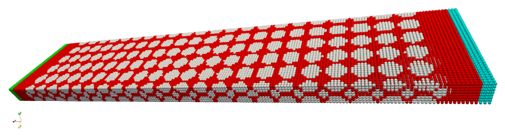
(c) The gauge region of the tensile test specimen domain highlighted in white.

(d) A two-dimensional view of the tensile test specimen.

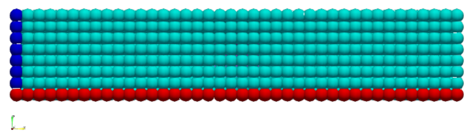
Figure 10: Four views of the standard tensile test specimen used in this project.

As the domain is already discretised in Peridigm, the domain cannot be subjected to refining or coarsening, and is used as-is. The initial gauge measurements, the gauge being indicated in Figure 10c, are 24.8 mm long (y), 5.7 mm wide (x), and 2.5 mm deep (z). The full specimen measures 100.0 mm long, 12.0 mm wide, and 2.5 mm deep. Three boundary conditions are applied on each side of the specimen, pictured in Figure 10b for the right side. A displacement of $y \times 0.005 \times t$ m, equivalent to a displacement rate of $y \times 0.005\ \mathrm{m\,s^{-1}}$ is applied on the bulk (yellow) nodes. Also, rigid body motion is prevented by enforcing a displacement of 0 m in x and z applied on the right (purple) nodes and the on the bottom (brown) nodes, respectively.

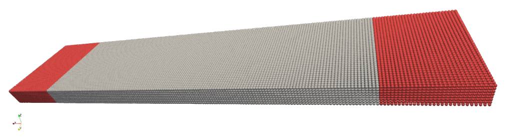
3.2.2 Thin Rectangular Block



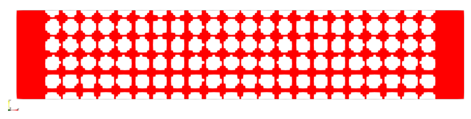
(a) The discretised domain of the harmonic structured material. The two blocks are illustrated in white (CG) and red (UFG). The colours of the end sides shows two different node sets.



(b) A view of the right end domain of 11a. The colours show the final three boundary condition regions imposed on the HS domain.



(c) The gauge region of the HS domain highlighted in white.



(d) A two-dimensional view of the HS domain.

Figure 11: Four views of the harmonic structured domain used in this project.

The domain chosen is similar to the one used in [7]. It is generated using the code in Appendix F, which automatically adjusts the radius of the CG regions to ensure that a certain percentage of the nodes will be UFG. The fraction chosen is 40 % UFG nodes, to be equivalent to both [7] and [4]. The domain is oriented differently from the tensile test specimen, x being the length, y being the width, and z being the depth. The size of the domain measures 5.0 mm in x, 1.0 mm in y, and a fifth of the width being $(1.0/5 = 0.2)$ mm in z. Several boundary conditions for the prevention of rigid body motion are tested, and the final choice, being the same as the tensile test specimen, is pictured in Figure 11b. The bulk (turquoise) nodes are displaced in x direction, nominally at a rate of $x \times 3.6e - 3\ \mathrm{m\,s^{-1}}$, but higher magnitudes are used in the Verlet solver simulations. The bottom (red) nodes prevent rigid body motion in z, and the left (blue) nodes prevent rigid body motion in y.

Some other boundary conditions are also examined on the harmonic structured domain. The application regions are shown in Figure 12 on the next page, and the boundary conditions applied to the different node sets (NS) are

- Preventing the whole domain from displacement in z.

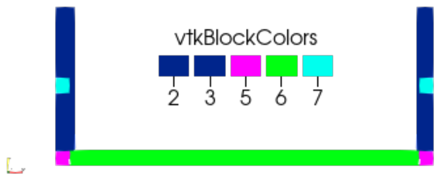


Figure 12: Schematic representation of different node sets (indicated in Paraview as vtk-BlockColors) used for preliminary boundary conditions in the harmonic structured domain in Figure 11d on the facing page. For clarity on the dark blue node sets, NS 2 denotes the left end, and NS 3 denotes the right end.

- Enforcing an artificial shrinking of the boundary region nodes in y, as the domain expands in x (NS 2 and 3).
- Prevention of y-movement of the bottom border in x (NS 6).
- Two centre end points locked in y-direction (NS 7).

These alternative boundary conditions and the resulting issues are derived from trying to conform to the theory of having a boundary region the size of the horizon. The tensile test example in Peridigm does not utilise this principle for the rigid body motion boundary conditions, showing that it is not necessary nor suitable to have a full horizon region for such a boundary condition. However, neither is assigning the boundary condition to one node only, as this would result in a too large force acting on the single node. For the application of a displacement affecting the rest of the domain, a boundary condition region, as opposed to a boundary condition *surface*, is applied. Due to this necessity, an extra width is added where the boundary conditions are applied, and the CG blocks are not generated here. This is illustrated in Figure 11a on the facing page, where the coarse grains are not present at the endings. As a result, the UFG fraction calculation is not based on the full domain, and instead calculated only in the region where both UFG and CG is present.

The gauge region, designated by white in Figure 11c on the preceding page, is even smaller, to mitigate the effect of the boundary region on parts of the domain. As Peridigm as of spring 2020 have not implemented two-dimensional simulations, the domain is a thin slice of the three-dimensional FCC structure to reduce computational effort. Its outer boundaries can be adjusted freely, and is here chosen to contain one unit cell of FCC in the depth (z) direction. Besides the FCC domain, a 2D "random" domain structure is also created using the code in Appendix G. Its purpose is to be used for random-distributed simulations, in order to compare topology effects on the material. These simulations are superseded in priority by calibration and the HS simulation due to time restrictions, and thus not performed. It therefore only included as a reference for further work

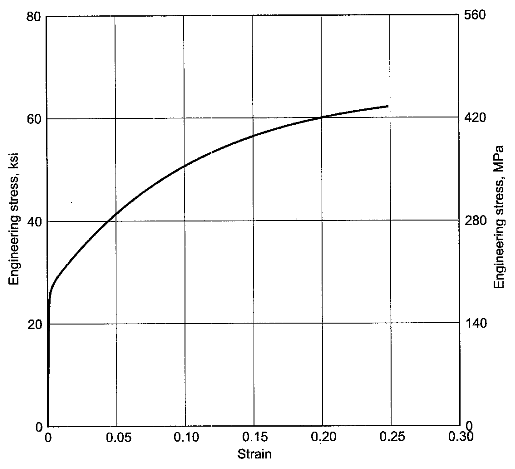
The discretisation is saved to a domain file and corresponding node set files in `.txt` format, and subsequently converted to ExodusII using the `text_to_genesis.py` script, as explained in the theory. Furthermore, for parallelisation the domain is decomposed using the `decomp.py`-script into as many parts as processor cores used for the simulation.

3.3 Material and Damage Models

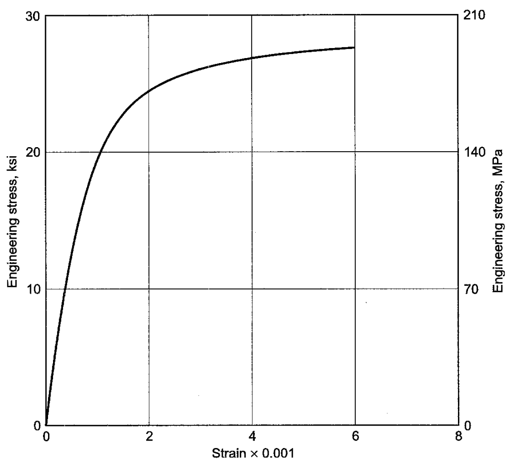
Experimental engineering stress-strain data to calibrate the models with has been found for both the CG and the UFG material. It is discretised using GIMP 2 [46] and converted to true stress-strain and presented in figures using the Matlab code in Appendix J. A power law function on the form $f = kx^y$ based on the plastic stress points are fitted to the plastic stress data available using a linear fit on the log-log plot of the true stress strain curves. Since no data on the cross-sectional area is available after the ultimate tensile strength point, true stress cannot be derived analytically using (71) after this point since knowledge of the complex stress state within the neck region is unknown. The values are found in SI units, but depending on the units of the discretisation, they can be converted between different unit systems. The tensile test specimen domain is defined in centimetres, necessitating a conversion from SI to CGS (Centimetre-Gram-Second). Conversion tools are readily available online.

3.3.1 Coarse Grain Material Model

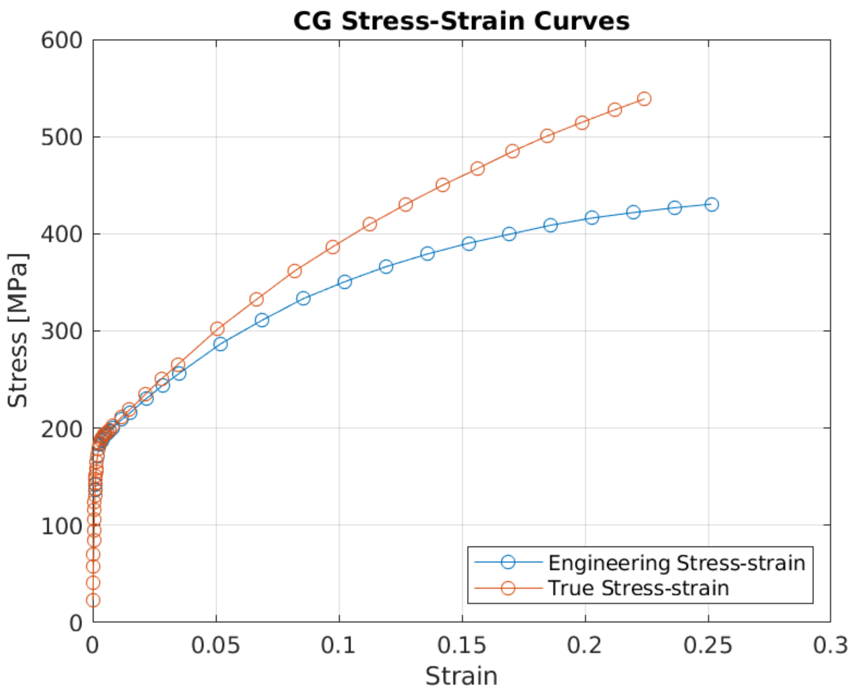
The CG material is assumed to behave as the Ni-200 material in [47], based on the material provided from [48], which specifies the ultimate tensile strength to be nominally between 380-520 MPa for engineering stress. The curve in Figure 13a and Figure 13b on the facing page shows the engineering stress-strain behaviour of the material. Initial plasticity is assumed from the picture and data in [48] to be 185 MPa. The highest stress value of the curve is presumed to be the ultimate tensile strength (UTS), and as a result, the whole curve can be used for discretisation and conversion to true stress, as seen in Figure 13c on the next page.



(a) The full uniaxial tension engineering stress-strain curve. Image adopted from [47].



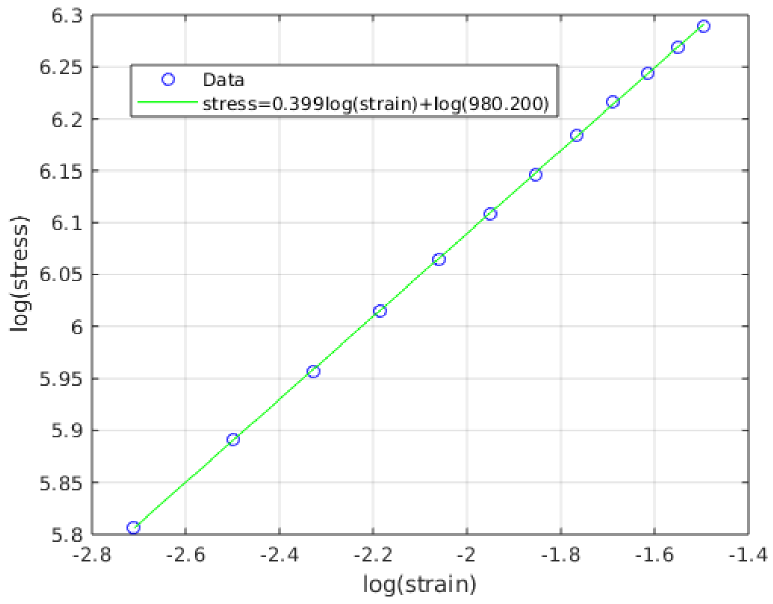
(b) A closer look at the initial phase of the uniaxial tension engineering stress-strain curve. Image adopted from [47].



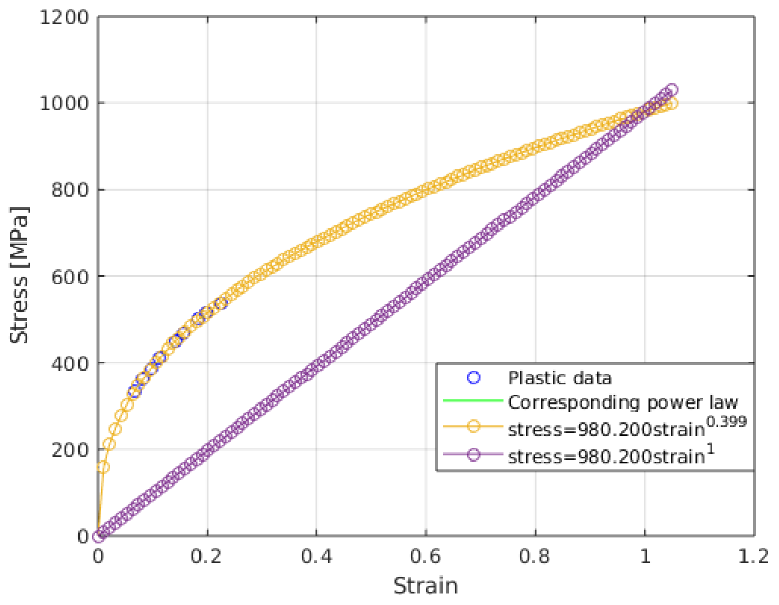
(c) The numerical discretisation of the CG engineering stress-strain curve and the converted true stress-strain curve.

Figure 13: The uniaxial tension stress-strain curves of Nickel 200 [47], and the corresponding discretisation and conversion to true stress-strain.

The linear fit of the log-log stress-strain curve is seen in Figure 14a on the following page, and the extrapolation using the complete power law as well as a linear approximation is seen in Figure 14b. Note that the stress values are in MPa, which transfers to the values of the inclination and y-intercept of the fitted function in both the CG and the UFG fits. The power law approximation yielded is $\sigma = 980 \times 10^6 \varepsilon^{0.399}$. The linear approximation is based on the power law being valid from no strain to max strain, which is not correct, as the initial strain is elastic. Due to this, the hardening modulus is taken to be 898e6, which is the same value as in [49].



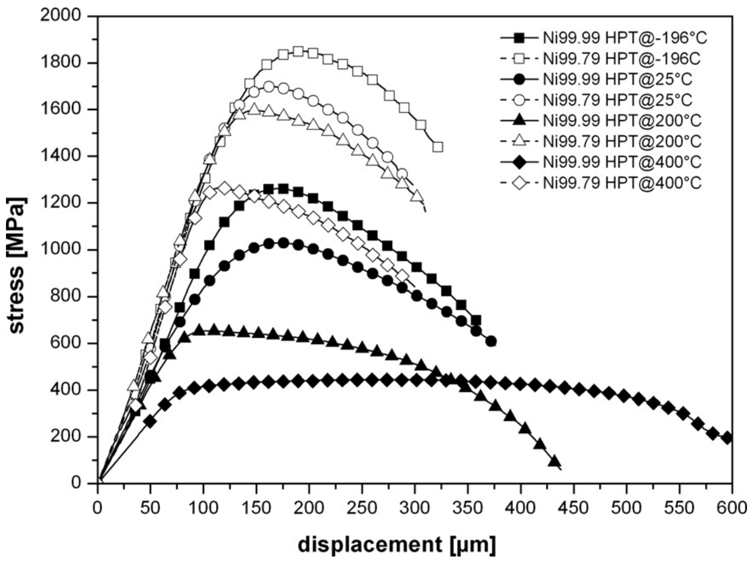
(a) The log-log curve fit of the plastic region of Nickel 200 .



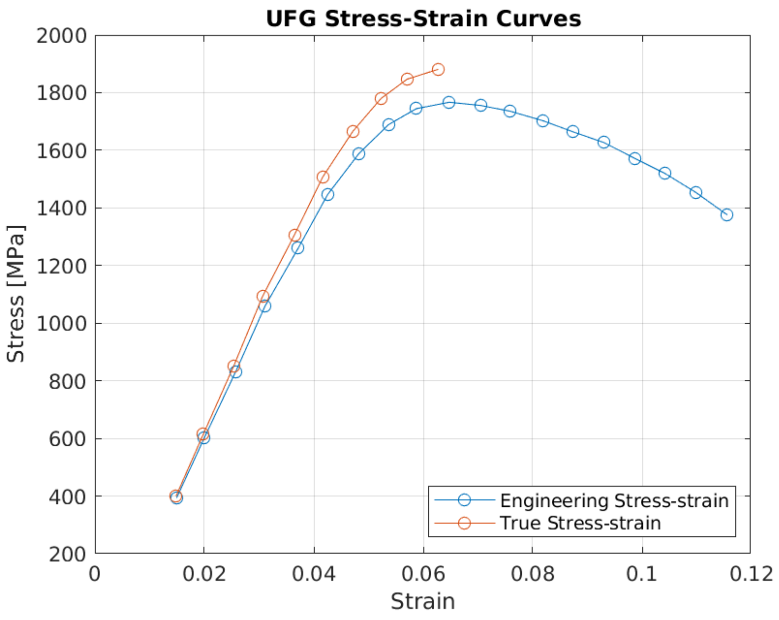
(b) The extrapolation of the power law yielded in 14a, with power law exponent (curved yellow line) as well as exponent equal to one (straight purple line). The plastic data is seen as blue dots along the curved yellow line.

Figure 14: The log-log curve fit and extrapolation of the plastic region of the CG material

3.3.2 Ultra-Fine Grain Material Model



(a) Nickel stress-strain curves for different temperatures and purity levels [50]. The data used is Ni99.97 HPT at 25 °C .



(b) The numerical discretisation of the CG engineering stress-strain curve and the converted true stress-strain curve.

Figure 15: The uniaxial tension stress-strain curves of high-pressure torsion nickel, and the corresponding discretisation and conversion to true stress-strain.

The UFG material model is based on data from an experiment using high-pressure torsion (HPT) to achieve small grain sizes in different purities of nickel at different temperatures [50]. UFG is assumed to correspond to 99.97 % pure nickel deformed at a temperature of 25 °C, as seen in Figure 15a. Initial plasticity is interpreted to

Table 2: A summary of the material parameter values initially chosen.

	UFG and CG
ρ	7850 $\mathrm{kg\,m^{-3}}$
E	210 GPa
ν	0.31

	CG Specific
σ_{yield}	185 MPa
K_H	898×10^6

	UFG Specific
σ_{yield}	1502 MPa
K_H	4356×10^6

occur at 1502 MPa based on Figure 15a on the preceding page. The displacement is converted to strain using the initial gauge length of 2.5 mm given in the article.

At the point of necking, analytical conversion to true stress is no longer valid since the strain becomes localised, resulting in three data points utterly defining the plastic behaviour used to fit a linear log-log curve fit in Figure 16a on the next page. Once again, it is noted that the stress values are given in MPa, and constants resulting from the curve-fit inherits this prefix. Based on the three available plastic stress data points, the power law approximation is $\sigma = 4356 \times 10^6 \varepsilon^{0.302}$ and the hardening modulus is taken to be 4356×10^6.

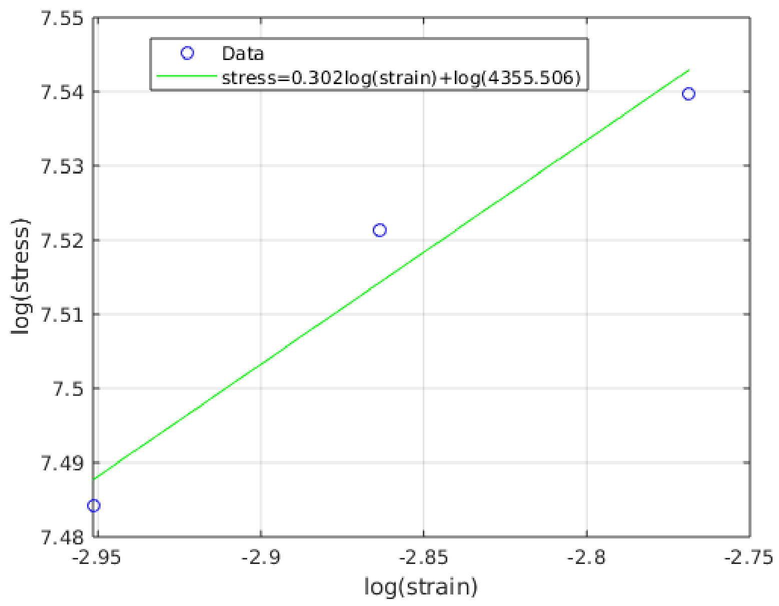
3.3.3 General Nickel Data

For both UFG and CG, the Poisson's ratio and elasticity modulus are assumed to be the same for both grains, and taken from [47]. This gives a Poisson's ratio of $\nu = 0.31$, and an elasticity module of $E = 210$ GPa. The density was faultily taken to be as 7580 $\mathrm{kg\,m^{-3}}$, which is the density of iron. The true density of nickel varies around 8900 $\mathrm{kg\,m^{-3}}$ [3]. As this error was noticed after all the simulations have been made, the error is mentioned here for clarity only.

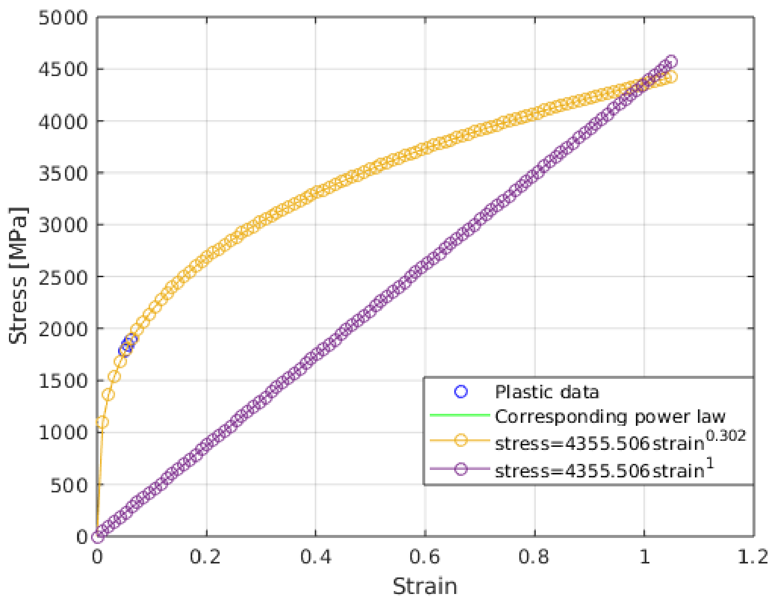
Summarised in Table 2 are the material parameters initially used for the project. The material data for the correspondence model also requires an hourglass coefficient $C_{hk}$to be specified. To avoid having to calibrate too many variables, it was chosen to consistently be exactly half of the interval specified in [21], giving $C_{hk} = 0.025$. Since the data for UFG and CG are both derived from tensile testing, the yield criterions are assumed to be valid for the uniaxial testing done in the Peridigm simulations as well, which should be checked according to [37].

3.3.4 Damage Model

For the damage model, an arbitrary initial value of the critical stretch was chosen to 0.002, as it was the stretch present in one of the Peridigm examples. Apart from the



(a) The log-log curve fit of the plastic region of the UFG material.



(b) The extrapolation of the power law yielded in 14a, with power law exponent (curved yellow line) as well as exponent equal to one (straight purple line). The plastic data is seen as blue dots along the curved yellow line.

Figure 16: The log-log curve fit and extrapolation of the plastic region of the UFG material

critical stretch value, pre-cracks and material flaws, as explained in the theory, are also used as tools for fracture actualisation experiments. Iteratively a value for the critical stretch, yielding similar behaviour as the material models, is sought.

3.4 Solvers

Both the NOXQuasiStatic and the Verlet solvers are used. The displacement rate in tensile test experiments are on an order of magnitude of $10^{-3}\ \mathrm{m\,s^{-1}}$, and neglectable compared to the force resulting from the stretch of the constitutive model as illustrated by the rough estimate below. For these reasons, the domain is initially believed to behave quasi-statically.

$$
\begin{aligned}
&F_{\ddot{\mathbf{u}}} = \rho V \ddot{\mathbf{u}} = 8000 \times 1e - 7 \times 10 = 0.02\mathrm{N} \\
&F_{\sigma} = E\varepsilon A = 210 \times 10^{9} \times 1 \times 10^{-3} \times 1 \times 10^{-5} = 2100\mathrm{N} \\
&\frac{F_{\ddot{\mathbf{u}}}}{F_{\sigma}} = 9.5 \times 10^{-6}
\end{aligned}
$$

The Verlet solver, as explained in the theory, yields very short time steps. Using the values specified in Table 2 on page 42, on a magnitude of 1×10^{-13}. To address this issue, both mass-scaling, numerical dampening, safety factor modification, and increase of displacement rate are tested.

3.5 Simulations

Simulations are carried out on three different platforms. Both a desktop and a laptop, as well as on the computer cluster Aurora on LUNARC, the centre for scientific and technical computing at Lund University. This allows the computations to be tested at different levels of parallelisation, and reduces time consumption as more simulations can be made in parallel. An example of a job submission script to Aurora can be seen in Appendix D.

3.6 Post Processing

The Exodus II file is loaded into Paraview [43], where various cell, point and field data distributions can be visualised. A python script is generated to automate the process of data exportation and image generation using the built-in trace function. The trace has to be re-made depending on which data is available, which domain is used, etc. The Paraview and Matlab post-processing scripts are all generated using a bash file which produces the final data. It is directory-independent and stand-alone, generating all necessary `.m` files, included in Appendix E for the full harmonic structured domain.

The corresponding data is exported as cell, field, and point data to `.csv` files, which are then read by one of the Matlab scripts generated, converting them to `.mat` files, see Appendix H. The data, on total force along the side boundaries and the elongation of the gauge region, calculated from the compute class parameters, as well as the the smallest cross-sectional area of the specimen, are used to plot the true and engineering stress-strain curves using scripts similar to Appendix I, also generated by the stand-alone post processing script. The true stress-strain curves are based on the force component in the axial direction, resulting in the correct stress-strain curve being calculated for axial tension testing.

The workflow explained in the sections above is illustrated in Figure 17 on the following page, where the flow of data is indicated as arrows between different actions, processes, configuration files, and programs. The figure is simplified to reduce the complexity of the image, and governing scripts such as job farms or job queues are not included.

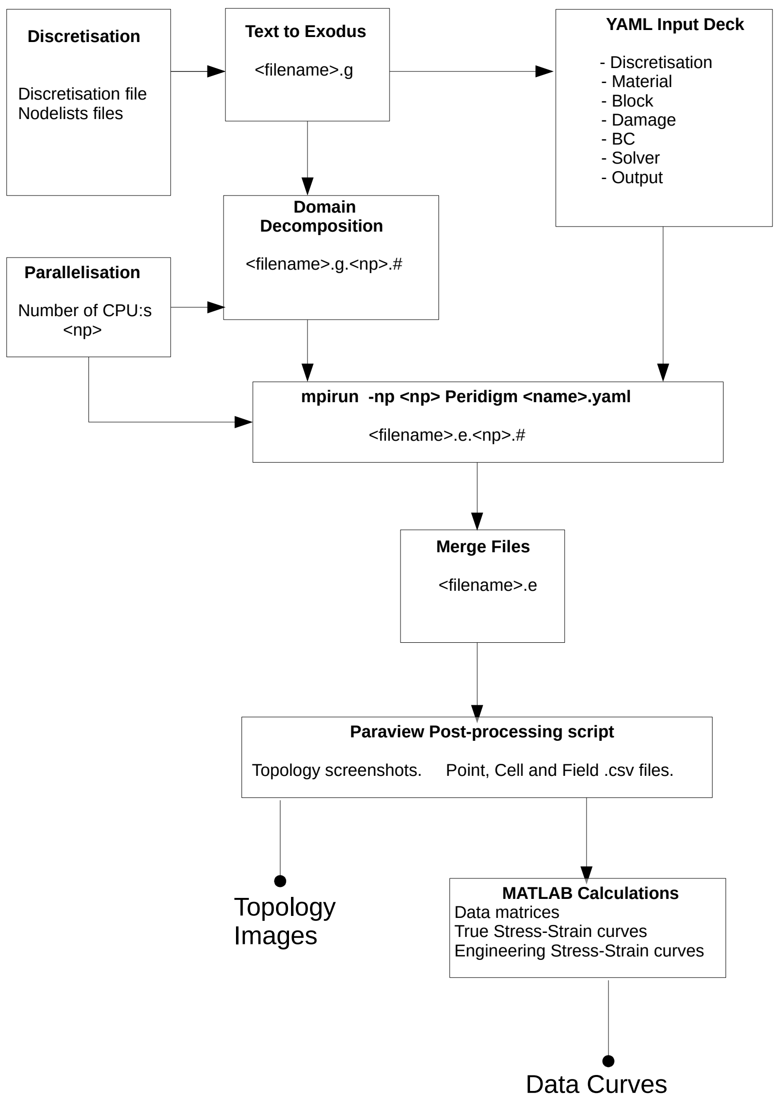


Figure 17: A simplified workflow scheme of the Peridigm usage, illustrating data flow between different scripts, files, and programs.

4 Results

The results are presented in separate tables for simulations regarding boundary conditions, material calibration, and the Verlet-solver simulations. The tables reference to figures and graphs, depending on what. A short comment is made for each simulation noting any interesting results or lack thereof. Below is a short explanation on the details of the different simulations.

Boundary Conditions: The material parameters are not compared with experimental data in these simulations, only the macroscopic deformation due to boundary conditions is investigated. A summary of the BC tests can be seen in Table 3 on the next page.

Calibration: Simulations are conducted on both the tensile test specimen domain, and the harmonic structured domain. The calibration simulations are summarised in Table 5 on page 49.

Verlet Simulations: The Verlet explicit time integration solver is used. Mass scaling does not result in larger time step increments, since mass scaling influence both density and bulk modulus, which determines the critical time step. Reduced total time combined with increased displacement velocities are simulated and presented in Table 4 on page 49.

Table 3: A summary of the different boundary conditions examined. The boundary condition node sets (NS) are denoted using the numbers in Figure 12 on page 37 and the colours used in Figure 11b on page 36. t is the time, and x, y, z are the nodal coordinates.

N.	BC: X	NS	BC: Y	NS	BC: Z	NS	Comment
1	0.0; t*x*3.6e-3 m	2; 3	0.0	7	0.0	Domain	No shrinking along NS 2 or NS 3. Displacement at NS 3 causes asymmetric behaviour and damage localisation compared to NS 2, Figure 34 on page 59.
2	0.0; t*x*3.6e-3 m	2; 3	0.0	6	0.0	Domain	No shrinking along NS 2, but at NS 3. Displacement at NS 3. Asymmetric behaviour in whole domain, Figure 35 on page 59.
3	t*x*3.6e-3 m	2, 3	-	-	0.0	Domain	No shrinking along NS 2 or NS 3, Figure 36 on page 60. Damage localisation in several places.
4	t*x*3.6e-3 m	2, 3	0.0	Blue	0.0	Red	No shrinking along the red or blue NS pictured in Figure 11b on page 36. Full domain capable of cross-sectional compression due to elongation in x. Simulation made using CG material only. The lack of asymmetric behaviour can be seen in Figure 37 on page 60.

Table 4: The two Verlet time integration simulations made by increasing speed and reducing total time only. No mass scaling made. Solver safety factor is 0.01 and numerical damping is 160.0e3 for both simulations. Both are made using the non-correspondence material and its parameters, resulting in the same time step $\Delta t = 1.78e - 13$ s. The critical stretches are $s_{c,UFG} = 0.10$ and $s_{c,CG} = 0.22$

N.	v_0	Final Time	Comment
1	450 m/s	22.77e-7 s	Initial velocity waves propagates through material and causes fracture as they meet, figures 48, 49, 50, 51, and 52. Vibrations progress throughout the material. Fracture occurs along grain boundaries as seen in figure 54. Following initial central fracture, outer ends fracture as well as seen in figures 51 52 and 53, where the nodes preventing rigid body motion in z are noted to be damaged as well.
2	45 m/s	27.76e-7 s	Initial velocity waves does not cause fracture upon meeting. Propagation of the waves illustrated in figures 55, 56, 57, 58, 59, 60, and 61.

Table 5: A summary of the calibration simulations. An interface aware (IA) damage model indicates a correspondence material model. Units are converted from CSG to SI. All simulations use NOXQuasiStatic solver unless stated otherwise.

N.	**Material - $\mathbf{K_H}$**	**Dmg - $\mathbf{s_c}$**	**Δt**	**Comment**
1	CG - 898e6	IA - 0.002	0.033 s	No fracture. No necking. Stress strain figure 18. CPU Core decomposition differences: figures 38 and 39.
2	CG - 898e6	IA - 0.0005	0.023 s	No fracture. No necking. Figure 19.
3	CG - 898e6	IA - 0.0001	0.023 s	Strain localisation outside gauge region. Figures 22 and 40.
4	CG - 898e6	IA - 0.0002	0.020 s	Fracture occurs. No necking. Figure 20.
5	CG - 898e6	IA - 0.0002	0.010 s	No necking . No necking. Figure 21.
6	CG - 2100e6	IA - 0.0002	0.015 s	Modified hardening. No fracture. No necking. Figure 23.
7	CG - 2100e6	IA - 0.00005	0.012 s	Fracture occurs. No necking. Figures 24 and 44.
8	CG - 2100e6	IA - 0.00005	0.010 s	No fracture. No necking. Figure 25.
9	CG - 2100e6	IA - 0.001	0.010 s	Flaw: $C_{f,m} = 0.3$, $C_{f,s} = 0.05$ at origo. No fracture. No necking. Figure 26.
10	CG - 2100e6	IA - 0.001	0.010 s	Flaw: $C_{f,m} = 0.3$, $C_{f,s} = 0.4$ at origo. No fracture. No necking. Noticeable gradual yield. Figure 27.

11	CG - 2100e6	IA - 0.0001	0.013 s	Bond filter: One corner. No fracture. No necking. Figures 28 and 41.
12	CG - 2100e6	IA - 0.0001	0.012 s	Bond filter: Centre, size is half of full cross-sectional area. No fracture. No necking. Figures 29 and 42.
13	CG - 2100e6	IA - 0.0001	0.012 s	Bond filter: Centre, larger than cross-sectional area. Domain split in two. Figure 43.
14	UFG - 898e6	IA - 0.00425	0.033 s	Misconfigured UFG material, using the hardening coefficient of the CG material. Necking occurs. No Fracture. Figures 46 and 45.
15	UFG - 4356e6	IA - 0.02	0.1 s	Necking occurs. No Fracture. Figure 33 .
16	CG - 2100e6	IA - 0.01	0.004 s	HS Domain, Volume: 6.1×10^{-18} m^3. Figure 30.
17	CG - 2100e6	IA - 0.01	0.004 s	HS Domain, Volume: 6.1×10^{-2} m^3. Figure 31.
18	CG - 2100e6	IA - 0.01	-	Verlet solver, Interface Aware solver produces error message.
19	Elastic CG - -	Crit. Stretch - 0.02	6e-8 s	Verlet solver, correspondence elastic material, Interface Aware solver produces error.
20	Elastic CG - -	Crit. Stretch - 0.02	6e-8 s	Verlet solver, non-correspondence elastic material. Fracture occurs. No necking possible due to no plasticity modelling. Displacement velocity of y*500 m/s. Figure 47.

4.1 Graphs

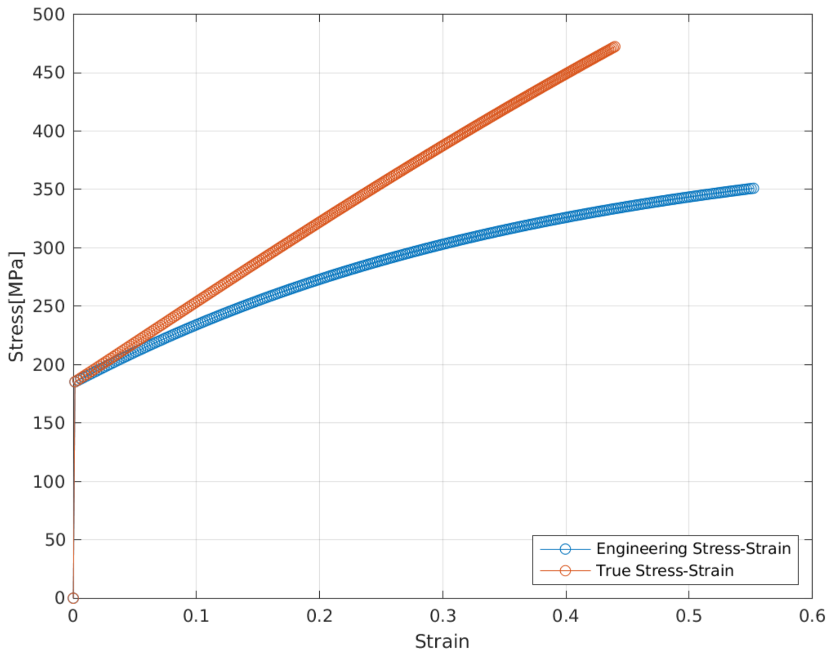


Figure 18: CG with $K_H = 898 \times 10^6$, $s_c = 0.002$, and $\Delta t = 0.033$ s.

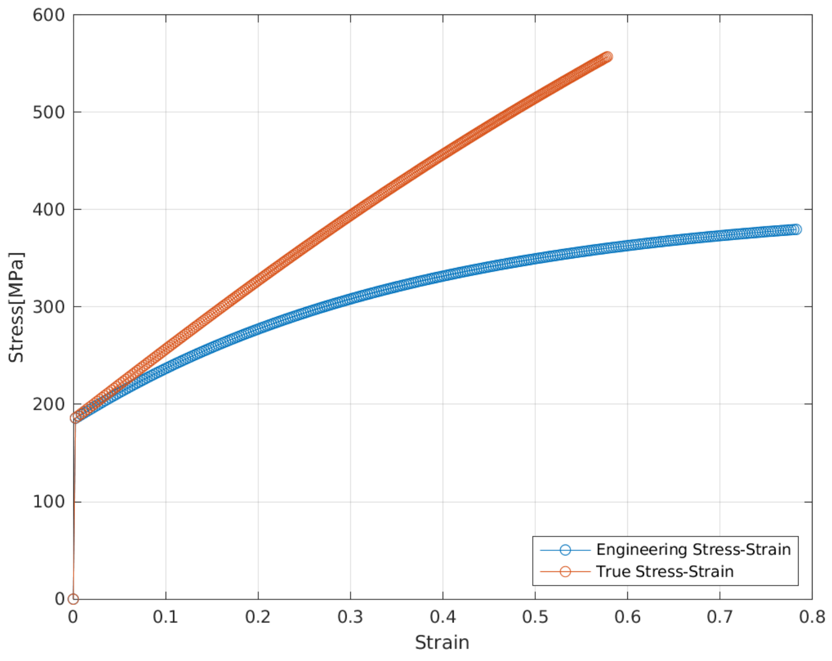


Figure 19: CG with $K_H = 898 \times 10^6$, $s_c = 0.0005$, and $\Delta t = 0.023$ s.

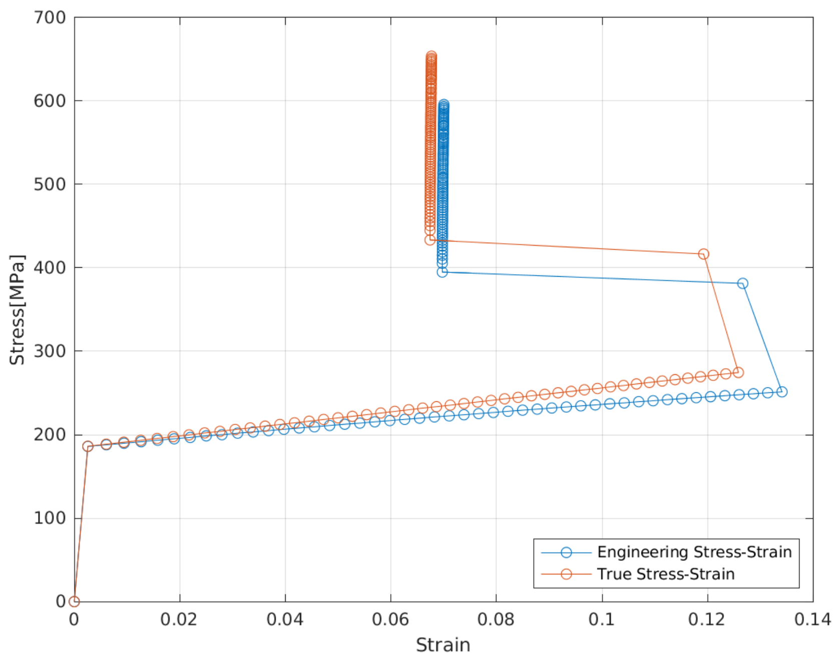


Figure 20: CG with $K_H = 898 \times 10^6$, $s_c = 0.0002$, and $\Delta t = 0.02$ s. Fracture occurs abruptly.

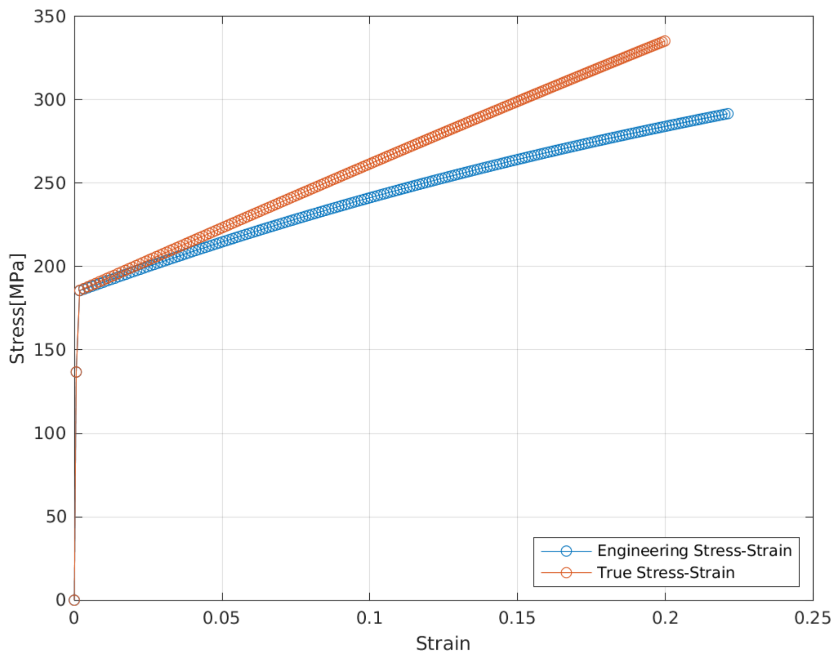


Figure 21: CG with $K_H = 898 \times 10^6$, $s_c = 0.0002$, and $\Delta t = 0.01$ s.

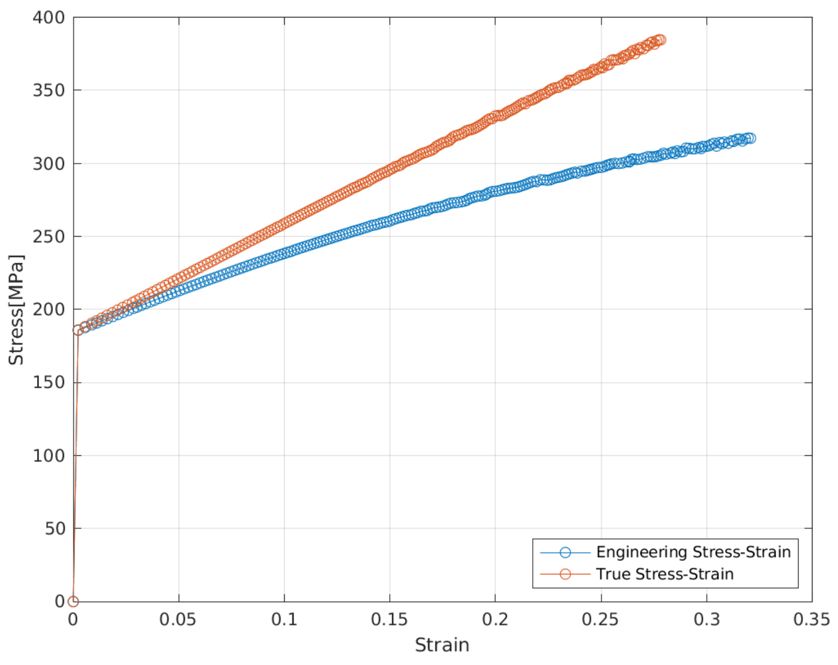


Figure 22: CG with $K_H = 898 \times 10^6$, $s_c = 0.0001$, and $\Delta t = 0.023$ s. Localised strain occurs outside the gauge region. Oscillations occur.

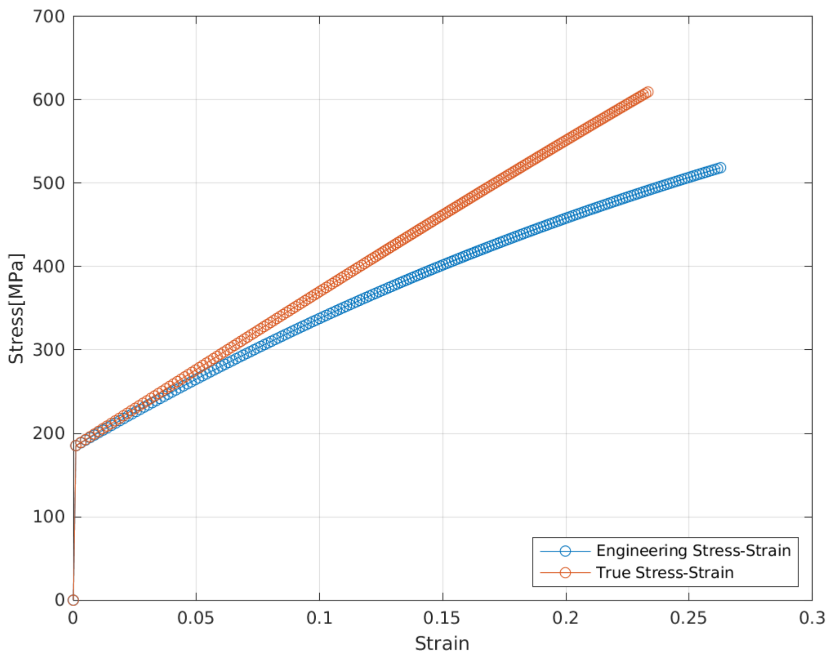


Figure 23: CG with $K_H = 2100 \times 10^6$, $s_c = 0.0002$, and $\Delta t = 0.015$ s.

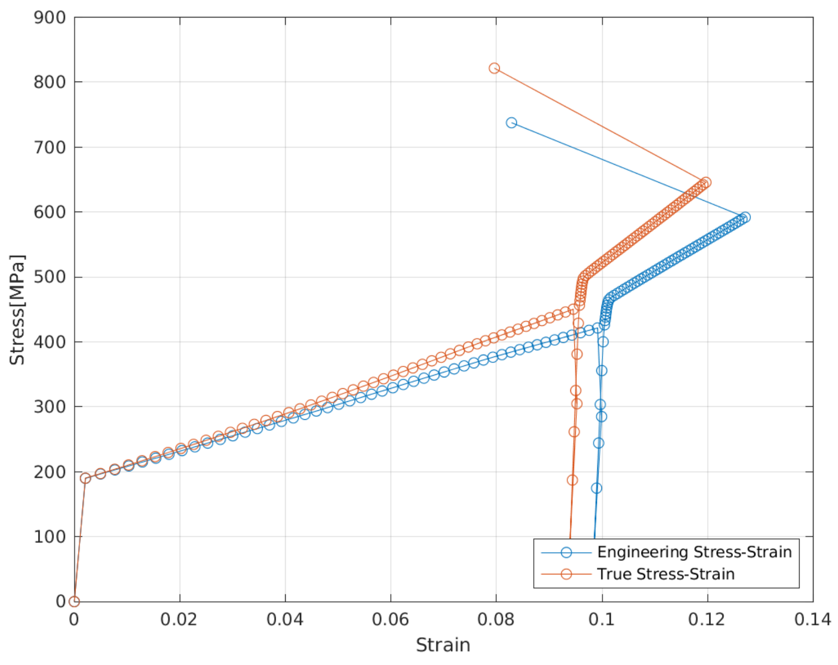


Figure 24: CG with $K_H = 2100 \times 10^6$, $s_c = 0.00005$, and $\Delta t = 0.012$ s. Fracture occurs abruptly.

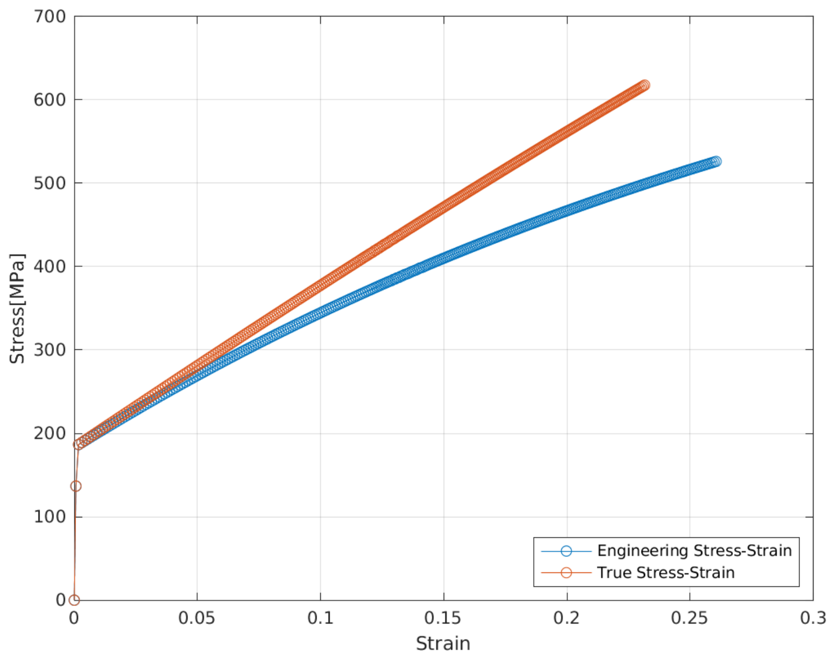


Figure 25: CG with $K_H = 2100 \times 10^6$, $s_c = 0.00005$, and $\Delta t = 0.010$ s.

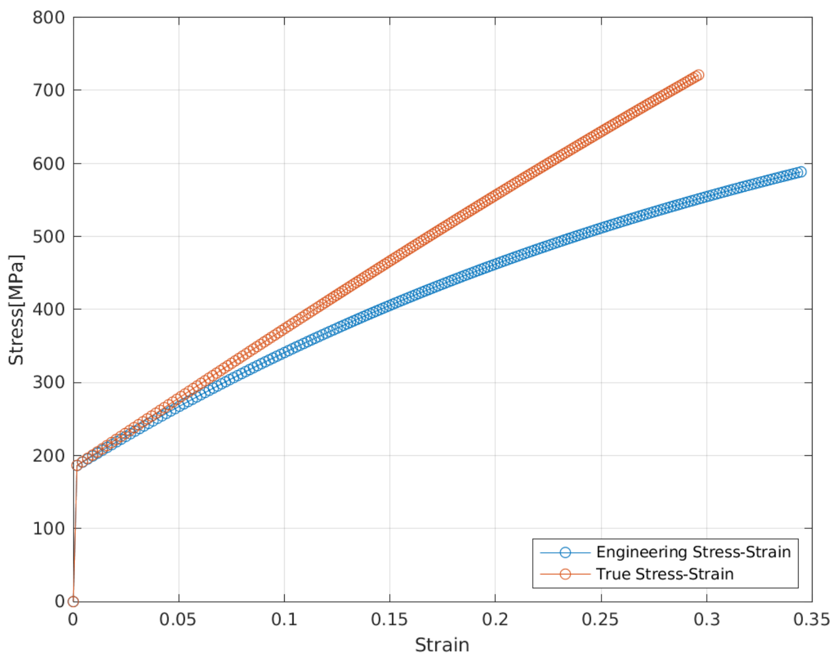


Figure 26: CG with $K_H = 2100 \times 10^6$, $s_c = 0.001$, and $\Delta t = 0.010$ s. Flaw: $C_{f,m} = 0.3$, $C_{f,s} = 0.05$ at origo.

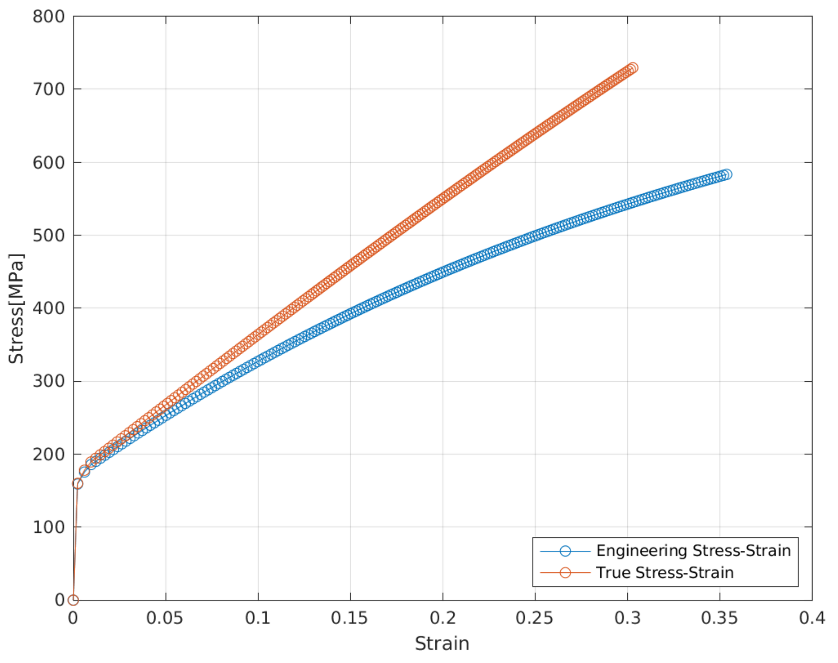


Figure 27: CG with $K_H = 2100 \times 10^6$, $s_c = 0.001$, and $\Delta t = 0.010$ s. Flaw: $C_{f,m} = 0.3$, $C_{f,s} = 0.4$ at origo. Initial yielding gradual.

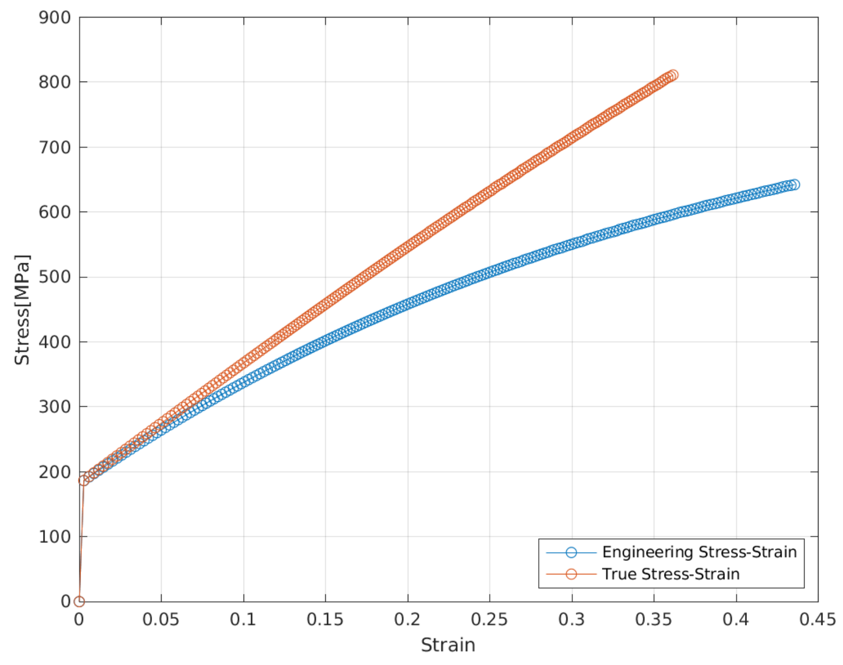


Figure 28: CG with $K_H = 2100 \times 10^6$, $s_c = 0.0001$, and $\Delta t = 0.013$ s. Bond filter at one corner. No further crack propagation. No necking.

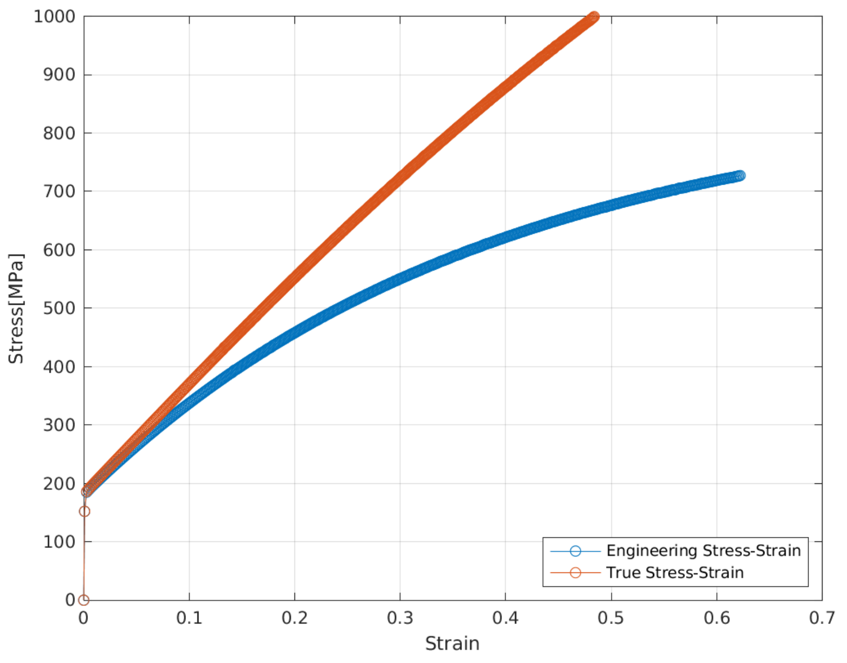


Figure 29: CG with $K_H = 2100 \times 10^6$, $s_c = 0.0001$, and $\Delta t = 0.012$ s. Bond filter at centre of domain. No further crack propagation. No necking.

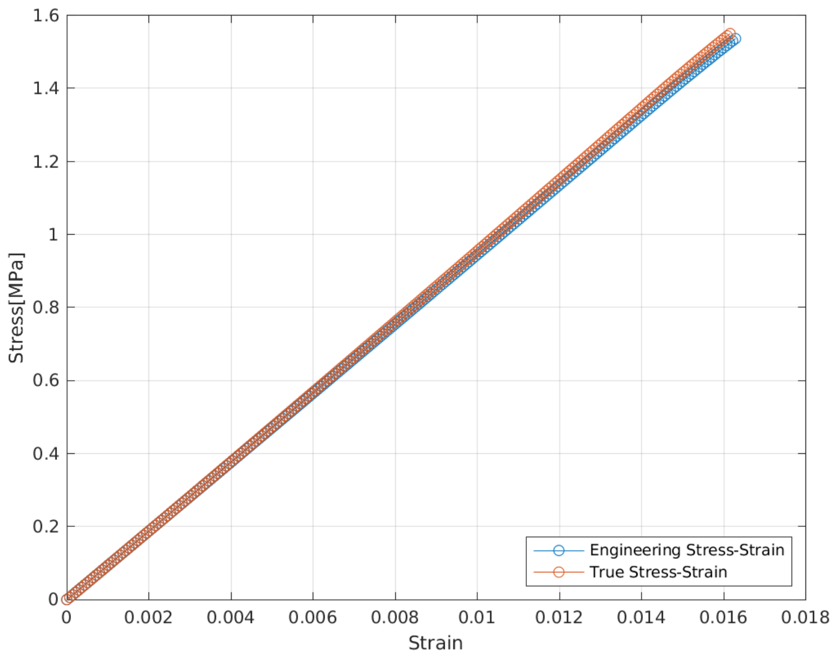


Figure 30: Harmonic structured CG with $K_H = 2100 \times 10^6$, $s_c = 0.01$, and $\Delta t = 0.004$ s. Small volume increases stress magnitude but does not impact other aspects.

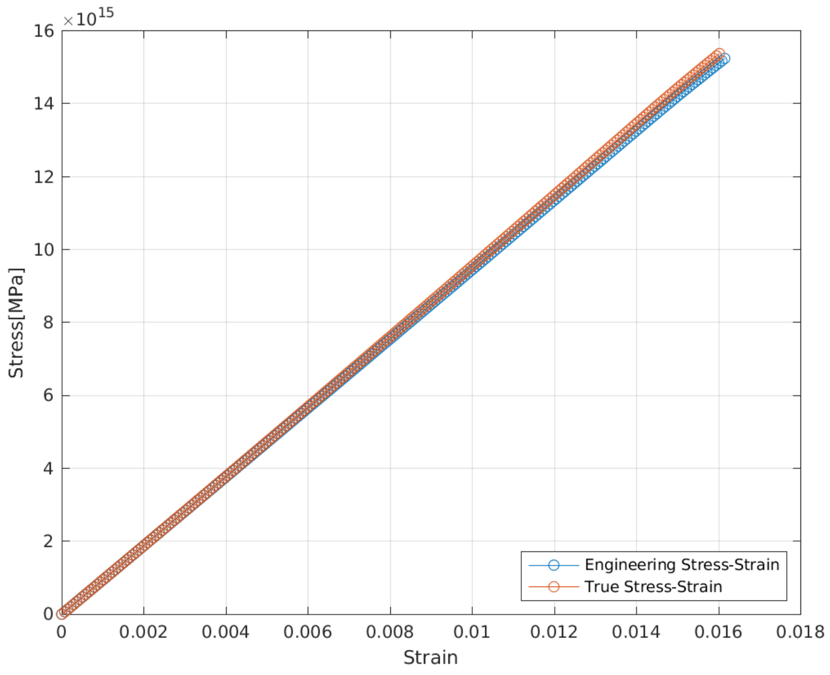


Figure 31: Harmonic structured CG with $K_H = 2100 \times 10^6$, $s_c = 0.01$, and $\Delta t = 0.004$ s. Large volume increases stress magnitude but does not impact other aspects.

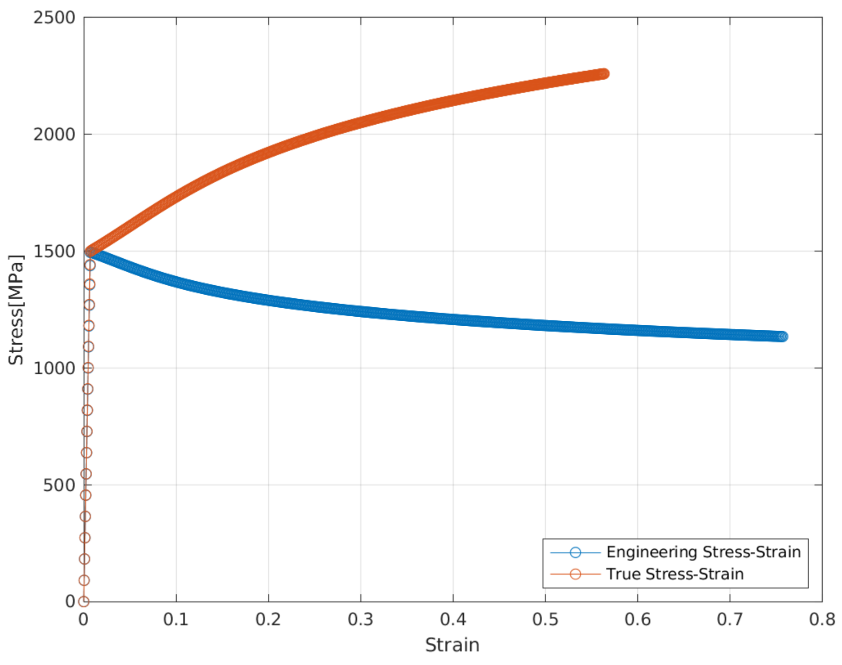


Figure 32: UFG with $K_H = 898 \times 10^6$, $s_c = 0.00425$, and $\Delta t = 0.033$ s. Erroneous simulation made using the hardening coefficient of the CG material. Necking occurs instantly after yielding as seen in the engineering stress strain curve.

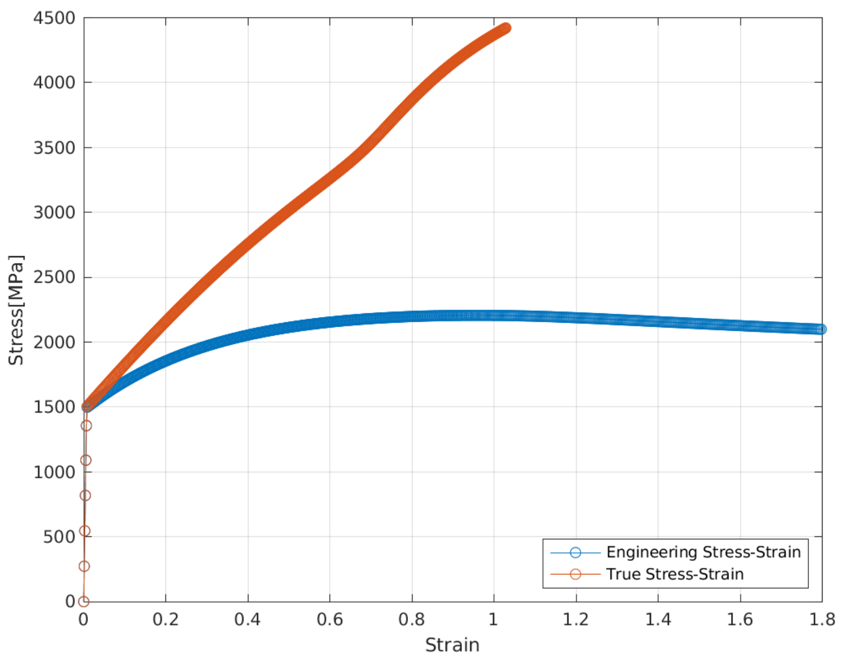


Figure 33: UFG with $K_H = 4356 \times 10^6$, $s_c = 0.02$, and $\Delta t = 6 \times 10^{-8}$ s. Necking occurs as seen in the peak of the engineering stress-strain curve.

4.2 Images

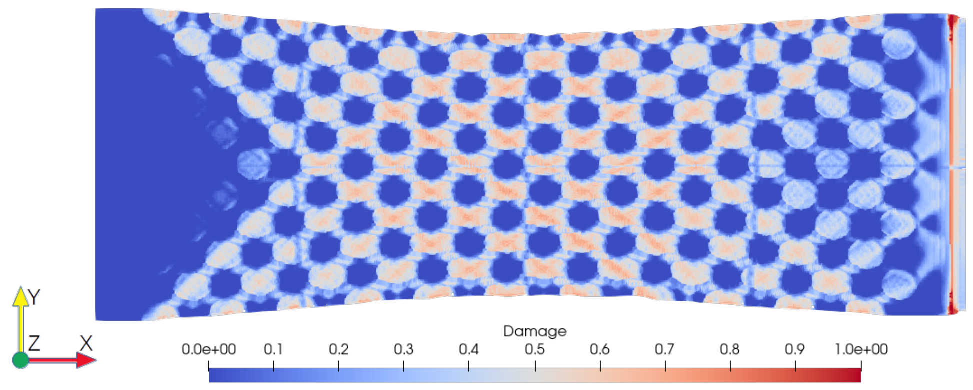


Figure 34: NS 3 displaced at x*3.6e-3 m/s. Domain locked in z and NS 7 locked in y. Asymmetric behaviour in x-direction.

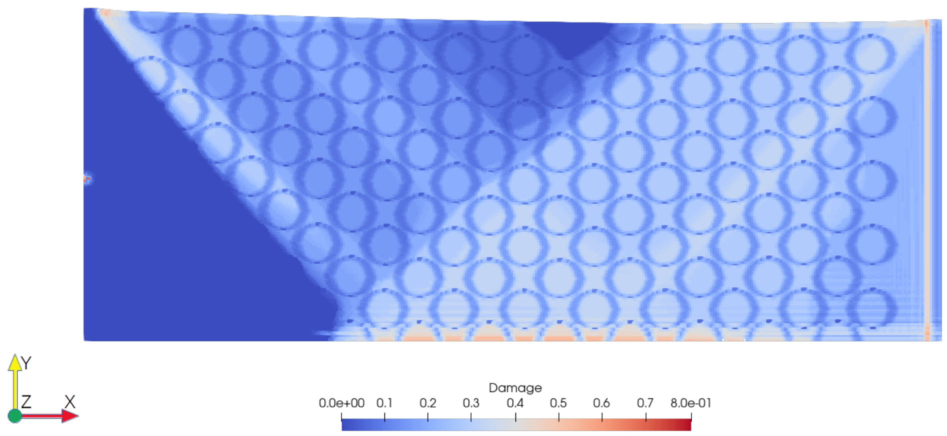


Figure 35: NS 3 displaced at x*3.6e-3 m/s. Domain locked in z and NS 6 locked in y. Asymmetric behaviour in x- and y-direction.

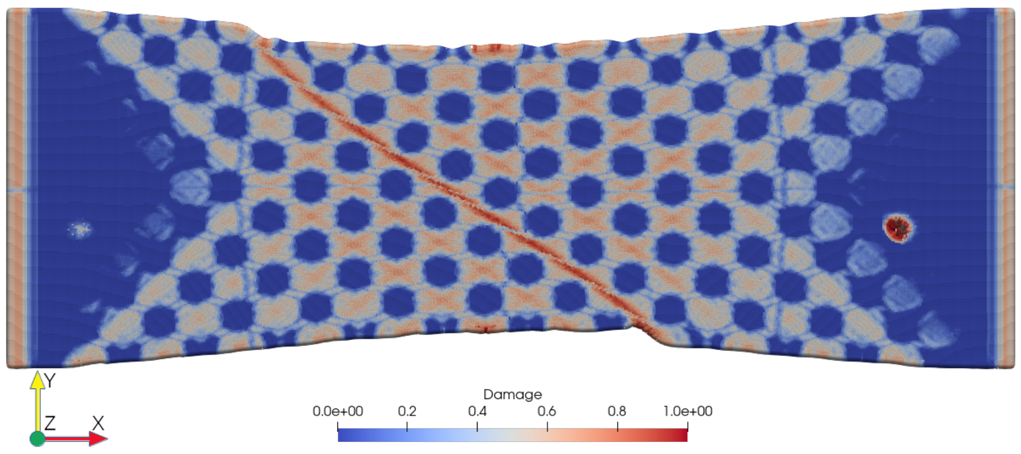


Figure 36: NS 2 and NS 3 displaced at x*3.6e-3 m/s. Domain locked in z and no nodes locked in y. Localised damage across the domain at an angle with displacement direction.

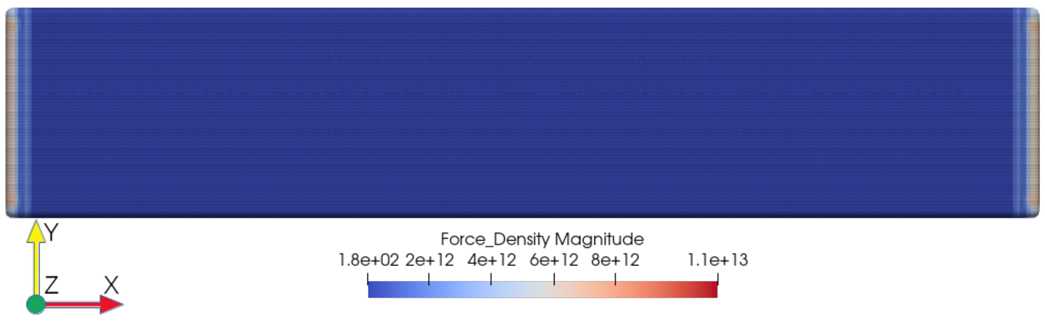


Figure 37: Single-row BC preventing movement in x and z as specified in 11b. Homogeneous displacement of the whole domain. Force is symmetrical. Material is CG only.

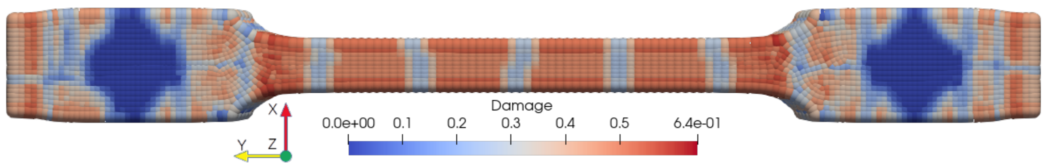


Figure 38: CG with $K_H = 898 \times 10^6$, $s_c = 0.002$, and $\Delta t = 0.033$ s. Domain decomposed to 20 CPU cores. Borders with reduced damage can be seen as light blue lines, which is due to a failure of the CPU cores to communicate damage between them.

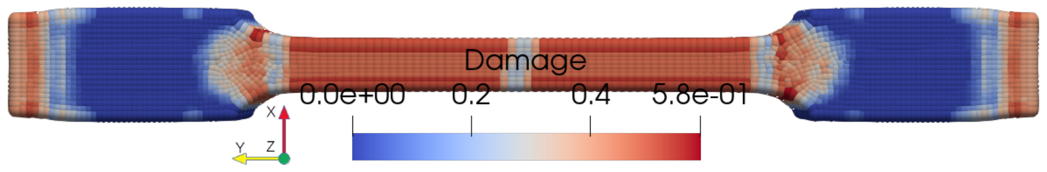


Figure 39: CG with $K_H = 898 \times 10^6$, $s_c = 0.002$, and $\Delta t = 0.033$ s. Domain decomposed to 8 CPU cores. Borders with reduced damage can be seen as light blue lines, which is due to a failure of the CPU cores to communicate damage between them.

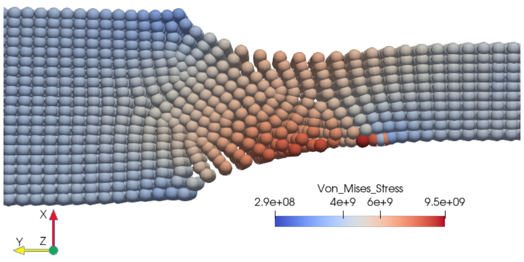


Figure 40: CG with $K_H = 898 \times 10^6$, $s_c = 0.0001$, and $\Delta t = 0.023$ s. Localised strain occurs outside the gauge region. Asymmetric stress and displacement on specimen shoulder.

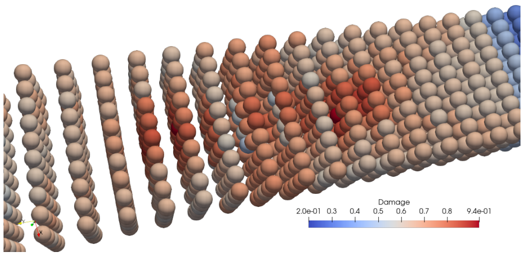


Figure 42: CG with $K_H = 2100 \times 10^6$, $s_c = 0.0001$, and $\Delta t = 0.012$ s. Bond filter at centre of domain. Lower damage at bond filter. Yielding occurs further from origo.

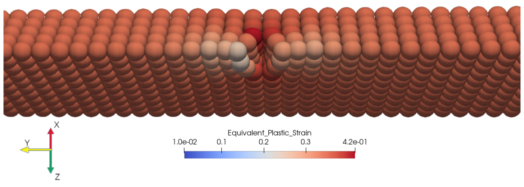


Figure 41: CG with $K_H = 2100 \times 10^6$, $s_c = 0.0001$, and $\Delta t = 0.013$ s. Bond filter at one corner. Plastic strain is lower around the bond filter.

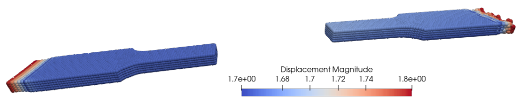


Figure 43: CG with $K_H =$, $s_c = 0.0$, and $\Delta t = 0.$ s. Anomalies at right end of specimen.

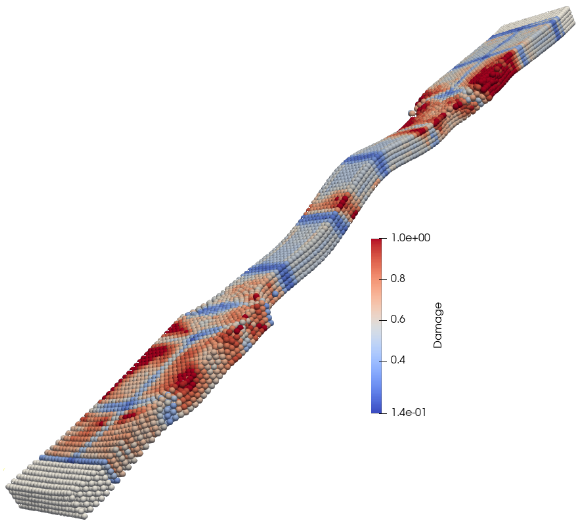


Figure 44: CG with $K_H = 2100 \times 10^6$, $s_c = 0.00005$, and $\Delta t = 0.01167$ s. Fracture occurs abruptly. Three-dimensional deformation.

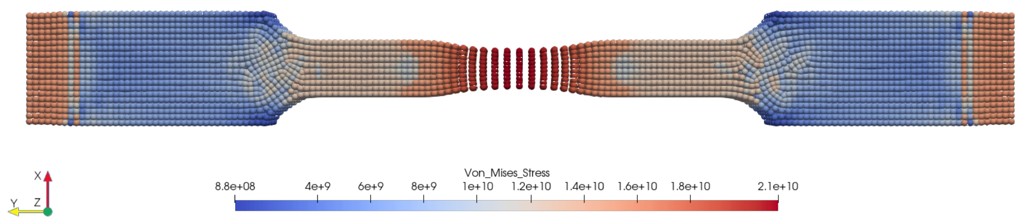


Figure 45: UFG with $K_H = 898 \times 10^6$, $s_c = 0.00425$, and $\Delta t = 0.033$ s. Erroneous simulation made with the hardening coefficient of CG material. Stress localisation occurs instantaneously following yielding.

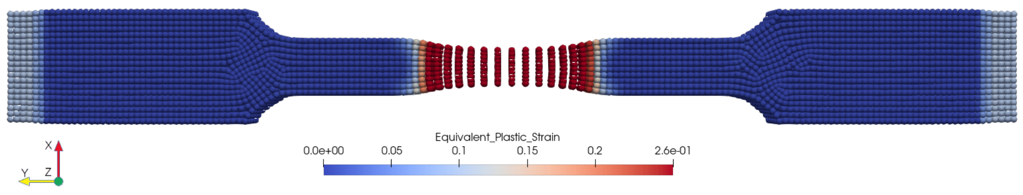


Figure 46: UFG with $K_H = 898 \times 10^6$, $s_c = 0.00425$, and $\Delta t = 0.033$ s. Erroneous simulation made with the hardening coefficient of CG material. Plastic deformation occurs only in the neck region.

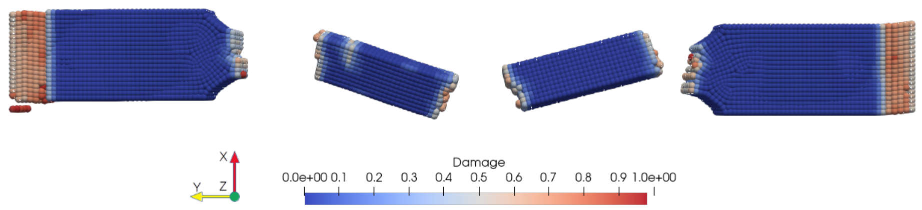


Figure 47: Elastic CG, non-correspondence $s_c = 0.002$, and $\Delta t = 6 \times 10^{-8}$ s. Anomalies at right end of specimen.

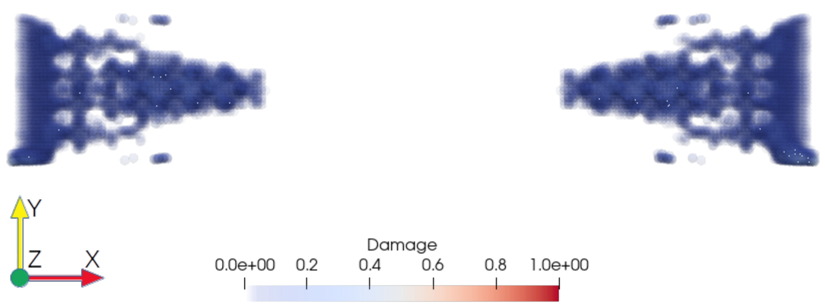


Figure 48: Fracture propagation: $t = 3.9 \times 10^{-7}$ s. Non-damaged and fully damaged nodes are hidden.

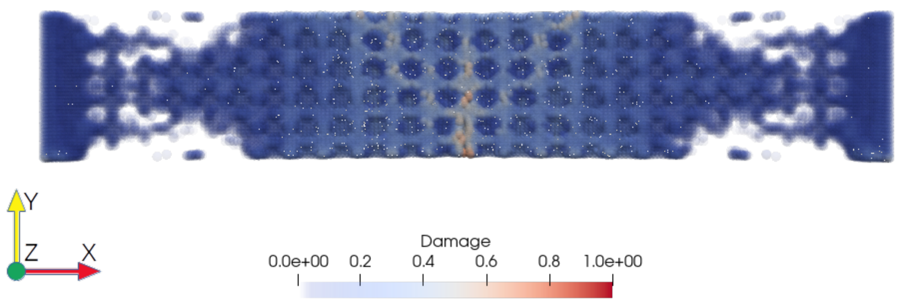


Figure 49: Fracture propagation: $t = 7.1 \times 10^{-7}$ s. Non-damaged and fully damaged nodes are hidden.

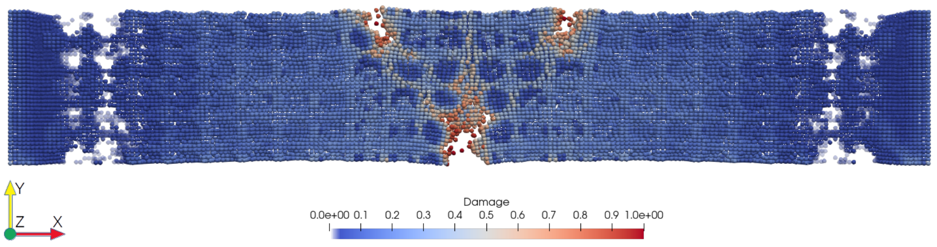


Figure 50: Fracture propagation: $t = 8.2 \times 10^{-7}$ s. Non-damaged and fully damaged nodes are hidden.

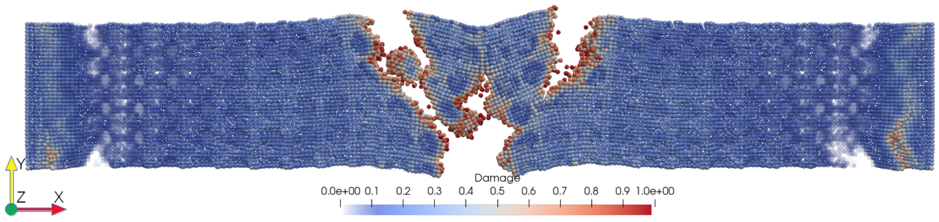


Figure 51: Fracture propagation: $t = 9.6 \times 10^{-7}$ s. Non-damaged and fully damaged nodes are hidden.

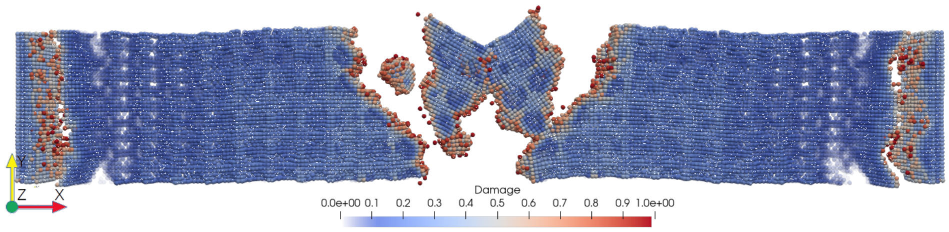


Figure 52: Fracture propagation: $t = 11.0 \times 10^{-7}$ s. Non-damaged and fully damaged nodes are hidden.

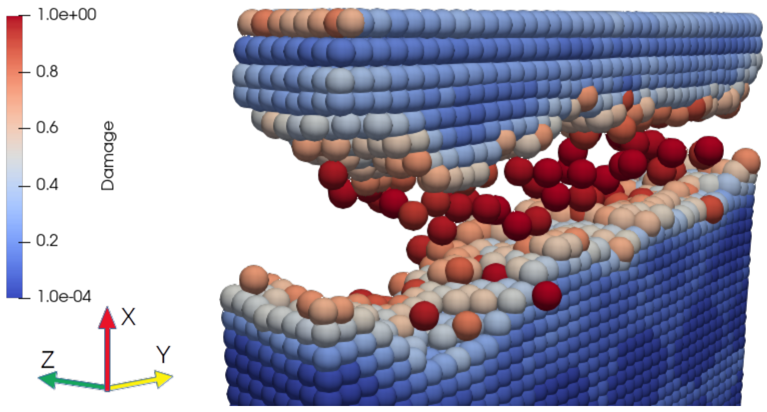


Figure 53: Fracture propagation on the corner: $t = 11.4 \times 10^{-7}$ s.

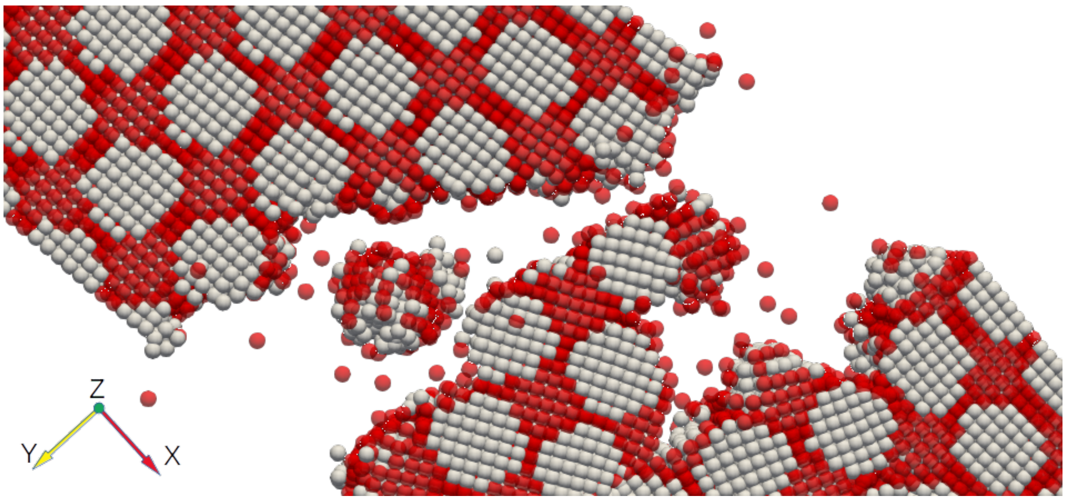


Figure 54: Close up of the fracture region at $t = 10.3 \times 10^{-7}$ s. Grain materials highlighted in red and white. Separation seems to occur along grain boundaries.

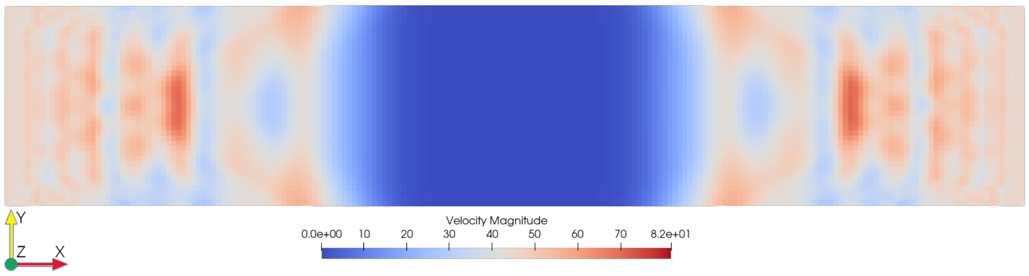


Figure 55: Initial velocity of the nodes at $t = 2.9 \times 10^{-7}$ s. Wave propagating towards centre.

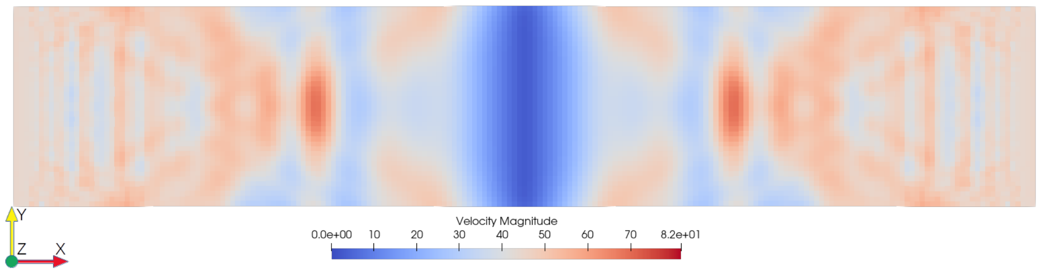


Figure 56: Velocity of the nodes at $t = 4.0 \times 10^{-7}$ s. Slightly before waves interference.

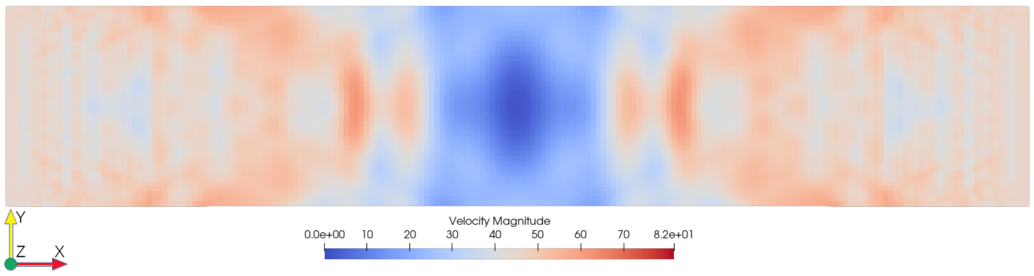


Figure 57: Velocity of the nodes at $t = 4.8 \times 10^{-7}$ s. Destructive interference of the velocity waves.

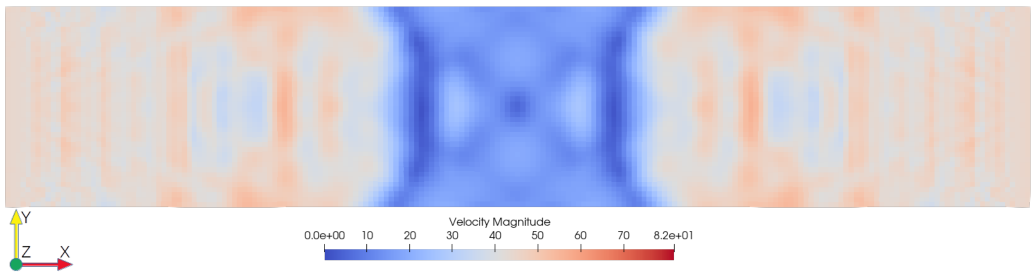


Figure 58: Velocity of the nodes at $t = 5.6 \times 10^{-7}$ s. Waves propagating past each other.

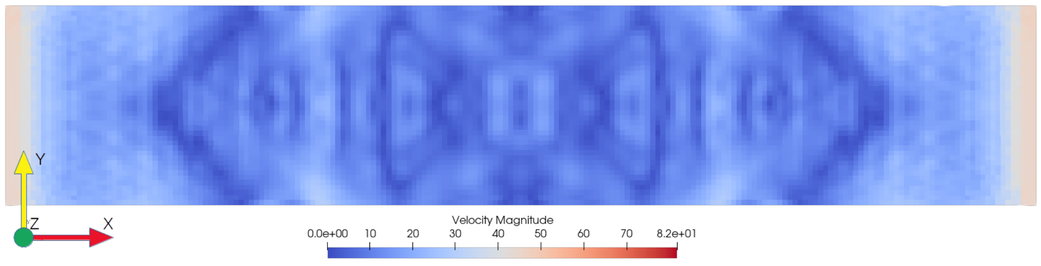


Figure 59: Velocity of the nodes at $t = 8.8 \times 10^{-7}$ s. Waves reach the opposite end.

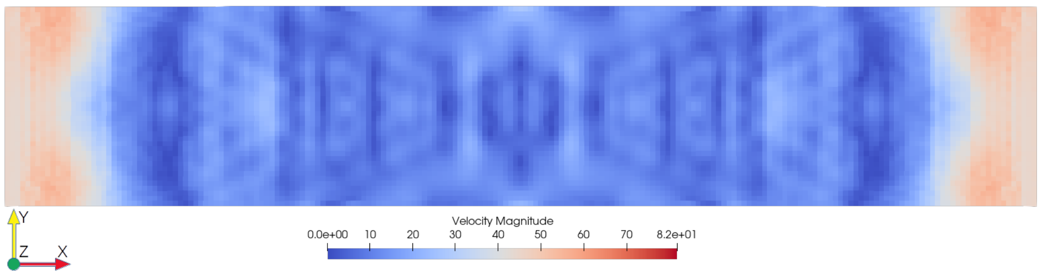


Figure 60: Velocity of the nodes at $t = 9.6 \times 10^{-7}$ s. Reflection of end wall.

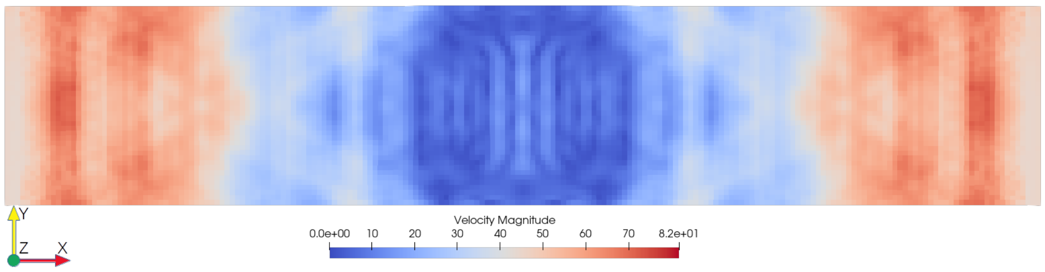


Figure 61: Velocity of the nodes at $t = 12.0 \times 10^{-7}$ s. Destructive and constructive interference present.

5 Discussion

No satisfying results on fracture in static tensile testing of harmonic structured materials are achieved. Material calibration of neither UFG nor CG material models were successful. However, successful fracture simulations were made and a general framework including installation recommendations, automation scripts, and post-processing used with Peridigm has been developed.

5.1 Necking

Necking is only seen to occur in the UFG calibration simulation, illustrated in figure 32 and Figure 33 on page 58. The necking in the former case occurs instantaneously upon first yield of the material. As a result, the plastic engineering stress never increases as the cross-sectional area is reduced upon initial plasticity. This is confirmed by Figure 46 on page 63, where the equivalent plastic strain is seen to only occur in the necking region as well as on the boundary blocks. In the latter necking case, a substantial plastic strain is required before a minor necking manifests.

As the difference in hardening modulus between the two is several times larger than the smaller hardening modulus, this difference could be a reason for the different UTS points. The simulation made in Figure 32 however, shows that necking clearly can be achieved without significant displacement. Why simulation 1 in Table 5 on page 49 does not result in a similar yield despite having the same time increment, and a lower critical stretch, could then be attributed to the lower yield stress, as neither the 8 core or 20 core decompositions result in necking, the 8 core decomposition being the same decomposition as the UFG material is simulated on.

There could be several reasons for the lack of necking in the CG calibration simulations. The largest problem is that none of them are elongated as far as the UFG simulation in Figure 33 on page 58. Not running a simulation until necking occurs was a conscious decision motivated by restricted computational resources initially in the project, before Peridigm was installed on the cluster. If UTS did not occur around the same strain as for the experimental data, it would be assumed to be a faulty calibration and not run further. This would allow more simulations to be run on the same amount of resources, yielding a properly calibrated material. The issue with this approach is that since the sought behaviour did not occur, the impact of the parameters on the UTS point remains unknown. In hindsight, a better approach would have been to elongate at least until UTS, irrespective of whether it occurs around the expected strain or not. In that way, the simulation parameters' effect on the UTS point could perhaps have been established more clearly.

5.2 Fracture and Solvers

Evident from the results of the material calibration as well as the theory, quasi-static solvers do not model fracture as expected initially in the project. This is mainly due to the fact that fracture is a dynamic and relatively fast phenomenon compared to the slow axial displacement. As such, the neglect of velocities is not a valid assumption when modelling fracture. The explicit solver on the other hand, models fracture in a realistic way. Both fracture initiation and propagation between nodes occur in a physically sound way. As seen in figures 49 to 52, the fracture paths are noticeable before the full crack occurs. The direction of fracture propagation is consequent once it arises. This conforms to the theory specifying that crack propagation occurs progressively due to the strain density vector states removing bonds as they reach the critical stretch. The behaviour is seen both for the three central cracks, and the two end-side cracks, which both propagate continuously upwards. One-node, free-flying particles are created during fracture as well as seen in Figure 53 on page 65. When fracture occurs in the quasistatic solver, the nodes not subjected to any external force would remain still as soon as they disconnected from the domain body.

One error which manifested itself early in this project, and the reason the Verlet solver was disregarded initially, is that it is currently incompatible with the Interface Aware damage model in Peridigm (at least in the version installed using the installation script in Appendix A). This is the reason there is an elastic correspondence model simulated in the Verlet solver, to show that it is not caused by an error in the correspondence material model. There could of course be other errors in the material model as well, given the messaged mentioned during Peridigm installation in Appendix A. For example, the fact that different domain decompositions yield different damage results in Figure 38 and Figure 39 on page 61. According to the answer to the issue posted to the Peridigm GitHub, (`https://github.com/Peridigm/Peridigm/issues/119`), the processors are not communicating the damage properly between them. The cause for this remains unknown, and until this issue is resolved, the interface aware damage model, and as a result all correspondence models, are of no use.

The implicit solver was not used at all, mainly due to time limitations following the late realisation that quasistatic solvers were not suitable. It might also be more suitable compared to the explicit time integration given the larger load steps allowed. An issue might arise with this approach if fracture propagation turns out to be larger than the horizon for a load step. In that case, the solution will probably diverge. Another issue might be that the amount of iterations required for each load step results in long simulation times as well.

To mitigate both of these issues, the combination of different solvers should be utilised. For tensile testing, the quasistatic solver or the implicit solver could be used initially up to a time when fracture is expected to occur, where the switch to the explicit solver takes place. The switch point could be decided by running the simulation using purely a quasistatic or implicit solver, preferably the implicit solver, since fracture seems to be initiated at different times depending on load step size as seen in the pairs encompassing figure 20 and Figure 21 on page 52, as well as Figure 24 and Figure 25 on page 54. The solver combination is mentioned as a solution to small time increments for the explicit solver in [21] as well, together with coarsening of the resolution and increasing the horizon.

Wave behaviour, as seen in figures 55 to 61 for the slower explicit HS simulation, looks promising. Both destructive and constructive interference is exhibited, as well as wave reflections along the outer boundaries of the material. The number of time steps between each output is 200000, output aliasing problems could be an issue, where higher frequencies are indistinguishable from lower ones due to not sampling at least twice the highest frequency. This error is separate from aliasing issues caused by not modelling enough time steps to simulate the highest frequencies.

In any case, the wave behaviour could perhaps be useful for modelling more dynamic problems than tensile testing, for example the cutting-edge research at the division of Production and Materials Engineering at the faculty of engineering at Lund University, which has the added benefit of short time periods. The fractures mentioned earlier and pictured in 49 to 52 also demonstrates promising wave modelling, as the initial waves, already causing damage while propagating towards the centre, results in a fracture, as well as the secondary fractures caused by reflection of the waves at the end walls.

An interesting problem that comes to mind is mechanical vibrations resulting from an non-centred mass centre in a rotating axle. The setup for such an experiment would only require another discretisation, as well as displacement boundary conditions translated from a spherical coordinate system into a Cartesian one, assuming that proper material calibration can be made. Another interesting wave-phenomenon to investigate would be wave propagation, dispersion, and transmission between UFG and CG in the HS material.

Knowing that the resonance frequency is given by (85), and the spring constant in (84) is dependent solely on the bulk modulus and horizon, the resonance frequency would be the same for both materials as they possess the same elasticity modulus and Poisson's number, used in (56). However, waves would perhaps transmit in a non-homogeneous fashion given the earlier onset of plasticity in the CG material. This could possibly act as vibrational damping. Also, possible heterogeneous fracture behaviour might affect the propagation as well.

5.3 Material Models

The density value error could easily have been mitigated by double checking the value. A possible explanation is that the density was not mentioned in the same sources as the elasticity modulus, and as a result density values were never read from the source following the first occurrence of the error. The macro-scale yield point for both UFG and CG materials are consistently equal to the yield point specified in the material parameters of the input and the experimental data, irrespective of variation of time increments, critical stretch, or hardening modulus. This should be expected but given other unexpected behaviour, having it confirmed is positive.

The yield flaw affects the curve by smoothing the total transition to a fully plastic curve. This has potential usability in material simulation where the experimental initial yield occurs over a strain interval, as many metals do, instead of instantaneously. More concrete, this could be implemented by creating many small yield flaw locations evenly or "randomly" spaced in the deformation region. This would potentially create a quite extensive and intricate input file with many lines of code. An alternative would be to assigning a certain amount of randomly selected nodes in the deformation region, to a block with identical properties, apart from the yield stress property which could be set to a lower value.

The hardening models used are linear, not following the power law, which is not completely realistic for the nickel metal. However, the impact of the hardening curve being straight rather than curved might be lower than the impact from the fact that model extrapolation was made using three data points for the UFG material, taken analogous from a PDF image. Another reason is the wide gap for ultimate tensile strength mentioned in [48], which allows the UTS point to vary 200 MPa. Due to these reasons, the linear approximation of the power law might be sufficient for the modelling problem, unless necking is dependent solely on the inclination of the hardening curve.

This dependence on the inclination would imply an instantaneous or never-occurring necking, depending on whether the linear approximation of the hardening modulus is above or below a certain threshold. The stress-strain curve in Figure 32 on page 58 is serving as a base for this argument since necking occurs as soon as the material yields. A counterargument can be made using the behaviour of the stress-strain curve in Figure 33 on page 58, which features a high hardening rate but still experience necking at a point of extensive plastic strain.

Assuming necking is not only dependent on the hardening modulus inclination, the linear approximation could still have been adapted in a different way. The approximation made in this project is taken to be the line between the value of the power law at $\varepsilon = 0$, and at $\varepsilon = 1$. This approximation is faulty due to two main reasons. It does not take the elastic region into account, and intersects the power law at a later point than the UTS for both materials. The exact point at which plasticity is initiated in the UFG data is also not evident due to the low amount of data points.

The HS simulations made with the explicit solver shows preliminary positive results regarding bi-modal modelling. Fracture occurs mostly in the UFG material (red nodes), while propagating around the CG material (white nodes) in Figure 54 on page 66. The resolution is a bit coarse however, as there is only one node of UFG material between the CG cores at the shortest distance between them.

UFG material was used on the ends of the discretised material due to the issues which would arise with stress distribution in a displacement-controlled region, as well as on the border between the boundary condition region and the regular domain. This issue seems to be confirmed by Figure 35 on page 59, where the CG material at the y-displacement-locked bottom is seen to be damaged more compared to the UFG material.

The alternative of using CG material would not be suitable since a border where the CG BC domain is transformed into cores would be abrupt and result in a virtual grain-boundary across the whole cross-section of the domain. The downside of using the UFG material for the BC domain, is that it will possess less ductile properties, which could result in fracture in this region before the ductile CG cores can carry the plastic strain. But since the nodal stretch is lower further out due to the centre nodes having families containing nodes displaced in both axial directions, this might not be an issue.

By applying the peridynamic theory, a continuum approximation is made. Given that the current theory on HS materials suggest that dislocation movement plays a central role in its behaviour, the continuum approximation might not be suitable at all. If that is the case, it would also mean that CCM simulations would be able to model the behaviour neither. If the dislocation length scales are very small, the resource benefits of peridynamics over molecular dynamics might be lost since the number of nodes would have to increase to sufficiently model the dislocations.

5.4 Peridigm

A lot of manual tweaking was put into making Peridigm run. Achieving the installation used for the results presented in the article took about two months to complete. The documentation available online is not written for non-tech-savvy individuals, including the writer of this project. The shear amount of dependencies between libraries and programs which have to match takes a long time to figure out. Some of the versions of the software libraries prescribed in the DLR manual [21] where not found online, prompting other version to be used, possibly contributing to the errors in the final Peridigm installation. A new installation should preferably be made, one which does not yield any errors. Given that not even the computer cluster team at LUNARC managed to install it without using the script in Appendix A indicates that this might be a tedious task which takes time.

Expanding Peridigm to encompass new material models should be of no significant issue, given that the software as mentioned previously is open-source. With knowledge in C++, all the tools are available for writing new material models. There are also some instructions on user development in the PeriDox documentation [25]. On a related note, some of the material models also show signs of not being completely finished, as the code contains TODO- and debug comments as well as questions on implementation of certain methods.

In order to establish some sort of verification tool for material models, it is recommended to construct a Peridigm-specific material calibration manual. It is not something which has been found in the peridynamic literature during this project, and could be of great contribution to the international Peridigm community. The framework should preferably encompass the creation and discretisation of suitable domains, boundary conditions, modification of material parameters, as well as how to post-process the data.

Regarding Peridigm's viability for HS simulations, a more commercially established simulation tool should be used, unless time can be allocated to solving issues that will undoubtedly arise using Peridigm. This is mainly due to the existing issues in the code and the lack of support currently available, which is limited to the GitHub and the Peridigm-users email list [51].

For research on peridynamics it is on the other hand a very useful tool, compared to writing in-house code from scratch. Due to the already implemented parallelisation tools and existing libraries, the simulation framework and post-processing is ready-made, allowing researchers to focus on the more important aspects of peridynamics, such as new material formulations, multi-physics, and multi-scale modelling. Contributions to the software at this stage could perhaps substantially influence Peridigm's direction of development in favour of the contributor.

6 Conclusion

The viability of peridynamics in the research field of harmonic-structured materials has been evaluated in this report. It has also contributed several useful methodologies and scripts applicable to the Peridigm work-flow including pre- and post-processing, establishing a basis on which further research on peridynamics can be built.

The peridynamic theory is found to possesses the necessary tools to model the different behaviours of HS materials just as well as classical continuum mechanics, and especially well for fracture modelling. Whether the HS material is sufficiently modelled using the continuum approximation remains to be scrutinised. If the effect of discontinuities on the micro-structure is not readily modelled on a continuum scale, a molecular dynamics model might be better.

The Peridigm software however is still too much of a development project to justify the amount of time and programming knowledge required to make it run without errors, especially given the more user-friendly commercial software available.

At its current stage of development, Peridigm is not fully usable beyond the research field of computational mechanics. It will probably remain so for a foreseeable future, especially given the open-source nature of Peridigm, currently being the work of fourteen people in total. The low usability outside of computational mechanics could, and should, be changed by making use of the open-source concept. Contribution to new material and damage models on the Peridigm GitHub would increase its usefulness to other scientific fields, leading to more contributions and an increased spread, effectively propelling peridynamics to the front line of fracture computational modelling. Peridynamics is not the Swiss knife of computational modelling, and does not outshine either classical continuum mechanics or molecular dynamics in their respective fields of excellence. However, the ease with which it models fracture, simply by removing an element from the state vector, makes it an effective tool in continuum damage mechanics. As Florin Bobaru puts it [52]:

In peridynamics, cracks are part of the solution, not part of the problem.

6.1 Suggestions for Further Work

In order to provide an overview of the suggestions for further work mentioned in the discussion, the suggestions are compiled in the list below, in no specific order.

- Develop a material calibration manual for use in Peridigm, encompassing creation of test domain, discretisation, boundary conditions, material parameter modification, and post-processing.
- Combine solvers for faster computations.
- The linear approximation should be redefined by the line between initial yielding and UTS.
- Write new material models, especially power law elastic-plastic hardening models.
- An updated Peridigm installation with no errors
- Apply peridynamics to other research fields, like the metal cutting research at the faculty of engineering, due to the promising behaviour of the dynamic simulations.
- Apply peridynamics to model the dynamic behaviour of HS materials, especially wave propagation and dispersion due to heterogeneous plasticity and fracture.

References

[1] Dmytro Orlov and Kei Ameyama. Critical assesment 37: harmonic-structure materials - idea, status and perspectives. *Materials Science and Technology*, 36(5):517–526, 2020.

[2] Yuri Estrin and Alexei Vinogradov. Extreme grain refinement by severe plastic deformation: A wealth of challenging science. *Acta materialia*, 61(3):782–817, 2013.

[3] William D Callister et al. *Fundamentals of materials science and engineering*, volume 471660817. Wiley London, 2000.

[4] Dmytro Orlov, Jinming Zhou, Stephen Hall, Mie Ota-Kawabata, and Kei Ameyama. Advantages of architectured harmonic structure in structural performance. In *IOP Conference Series: Materials Science and Engineering*, volume 580, page 012019. IOP Publishing, 2019.

[5] Sanjay Kumar Vajpai, Mie Ota, Zhe Zhang, and Kei Ameyama. Three-dimensionally gradient harmonic structure design: an integrated approach for high performance structural materials. *Materials Research Letters*, 4(4):191–197, 2016.

[6] Dmitry Orlov, Hiroshi Fujiwara, and Kei Ameyama. Obtaining copper with harmonic structure for the optimal balance of structure-performance relationship. *Materials Transactions*, 54(9):1549–1553, 2013.

[7] Besim Ibishi. Finite-element simulations of harmonic structured material. Master's thesis, Lund University, Division of Mechanics, 2016.

[8] Jia Liu, Jia Li, Guy Dirras, Kei Ameyama, Fabien Cazes, and Mie Ota. A three-dimensional multi-scale polycrystalline plasticity model coupled with damage for pure ti with harmonic structure design. *International Journal of Plasticity*, 100:192–207, 2018.

[9] Han Yu, IKUMU Watanabe, and KEI Ameyama. Deformation behavior analysis of harmonic structure materials by multi-scale finite element analysis. In *Advanced Materials Research*, volume 1088, pages 853–857. Trans Tech Publ, 2015.

[10] Ellad B Tadmor, Ronald E Miller, and Ryan S Elliott. *Continuum mechanics and thermodynamics: from fundamental concepts to governing equations*. Cambridge University Press, 2012.

[11] Ellad B Tadmor and Ronald E Miller. *Modeling materials: continuum, atomistic and multiscale techniques*. Cambridge University Press, 2011.

[12] A Cemal Eringen and DGB Edelen. On nonlocal elasticity. *International Journal of Engineering Science*, 10(3):233–248, 1972.

[13] Erdogan Madenci and Erkan Oterkus. *Peridynamics Theory and Its Applications*. Springer, 2014.

[14] Stewart A Silling. Reformulation of elasticity theory for discontinuities and long-range forces. *Journal of the Mechanics and Physics of Solids*, 48(1):175–209, 2000.

[15] Stewart A. Silling, M. Epton, Olaf Weckner, J. Xu, and Ebrahim Askari. Peridynamic states and constitutive modeling. *Journal of Elasticity*, 88:151–184, 2007.

[16] Yinmin Wang, Mingwei Chen, Fenghua Zhou, and En Ma. High tensile ductility in a nanostructured metal. *Nature*, 419(6910):912–915, 2002.

[17] NA Yefimov. *Handbook of non-ferrous metal powders: technologies and applications*. Elsevier, 2009.

[18] F. Bobaru, J.T. Foster, P.H. Geubelle, and S.A. Silling. *Handbook of Peridynamic Modeling*. Modern Mechanics and Mathematics. Taylor & Francis, 2015.

[19] James L Meriam and L Glenn Kraige. *Statics*, volume 1. John Wiley & Sons, 2008.

[20] Richard W Macek and Stewart A Silling. Peridynamics via finite element analysis. *Finite Elements in Analysis and Design*, 43(15):1169–1178, 2007.

[21] Martin Rädel. Peridigm users guide. Technical report, German Aerospace Center, 2018.

[22] Stewart A Silling and Ebrahim Askari. A meshfree method based on the peridynamic model of solid mechanics. *Computers & structures*, 83(17-18):1526–1535, 2005.

[23] Michael L Parks, David J Littlewood, John A Mitchell, and Stewart A Silling. Peridigm users' guide v1. 0.0. *SAND Report*, 7800, 2012.

[24] Michael L Parks, David J Littlewood, John A Mitchell, and Stewart A Silling. Peridigm. `https://github.com/Peridigm/Peridigm`, 2020. Version 1.5.0. Accessed: 2020-06-08.

[25] Martin Rädel and Christian Willberg. PeriDoX. GitHub repository, 03 2018.

[26] Extensible markup language (XML). `https://www.w3.org/XML/`. Accessed: 2020-05-15.

[27] YAML Ain't Markup Language (YAML™) Version 1.2. `https://yaml.org/spec/1.2/spec.html`. Accessed: 2020-05-15.

[28] Larry A. Schoof and Victor R. Yarberry. *Exodus II: A Finite Element Data Model*. Albuquerque, New Mexico 87185 and Livermore, California 94550, USA, 1994.

[29] David Littlewood. `https://github.com/peridigm/peridigm/issues/114#issuecomment-590420076`. Posted: 2020-02-24.

[30] Michael L Parks, David J Littlewood, John A Mitchell, and Stewart A Silling. Peridigm_Discretization.cpp. https://github.com/peridigm/peridigm/blob/master/src/io/discretization/Peridigm_Discretization.cpp, 2020. Version 1.5.0. Accessed: 2020-05-17.

[31] John A. Mitchell. A nonlocal, ordinary, state-based plasticity model for peridynamics. Sandia Report SAND2011-3166, Multiphysics Simulation Technologies, Sandia National Laboratories, 2011.

[32] Michael L Parks, David J Littlewood, John A Mitchell, and Stewart A Silling. Peridigm_CorrespondenceMaterial.cpp. https://github.com/peridigm/peridigm/blob/master/src/materials/Peridigm_CorrespondenceMaterial.cpp, 2020. Version 1.5.0. Accessed: 2020-05-15.

[33] Niels Saabye Ottosen and Matti Ristinmaa. *The mechanics of constitutive modeling*. Elsevier, 2005.

[34] Michael L Parks, David J Littlewood, John A Mitchell, and Stewart A Silling. elastic_plastic_hardening.cxx. https://github.com/peridigm/peridigm/blob/master/src/materials/elastic_plastic_hardening.cxx, 2020. Version 1.5.0. Accessed: 2020-05-27.

[35] Michael L Parks, David J Littlewood, John A Mitchell, and Stewart A Silling. Peridigm_isotropic_hardening_correspondence.cxx. https://github.com/peridigm/peridigm/blob/master/src/materials/isotropic_hardening_correspondence.cxx, 2020. Version 1.5.0. Accessed: 2020-05-17.

[36] Michael L Parks, David J Littlewood, John A Mitchell, and Stewart A Silling. Peridigm_InterfaceAwareDamageModel.cpp. https://github.com/peridigm/peridigm/blob/master/src/damage/Peridigm_InterfaceAwareDamageModel.cpp, 2020. Version 1.5.0. Accessed: 2020-06-07.

[37] Michael L. Parks, David J. Littlewood, John A. Mitchell, and Stewart A. Silling. *Peridigm Users Guide v1.0.0*. Albuquerque, New Mexico 87185 and Livermore, California 94550, USA, 2012.

[38] Michael L Parks, David J Littlewood, John A Mitchell, and Stewart A Silling. Peridigm.cpp. https://github.com/peridigm/peridigm/blob/master/src/core/Peridigm.cpp, 2020. Version 1.5.0. Accessed: 2020-05-29.

[39] Michael L Parks, David J Littlewood, John A Mitchell, and Stewart A Silling. Peridigm_CriticalTimeStep.cpp. https://github.com/peridigm/peridigm/blob/master/src/core/Peridigm_CriticalTimeStep.cpp, 2020. Version 1.5.0. Accessed: 2020-05-29.

[40] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[41] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[42] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[43] James Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. *The visualization handbook*, 717, 2005.

[44] MATLAB. *9.7.0.1319299 (R2019b)*. The MathWorks Inc., Natick, Massachusetts, 2020.

[45] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, September 2009.

[46] Spencer Kimball, Peter Mattis, and the GIMP Development Team. GNU image manipulation program. Version 2.8.22. Accessed: 2020-05-28.

[47] C Moosbrugger. Atlas of stress-strain curves. *ASM International, Materials Park*, page 375, 2002.

[48] Special Metals Corporation. Nickel 200 & 201. `https://www.specialmetals.com/assets/smc/documents/alloys/nickel-duranickel/nickel-200-201.pdf`, September 2006. Accessed 2020-05-28.

[49] Abdallah Shokry, Aylin Ahadi, Per Ståhle, and Dmytro Orlov. On the influence of topological arragnements in bimodal grain size distribution on the mechanical behaviour of pure nickel. Unpublished. Collaboration between Fayoum University and Lund University.

[50] Georg B Rathmayr and Reinhard Pippan. Influence of impurities and deformation temperature on the saturation microstructure and ductility of hpt-deformed nickel. *Acta materialia*, 59(19):7228–7240, 2011.

[51] Peridigm-users Info Page. https://software.sandia.gov/mailman/listinfo/peridigm-users. Accessed: 2020-06-08.

[52] Florin Bobaru. https://engineering.unl.edu/mme/florin-bobaru/. Accessed: 2020-06-08.

Appendices

```
##########################################################################
####                   PERIDIGMINSTALLATION 2.0                      ####
#### ,               theodor.desousa@gmail.com                       ####
##########################################################################

# This is the installation script used for installing peridigm on my laptop.
# Many directories, variables etc are not generalisable, the code must be adapted
# to the situation. Simply copy-pasting the code into a terminal will not work.

# It didn't generate a fully functional peridigm which pass all tests however.
# This is a large issue, but given limited time, it was deemed enough for a
# proof of concept.


#--------------------------------#
##        What hardware?        ##
#--------------------------------#

# OS: Ubuntu 18.04.4 LTS 64-bit
# Processor: Intel i7-7500U 2.70 GHz*4
# Graphics: Intel HD Graphics 620 Kaby Lake GT2
# Memory 15.5 GiB
# GNOME version: 3.28.2
#


#--------------------------------#
##       What to install?       ##
#--------------------------------#

# BASICS                        DEPENDENCIES:                      COMMENTS:
#   COMPILERS:
#       gfortran    4.8.5                   -
#       gcc         4.8.5                   -
#       g++         4.8.5                   -
#   CMake           3.5.1                   Compilers
#   Open-MPI        2.1.5 (DLR: 1.10.2)     Compilers                Bug in 1.10.x with HDF 1.10.x
#   Python          2.7.9                   -
# LIBRARIES
#   Boost           1.60.0                  OpenMPI
#   HDF5            1.10.3 (DLR: 1.10.0)    OpenMPI
#   NetCDF-C        4.4.0                   OpenMPI, HDF5, m4
#       m4                                  -
#   Trilinos        12.12.1 (DLR: 12.6.1)   CMake,OpenMPI,Boost,HDF5,NetCDF-C,Blas,Lapack,X11
#       Blas                                -
#       Lapack                              -
#       X11                                 -
# Peridigm          1.5.0                   CMake, OpenMPI, Python, Boost, Trilinos


#--------------------------------#
##       In which order?        ##
#--------------------------------#

# BASICS:
# 1:    gfortran, gcc, g++
# 2:    CMake
# 3:    OpenMPI
# 4:    Python - should be the newest
# LIBRARIES:
# 5:    bz2, zlib, m4, blas, lapack, libX11
# 6:    Boost
# 7:    HDF5
# 8:    NetCDF-C
# 9:    Trilinos
# 10:   Peridigm


# Main structure of installation:
# /home
#    /$USER ("theodor" in my case)
#      /peri   (the installation directory where I put the peridigm-related programs)
#      /src    (the source directory where I put tar.gz and the resulting sources)



#--------------------------------#
##        Installation          ##
#--------------------------------#

sudo apt-get update
#Checkinstall will allow for easier uninstallation as it can be uninstalled with "sudo apt remove $PACKAGE.
sudo apt-get install -y checkinstall

# Prepare .bashrc
sudo echo "##########################################################################
####                        USER SETTINGS                            ####
##########################################################################" >>~/.bashrc
```

```
sudo echo '' >>~/.bashrc
. ~/.bashrc
'

#Everything needed for peridigm should be installed in the peri folder
mkdir /home/$USER/peri/
#Their sources should be put here
mkdir /home/$USER/src/

#----COMPILERS----#
cd /home/$USER/src/
# Install from source these versions if not already installed.
gcc --version           # == 4.8.5
g++ --version           # == 4.8.5
gfortran --version      # == 4.8.5

sudo update-alternatives --remove-all gcc
sudo update-alternatives --remove-all g++
sudo update-alternatives --remove-all gfortran
sudo apt-get install gcc-4.8 g++-4.8 gfortran-4.8

sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 10

sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-4.8 10

sudo update-alternatives --install /usr/bin/gfortran gfortran /usr/bin/gfortran-4.8 10


sudo update-alternatives --install /usr/bin/cc cc /usr/bin/gcc 30
sudo update-alternatives --set cc /usr/bin/gcc

sudo update-alternatives --install /usr/bin/c++ c++ /usr/bin/g++ 30
sudo update-alternatives --set c++ /usr/bin/g++



#####################
#----CMake 3.5.1----#
#####################
cd /home/$USER/src

wget https://cmake.org/files/v3.5/cmake-3.5.1.tar.gz
tar -xf cmake-3.5.1.tar.gz
cd cmake-3.5.1/
#Specify compiler (has to be the same for all installations):
env CC=gcc CXX=g++ ./bootstrap >cmake_boostrap.log 2>&1
make >cmake_make.log 2>&1
sudo make test >cmake_test.log 2>&1

sudo make install >cmake_install.log 2>&1


#ALTERNATIV 1 yields these errors:
# sudo checkinstall
#The following tests FAILED:
#     7 - kwsys.testSystemTools (Failed)
#   103 - Simple_EclipseGenerator (Failed)
#   104 - Simple_CodeBlocksGenerator (Failed)
#   105 - Simple_KDevelop3Generator (Failed)
#   191 - CTestCoverageCollectGCOV (Failed)
#   223 - CTestLimitDashJ (Failed)
#   258 - CMakeOnly.AllFindModules (Failed)

#Alternativ 2 yields these errors:
# sudo make test >cmake_test.log 2>&1
#The following tests FAILED:
#     7 - kwsys.testSystemTools (Failed)
#   103 - Simple_EclipseGenerator (Failed)
#   104 - Simple_CodeBlocksGenerator (Failed)
#   105 - Simple_KDevelop3Generator (Failed)
#   181 - WarnUnusedCliUnused (Failed)
#   191 - CTestCoverageCollectGCOV (Failed)
#   258 - CMakeOnly.AllFindModules (Failed)
#Installed in usr/local/bin, usr/local/share, usr/local/doc

#########################
#----Open-mpi 2.1.5-----#
#########################
cd /home/$USER/src/
wget https://download.open-mpi.org/release/open-mpi/v2.1/openmpi-2.1.5.tar.gz
tar -xf openmpi-2.1.5.tar.gz
cd openmpi-2.1.5/

./configure --prefix=/home/$USER/peri/openmpi-2.1.5 > ompi_configure.log 2>&1
make -j 4 > ompi_make.log 2>&1
make check

sudo checkinstall

#Update path variables in .bashrc: $USER should perhaps be replaced with something else, or use
```

```
#double quotation marks "$USER" to allow it to print the current user.
sudo echo 'export CC=mpicc
export CXX=mpicxx
export FC=mpif90
export F77=mpif77
export CXXFLAGS="$CXXFLAGS -fPIC"' >>~/.bashrc
sudo echo '' >>~/.bashrc
sudo echo 'export PATH=$PATH:/home/$USER/peri/openmpi-2.1.5/bin' >>~/.bashrc
sudo echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/$USER/peri/openmpi-2.1.5/lib' >>~/.bashrc
sudo echo '' >>~/.bashrc
#Reload the file in the shell
. ~/.bashrc

################
#----Python----#
################
# Should be installed already on a linux distribution
python --version


#########################
#----Basic Libraries----#
#########################
sudo apt-get update
sudo apt-get install -y libbz2-dev      # bz2
sudo apt-get install -y zlib1g-dev      # zlib
sudo apt-get install -y m4              # m4
sudo apt-get install -y libblas-dev     # blas
sudo apt-get install -y liblapack-dev   # lapack
sudo apt-get install -y libx11-dev      # X11

# Make sure NetCDF and HDF5 are removed
sudo apt list | grep netcdf #If installed using checkinstall instead of make install, they can be found
here
sudo apt list | grep hdf5 #If installed using checkinstall instead of make install, they can be found here
#check other names as well
sudo apt remove netcdfperi
sudo apt remove hdf5peri

###############
#----Boost----#
###############
# MIGHT NOT BE NECESSARY FOR PERIDIGM 1.5.0, could try disregarding.
#   Depends on: OpenMPI
cd /home/$USER/src
wget http://sourceforge.net/projects/boost/files/boost/1.60.0/boost_1_60_0.tar.gz
tar xvfz boost_1_60_0.tar.gz
cd boost_1_60_0

echo '# Set environment variables for MPI compilers
export CC=mpicc
export CXX=mpicxx
export FC=mpif90
export F77=mpif77
# Run the Boost bootstrap script
./bootstrap.sh
# add using mpi to project-config.jam
echo "using mpi ;" >> project-config.jam
# Compile and install Boost using the Boosts bjam build system
./b2 install --prefix=/home/$USER/peri/boost-1.60.0
' >> boost_install.sh
chmod +x boost_install.sh

./boost_install.sh > boost_install.log 2>&1

sudo echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/$USER/peri/boost-1.60.0/lib' >>~/.bashrc
sudo echo '' >>~/.bashrc
. ~/.bashrc

##############
#----HDF5----#
##############
#   Depends on: OpenMPI
cd /home/$USER/src
wget https://support.hdfgroup.org/ftp/HDF5/prev-releases/hdf5-1.10/hdf5-1.10.3/src/hdf5-1.10.3.tar.gz
tar  xvfz hdf5-1.10.3.tar.gz
cd hdf5-1.10.3/

echo '# Set environment variables for MPI compilers
export CC=mpicc
export CXX=mpicxx
export FC=mpif90
export F77=mpif77
# Configure HDF5
./configure --prefix=/home/$USER/peri/hdf5-1.10.3/ --enable-parallel
# Make and install HDF5
make -j 4 >hdf5_make.log 2>&1
' >> hdf5_install.sh
chmod +x hdf5_install.sh
```

```
./hdf5_install.sh >hdf5_install.log

make check
# The parallel HDF5 test "t_pflush1" will pass, but the return value of mpirun will be non-zero since
MPI_Finalize is not called,
# which is listed as one of three reasons when googling. As a result, the "t_pflush1" test will fail. A
work-around is below, which just creates a fake file.
cd testpar
touch t_pflush1.chkexe
cd ..; make check



sudo checkinstall

sudo echo 'export PATH=$PATH:/home/$USER/peri/hdf5-1.10.3/bin' >>~/.bashrc
sudo echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/$USER/peri/hdf5-1.10.3/lib'  >>~/.bashrc
sudo echo '' >>~/.bashrc
. ~/.bashrc

##################
#----NetCDF-C----#
##################
#tråd om peridigm och NetCDF-C.
#https://www.unidata.ucar.edu/support/help/MailArchives/netcdf/msg14000.html
#   Depends on: OpenMPI, HDF5, m4
cd /home/$USER/src

#Important to use 4.4.1.1, many bugfixes
wget https://github.com/Unidata/netcdf-c/archive/v4.4.1.1.tar.gz
tar xvfz v4.4.1.1.tar.gz
cd netcdf-c-4.4.1.1/


# Below is a suggested modification to some MAX values. This makes the installation fail.
# Other values could perhaps be tried as well, I just disregarded it however.

#Insert into : ./include/netcdf.h
#cd include/
#sed -i 's/#define\sNC_MAX_DIMS.*$/#define NC_MAX_DIMS 65536/g' netcdf.h
#sed -i 's/#define\sNC_MAX_ATTRS.*$/#define NC_MAX_ATTRS 8192/g' netcdf.h
#sed -i 's/#define\sNC_MAX_VARS.*$/#define NC_MAX_VARS 524288/g' netcdf.h
#sed -i 's/#define\sNC_MAX_NAME.*$/#define NC_MAX_NAME 256/g' netcdf.h
#sed -i 's/#define\sNC_MAX_VAR_DIMS.*$/#define NC_MAX_VAR_DIMS 8/g' netcdf.h
#cd ..

echo '# Set environment variables for MPI compilers
export CC=mpicc
export CXX=mpicxx
export FC=mpif90
export F77=mpif77
# Configure NetCDF
#CPPFLAGS="-I/home/$USER/peri/hdf5-1.10.3/include" \
#LDFLAGS="-L/home/$USER/peri/hdf5-1.10.3/lib" \
./configure --prefix=/home/$USER/peri/netcdf-4.4.1.1/ --disable-netcdf-4 --disable-dap
# Make and test NetCDF
make -j 4
make check
'>>netcdf_install.sh
chmod +x netcdf_install.sh
./netcdf_install.sh >netcdf_install.log 2>&1

sudo checkinstall

sudo echo 'export PATH=$PATH:/home/$USER/peri/netcdf-4.4.1.1/bin' >>~/.bashrc
sudo echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/$USER/peri/netcdf-4.4.1.1/lib' >>~/.bashrc
sudo echo '' >>~/.bashrc
. ~/.bashrc

##################
#----YAML-CPP----#
##################

cd /home/$USER/src
git clone https://github.com/jbeder/yaml-cpp.git
cd yaml-cpp
mkdir build
cd build

#Important for trilinos to be able to build shared libraries, which is necessary for the script
# text_to_genesis.py to work.
cmake -DYAML_BUILD_SHARED_LIBS=ON ..
make
make test

sudo make install # since it installs in /usr/local/ we need to do it using sudo. Could be installed
somewhere else using setprefix probably

##################
```

```
#----Trilinos----#
##################
#   Depends on: CMake, OpenMPI, Boost, HDF5, NetCDF-C, Blas, Lapack, yaml-cpp
# Found the trilinos version on a computer. Trilinos webpage was broken and didn't respond to wget

cd /home/$USER/src
tar xvfz trilinos-12.12.1-Source.tar.gz
mkdir trilinos-12.12.1
cd trilinos-12.12.1


#This file is the most critical part of the installation. It takes time to make, and errors might be
discovered first after an hour, or when installing peridigm.
echo 'rm -f CMakeCache.txt
rm -rf CMakeFiles/
cmake -D CMAKE_INSTALL_PREFIX:PATH=/home/$USER/peri/trilinos-12.12.1/ \
-D MPI_BASE_DIR:PATH="/home/$USER/peri/openmpi-1.10.2" \
-D CMAKE_CXX_FLAGS:STRING="-O2 -std=c++11 -pedantic -ftrapv -Wall -Wno-long-long" \
-D CMAKE_BUILD_TYPE:STRING=RELEASE \
-D BUILD_SHARED_LIBS=ON \
-D Trilinos_WARNINGS_AS_ERRORS_FLAGS:STRING="" \
-D Trilinos_ENABLE_ALL_PACKAGES:BOOL=OFF \
-D Trilinos_ENABLE_Teuchos:BOOL=ON \
-D Trilinos_ENABLE_Shards:BOOL=ON \
-D Trilinos_ENABLE_Sacado:BOOL=ON \
-D Trilinos_ENABLE_Epetra:BOOL=ON \
-D Trilinos_ENABLE_EpetraExt:BOOL=ON \
-D Trilinos_ENABLE_Ifpack:BOOL=ON \
-D Trilinos_ENABLE_AztecOO:BOOL=ON \
-D Trilinos_ENABLE_Amesos:BOOL=ON \
-D Trilinos_ENABLE_Anasazi:BOOL=ON \
-D Trilinos_ENABLE_Belos:BOOL=ON \
-D Trilinos_ENABLE_ML:BOOL=ON \
-D Trilinos_ENABLE_Phalanx:BOOL=ON \
-D Trilinos_ENABLE_Intrepid:BOOL=ON \
-D Trilinos_ENABLE_NOX:BOOL=ON \
-D Trilinos_ENABLE_Stratimikos:BOOL=ON \
-D Trilinos_ENABLE_Thyra:BOOL=ON \
-D Trilinos_ENABLE_Rythmos:BOOL=ON \
-D Trilinos_ENABLE_MOOCHO:BOOL=ON \
-D Trilinos_ENABLE_TriKota:BOOL=OFF \
-D Trilinos_ENABLE_Stokhos:BOOL=ON \
-D Trilinos_ENABLE_Zoltan:BOOL=ON \
-D Trilinos_ENABLE_Piro:BOOL=ON \
-D Trilinos_ENABLE_Teko:BOOL=ON \
-D Trilinos_ENABLE_SEACASIoss:BOOL=ON \
-D Trilinos_ENABLE_SEACAS:BOOL=ON \
-D Trilinos_ENABLE_SEACASBlot:BOOL=ON \
-D Trilinos_ENABLE_Pamgen:BOOL=ON \
-D Trilinos_ENABLE_EXAMPLES:BOOL=OFF \
-D Trilinos_ENABLE_TESTS:BOOL=ON \
-D TPL_ENABLE_Matio:BOOL=OFF \
-D TPL_ENABLE_HDF5:BOOL=ON \
-D HDF5_INCLUDE_DIRS:PATH="/home/$USER/peri/hdf5-1.10.3/include" \
-D HDF5_LIBRARY_DIRS:PATH="/home/$USER/peri/hdf5-1.10.3/lib" \
-D TPL_ENABLE_Netcdf:BOOL=ON \
-D Netcdf_INCLUDE_DIRS:PATH="/home/$USER/peri/netcdf-4.4.1.1/include" \
-D Netcdf_LIBRARY_DIRS:PATH="/home/$USER/peri/netcdf-4.4.1.1/lib" \
-D TPL_ENABLE_MPI:BOOL=ON \
-D TPL_ENABLE_BLAS:BOOL=ON \
-D TPL_ENABLE_LAPACK:BOOL=ON \
-D TPL_ENABLE_Boost:BOOL=ON \
-D Boost_INCLUDE_DIRS:PATH="/home/$USER/peri/boost-1.60.0/include" \
-D Boost_LIBRARY_DIRS:PATH="/home/$USER/peri/boost-1.60.0/lib" \
-D TPL_ENABLE_yaml-cpp:BOOL=ON \
-D yaml-cpp_INCLUDE_DIRS:PATH=/usr/local/include/yaml-cpp/include \
-D yaml-cpp_LIBRARY_DIRS:PATH=/usr/local/include/yaml-cpp/lib \
-D CMAKE_VERBOSE_MAKEFILE:BOOL=ON \
-D Trilinos_VERBOSE_CONFIGURE:BOOL=ON \
/home/$USER/src/trilinos-12.12.1-Source/
'>cmake_trilinos.cmake

 chmod +x cmake_trilinos.cmake
./cmake_trilinos.cmake >trilinos_cmake.log 2>&1
make -j 4   >trilinos_make.log 2>&1
make
make test

sudo checkinstall

sudo echo 'export PATH=$PATH:/home/$USER/peri/trilinos-12.12.1/bin' >>~/.bashrc
sudo echo 'export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/$USER/peri/trilinos-12.12.1/lib' >>~/.bashrc
sudo echo '' >>~/.bashrc
. ~/.bashrc

##################
#----Peridigm----#
##################
#   Depends on: CMake, OpenMPI, Python, Boost, Trilinos, yaml-cpp
```

```
cd /home/$USER/src
git clone https://github.com/peridigm/peridigm.git

mv peridigm/ Peridigm-1.5.0-Source

mkdir peridigm-1.5.0

cd peridigm-1.5.0

echo 'rm -f CMakeCache.txt
rm -rf CMakeFiles/
cmake \
-D CMAKE_BUILD_TYPE:STRING=Release \
-D CMAKE_INSTALL_PREFIX=/home/$USER/peri/peridigm-1.5.0 \
-D Trilinos_DIR:PATH=/home/$USER/peri/trilinos-12.12.1/lib/cmake/Trilinos/ \
-D CMAKE_C_COMPILER:STRING=/home/$USER/peri/openmpi-2.1.5/bin/mpicc \
-D CMAKE_CXX_COMPILER:STRING=/home/$USER/peri/openmpi-2.1.5/bin/mpicxx \
-D BOOST_ROOT=/home/$USER/peri/boost-1.60.0/ \
-D CMAKE_CXX_FLAGS:STRING="-O2 -Wall -std=c++11 -pedantic -Wno-long-long -ftrapv -Wno-deprecated" \
/home/$USER/src/Peridigm-1.5.0-Source' > cmake_peridigm.cmake
chmod +x cmake_peridigm.cmake
./cmake_peridigm.cmake
make -j 4
make
make test


#The following tests FAILED, must be solved:
#   187 - IsotropicHardeningPlasticFullyPrescribedTension_NoFlaw_np1 (Failed)
#   188 - IsotropicHardeningPlasticFullyPrescribedTension_NoFlaw_np4 (Failed)
#   189 - IsotropicHardeningPlasticFullyPrescribedTension_WithFlaw_np1 (Failed)
#   190 - IsotropicHardeningPlasticFullyPrescribedTension_WithFlaw_np4 (Failed)

sudo checkinstall

#Create symbolic link to Peridigm
sudo ln -s ~/peri/peridigm-1.5.0/bin/Peridigm /usr/local/bin/

#--------------------------------#
##  Checkinstall package names  ##
#--------------------------------#
#CMake:     cmakeperi
#Openmpi:   openmpiperi
#HDF5:      Not installed using checkinstall
#NetCDF:    netcdfperi
#Trilinos:  trilinosperi
#Peridigm:  peridigmperi

#--------------------------------#
##        Final .bashrc         ##
#--------------------------------#

export CC=mpicc
export CXX=mpicxx
export FC=mpif90
export F77=mpif77

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/$USER/peri/boost-1.60.0/lib

export PATH=$PATH:/home/$USER/peri/openmpi-2.1.5/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/$USER/peri/openmpi-2.1.5/lib

export PATH=$PATH:/home/$USER/peri/hdf5-1.10.3/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/$USER/peri/hdf5-1.10.3/lib

export PATH=$PATH:/home/$USER/peri/trilinos-12.12.1/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/$USER/peri/trilinos-12.12.1/lib
```

```
Peridigm:
    Verbose: false          #Limit output information for log files.
    Discretization:
        Type: "Exodus"      #If input is text file, change to "Text File"
        Input Mesh File: "domainMeshNoPert.g"  #Do no specify core decomposition
here.


    Materials:
        CG:
            Material Model: "Elastic Plastic Hardening"
            Density: 7850.0     #safe increase is \rho*60 according to
"Computational materials engineering"
            Young's Modulus: 210.0e9
            Poisson's Ratio: 0.31
            Yield Stress: 185e6
            Hardening Modulus: 980.9e6
        UFG:
            Material Model: "Elastic Plastic Hardening"
            Density: 7850.0
            Young's Modulus: 210.0e9
            Poisson's Ratio: 0.31
            Yield Stress: 1502.0e6
            Hardening Modulus: 4355.5e6

    Damage Models:
        My CG Dmg Model:
            Damage Model: "Critical Stretch"
            Critical Stretch: 0.2242 #Has to be found for every horizon by
comparing with real data,
        My UFG Dmg Model:
            Damage Model: "Critical Stretch"
            Critical Stretch: 0.1

    Blocks:
        My CG Block:
            Block Names: "block_1"
            Material: "CG"
            Damage Model: "My CG Dmg Model"
            Horizon: 7.3892e-5
        My UFG Block:
            Block Names: "block_2"
            Material: "UFG"
            Damage Model: "My UFG Dmg Model"
            Horizon: 7.3892e-5

    Boundary Conditions:
        Create Node Set For Rank Deficient Nodes: true

        # X-direction
        Prescribed Displacement Bottom In X:
            Type: "Prescribed Displacement"
            Node Set: "nodelist_1"
            Coordinate: "x"
            Value: "t*x*3.6e+5/2 "
        Prescribed Displacement Top In X:
            Type: "Prescribed Displacement"
            Node Set: "nodelist_2"
            Coordinate: "x"
            Value: "t*x*3.6e+5/2"

        # Y-direction
        Prescribed Displacement Fix Top Rigid Body Motion In Y:
            Type: "Prescribed Displacement"
            Node Set: "nodelist_9"
            Coordinate: "z"
            Value: "0.0"

        # Z-direction
        Prescribed Displacement Fix Bottom Rigid Body Motion In Z:
            Type: "Prescribed Displacement"
            Node Set: "nodelist_10"
            Coordinate: "y"
            Value: "0.0"

    Solver:
        Verbose: true                   #Provides more solver output
        Initial Time: 0.0
        Final Time: 9.0e-6
        Disable Heuristics: true        #Apparently useful for fractures.
        Verlet:                         #Specifies the solver
            Safety Factor: 0.01         #Decreases the time increment.
            Numerical Damping: 160.0e3  #Reduces vibrations from fractures etc.


##  Below is a quasi-static solver
#       Peridigm Preconditioner: "None"
#       NOXQuasiStatic:
#            Nonlinear Solver: "Line Search Based"
#            Number of Load Steps: 10000
#            Max Solver Iterations: 100
#            Relative Tolerance: 5.0e-4
#            Max Age Of Prec: 100
#            Direction:
#              Method: "Newton"
#              Newton:
#                Linear Solver:
#                  Jacobian Operator: "Matrix-Free"
#                  Preconditioner: "None"
#            Line Search:
#              Method: "Polynomial"


#   This section allows the user to specify calculations to be performed.
    Compute Class Parameters:
#       This compute class saves the initial position of the left side of the
strain gauge
        Strain Gage Left Initial Position:
            Compute Class: "Nearest_Point_Data"
#           Coordinates used for the Model_Coordinates variable
            X: -0.0018
            Y: 0.0
            Z: 0.0
            Variable: "Model_Coordinates"
#           Specify the output label in the output section as well
            Output Label: "Gage_Left_Initial_Position"
            Verbose: true

#       Same as above
        Strain Gage Right Initial Position:
            Compute Class: "Nearest_Point_Data"
            X: 0.0018
            Y: 0.0
            Z: 0.0
            Variable: "Model_Coordinates"
            Output Label: "Gage_Right_Initial_Position"
            Verbose: true

        Strain Gage Left Displacement:
            Compute Class: "Nearest_Point_Data"
            X: -0.0018
            Y: 0.0
            Z: 0.0
            Variable: "Displacement"
#           Specify the output label in the output section as well
            Output Label: "Gage_Left_Displacement"
            Verbose: true
        Strain Gage Right Displacement:
            Compute Class: "Nearest_Point_Data"
            X: 0.0018
            Y: 0.0
            Z: 0.0
            Variable: "Displacement"
#           Specify the output label in the output section as well
            Output Label: "Gage_Right_Displacement"
            Verbose: true

        Left Reaction Force:
            Compute Class: "Node_Set_Data"
            Calculation Type: "Sum"
            Node Set: "nodelist_1"
            Variable: "Force"
#           Specify the output label in the output section as well
            Output Label: "Left_Reaction_Force"

        Right Reaction Force:
            Compute Class: "Node_Set_Data"
            Calculation Type: "Sum"
            Node Set: "nodelist_2"
            Variable: "Force"
#           Specify the output label in the output section as well
            Output Label: "Right_Reaction_Force"

    Output:
        Output File Type: "ExodusII"
        Output Filename: "Verlet_Experiment"
        Output Frequency: 200000 #Increase as much as possible to reduce file size.
        Output Variables:        #Less variables reduce filesize.
#           Standard output and material model.
            Block_Id: true
            Coordinates: true
            Damage: true
            Deformation_Gradient: true
            Displacement: true
            Element_Id: true
            Force_Density: true
            Force: true
            Neighborhood_Volume: true
            Number_Of_Neighbors: true
            Proc_Num: true
            Radius: true
            Surface_Correction_Factor: true
            Velocity: true
            Volume: true
#           Compute class output
            Right_Reaction_Force: true
            Gage_Left_Displacement: true
            Gage_Left_Initial_Position: true
            Gage_Right_Displacement: true
            Gage_Right_Initial_Position: true
            Left_Reaction_Force: true
```

C Job Farm Prototype Script

```bash
#!/bin/bash
# FUNCTION FOR RUNNING PERIDIGM IN EACH DIRECTORY
Perirun () {
    for k in $(seq $1 1 $2 )
    do
        dir="NonCorrbulk_Nickel_plasticity_$k"
        echo "--------Starting simulation in $dir--------"
        cd $dir
        ret=$?
        if [ $ret -eq 0 ]
        then
            taskset --cpu-list $3 time Peridigm piripiri.yaml >peridigmSim.log 2>&1
            date
        else
            echo "        Simulation Failed in $dir          "
        fi
        cd ..
        echo "--------Finished simulation in $dir--------"
        sleep 1
    done
}


# MAIN
# Names still needs to be modified depending on circumstances, could be modified using $<variable> instead
if [ "$BASH" != "/bin/bash" ]; then
  echo "Please do ./$0"
  exit 1
fi
export LC_NUMERIC="en_US.UTF-8" #Solves issues with decimal comma during this run
# urVise should contain the yaml file named "piripiri.yaml", and discretisation named as specified in the
yaml file.
export urdir="/home/mekk/Documents/Peridigm de Sousa/Simulations/NonCorrPlasticitySims/urVise/"
cd "/home/mekk/Documents/Peridigm de Sousa/Simulations/NonCorrPlasticitySims/"


echo "Building Directories"

i=1
# The for loops are used for generating modifications to the yaml file. Can be used to modify any line
with the sed command.
for trigger in 1 2
do
 #   for cs in $(seq 0.0000063 0.0000005 0.000017 )
  #  do
        dir="NonCorrbulk_Nickel_plasticity_$i"
        mkdir "$dir"

        # Copies the generic files to be modified into the new dir.
        cp -r --strip-trailing-slashes "$urdir/."  "$dir"
        cd $dir

        # Modify the relevant parameters in the .yaml file
        if [ $trigger -eq 1 ]
        then
            sed -i -e "s/Material:\s.*/Material: \"UFG\"/" piripiri.yaml
        else
            sed -i -e "s/Material:\s.*/Material: \"CG\"/" piripiri.yaml
        fi
       #sed -i -e "s/Horizon:\s[0-9].*/Horizon: $cs/" piripiri.yaml
        sed -i -e "s/Output\sFilename:\s.*/Output Filename: \"$dir\"/" piripiri.yaml
        cd ..
        i=$(($i+1))
    #done
done


cd "/home/mekk/Documents/Peridigm de Sousa/Simulations/NonCorrPlasticitySims/"
export count=1
echo "Commencing Simulations"
export imax=$i
export float=$(($imax/4)) # Adjust this to the amount of processors
export increment=${float%.*}
for j in 0 1 2 3   # Loop on the amount of processors above, starting from 0.
do
    Perirun $(($j*$increment+1)) $((($j+1)*$increment)) $(($j+2))  &
    sleep 5
done
wait
echo "All simulations done."
```

```
#!/bin/sh

############################################################
#----------General comments regarding simulations---------#
############################################################



# --------------------------------------------------------------------------- #
#       Below are different directories which can be referenced due to slurm.
# Variable              Addressed Volume
# SNIC_TMP              node local disk, copy your input data here and start your
program from here
# TMPDIR                node local disk, use this environment variable to locate a
disk volume for temporary scratch space.
#                       If your application follows that convention nothing needs
to be done.
# SLURM_SUBMIT_DIR      submission directory where you ran sbatch

#       The directory specifier I define myself
# SIM_DIR               Directory to copy to and from on the permanent storage.

# --------------------------------------------------------------------------- #



# --------------------------------------------------------------------------- #
#   If you want to copy a directory from machine a to b while logged into a:
#       scp -r /path/to/directory user@ipaddress:/path/to/destination

#   If you want to copy a directory from machine a to b while logged into b:
#       scp -r user@ipaddress:/path/to/directory /path/to/destination

#   Copy directory from desktop to correct cluster server directory
#       scp ~/Documents/Peridigm/Kluster/Directory tdesousa@aurora.lunarc.lu.se:~/
simFolders

#   Copy this submission file to the cluster server. Should be one level above the
simulation directory
#       scp ~/Documents/Peridigm/Kluster/jobsub.sh tdesousa@aurora.lunarc.lu.se:~/


#   Copy multiple necessary files existing in simFolders from the cluster server
to current folder on desktop
#       scp tdesousa@aurora.lunarc.lu.se:\{~/simFolders/*.e*,~/simFolders/*.log,~/
simFolders/*.yaml\} .
# --------------------------------------------------------------------------- #



# --------------------------------------------------------------------------- #
#   Find and copy yaml files to external drive
#       find . -name "*.yaml" -exec sh -c 'cp "$1" "/media/peridyna/Middagsbox/
Kluster/$1"' _ {} \;
# --------------------------------------------------------------------------- #



# --------------------------------------------------------------------------- #
#   Find and rename png files in all subdirectories to "subdirectory/subdirectory-
filename"
#       find . -name "*.png" -exec sh -c 'mv "$1" "${1%/*}/${1%/*}-${1##*/}"' _ {}
\;
# --------------------------------------------------------------------------- #



# --------------------------------------------------------------------------- #
# Basic time calculation:
#  KNOWN
#   t_known   = Previous experimental simulation time.
#   c_known   = Previous experimental simulation number of cpu cores.
#   c_cluster = Desired amount of cpu cores on the cluster.

#  UNKNOWN
#   x         = Simulation constant.
#   t_cluster = Time it will take on the cluster.

#  FINDING x
#   x/c_known = t_known => x = t_known*c_known

#  FINDING t_cluster
#   t_cluster = x/c_cluster = t_known* (c_known/c_cluster)

#  EXAMPLE
#   t_known   = 9 hours
#   c_known   = 8 cpu cores
#   c_cluster = 20 cpu cores

#   x = 9*8 = 72 cpu-core-hours
#   t_cluster = 72/20 = 3.60 hours = 3 hours + 60*0.6 minutes + 60*0.0 seconds  =
03:36:00 hh/mm/ss needed
#   Probably worth to round to nearest hour
# --------------------------------------------------------------------------- #



############################################################
#------------------SBATCH Specifications------------------#
############################################################
# Simulation Name
#SBATCH -J TPeridigmLONG

# Time required on format hh:mm:ss
#SBATCH -t 148:13:37

# Project is an LU project
#SBATCH -p lu

# The project. In this case: FE simulation of manifacturing processes (LU
2019/2-38)
#SBATCH -A lu2019-2-38

# I want to be notified when the state of the simulation changes: BEGIN, END,
FAIL, REQUEUE, ALL
#SBATCH --mail-user=mas14tde@student.lu.se
#SBATCH --mail-type=ALL

# Exclusive access to node
#SBATCH --exclusive

# Number of nodes
#SBATCH -N 1

# Amount of cores: 20 tasks maximum per node
#SBATCH --tasks-per-node=20

# The %j in the file name will be replaced by the jobnumber SLURM assigns to your
job.
#SBATCH -o periOut_%j.out
#SBATCH -e periErr_%j.err

#SBATCH --no-requeue


############################################################
```

```
#---------------------Job Excecution---------------------#
############################################################
# write this script to stdout-file - useful for scripting errors. Place after last
#SBATCH
cat $0



echo 'Loading Peridigm module'
# Reset the models
module purge
# Loading the virtual peridigm module
module load Peridigm/virtual-1.5.0


echo 'Copy necessary files to node-side'

# Specify which directory to run.
SIM_DIR="SimFullDomain0"
cp -p -R $SLURM_SUBMIT_DIR/$SIM_DIR $SNIC_TMP
cd $SNIC_TMP/$SIM_DIR
#Remove the previous log since we append the file rather than overwrite
rm -f periLog.log

Perirun (){
    peri-virt mpirun -np 20 Peridigm piripiri.yaml >periLog.log 2>&1; touch
periDone.txt
}



#Start peridigm and wait
echo 'Starting simulation'
date
Perirun &
sleep 10 #Allows the first output from the log file to be part of first copy

# Ensuring we don't lose data if running out of time
# Comparison: if "peridigm NOT done OR scriptTime LESS THAN maxTime
while [ ! -e periDone.txt ]
do
    echo 'Copying intermittent data.'
        date
    cp -p -R ../$SIM_DIR $SLURM_SUBMIT_DIR
#       The check happens every five minutes
        sleep 300
done

rm -f periDone.txt

# Copy final data to permanent storage
echo 'Copying final iteration to permanent storage'
cp -p -R "../$SIMDIR" $SLURM_SUBMIT_DIR
echo 'Finished'
date
```

E Stand-alone Shell Script for Post-Processing an Exodus-II File

```
#!/usr/bin/env bash

#Write the necessary program files. Input is: "<filename>.e"
echo "Writing program source code"
filename=$1
pwd=$(pwd)

####################################################
#               Paraview data saver              ####################################################
####################################################
echo "# trace generated using paraview version 5.7.0
#
# To ensure correct image size when batch processing, please search
# for and uncomment the line `# renderView*.ViewSize = [*,*]`

#### import the simple module from the paraview
from paraview.simple import *
#### disable automatic camera reset on 'Show'
paraview.simple._DisableFirstRenderCameraReset()

# create a new 'ExodusIIReader'
tensile_teste = ExodusIIReader(FileName=['$pwd/$filename'])
tensile_teste.ElementVariables = []
tensile_teste.PointVariables = []
tensile_teste.GlobalVariables = []
tensile_teste.NodeSetArrayStatus = []

# get animation scene
animationScene1 = GetAnimationScene()

# get the time-keeper
timeKeeper1 = GetTimeKeeper()

# update animation scene based on data timesteps
animationScene1.UpdateAnimationUsingDataTimeSteps()

# Properties modified on tensile_teste
tensile_teste.ElementVariables = ['Block_Id', 'Damage', 'Deformation_GradientX', 'Deformation_GradientY',
'Deformation_GradientZ', 'Element_Id', 'Neighborhood_Volume', 'Number_Of_Neighbors', 'Proc_Num', 'Radius',
'Surface_Correction_Factor', 'Volume']
tensile_teste.PointVariables = ['Coordinates', 'Displacement', 'Force', 'Force_Density', 'Velocity']
tensile_teste.GlobalVariables = ['Gage_Left_Displacement', 'Gage_Left_Initial_Position',
'Gage_Right_Displacement', 'Gage_Right_Initial_Position', 'Right_Reaction_Force', 'Left_Reaction_Force']
tensile_teste.ElementBlocks = ['block_1', 'block_2']
tensile_teste.FilePrefix = ''
tensile_teste.FilePattern = ''

# get active view
renderView1 = GetActiveViewOrCreate('RenderView')
# uncomment following to set a specific view size
# renderView1.ViewSize = [1099, 562]

# show data in view
tensile_testeDisplay = Show(tensile_teste, renderView1)

# trace defaults for the display properties.
tensile_testeDisplay.Representation = 'Surface'
tensile_testeDisplay.ColorArrayName = [None, '']
tensile_testeDisplay.OSPRayScaleArray = 'Coordinates'
tensile_testeDisplay.OSPRayScaleFunction = 'PiecewiseFunction'
tensile_testeDisplay.SelectOrientationVectors = 'Coordinates'
tensile_testeDisplay.ScaleFactor = 0.0004975418793037534
tensile_testeDisplay.SelectScaleArray = 'Coordinates'
tensile_testeDisplay.GlyphType = 'Arrow'
tensile_testeDisplay.GlyphTableIndexArray = 'Coordinates'
tensile_testeDisplay.GaussianRadius = 2.487709396518767e-05
tensile_testeDisplay.SetScaleArray = ['POINTS', 'Coordinates']
tensile_testeDisplay.ScaleTransferFunction = 'PiecewiseFunction'
tensile_testeDisplay.OpacityArray = ['POINTS', 'Coordinates']
tensile_testeDisplay.OpacityTransferFunction = 'PiecewiseFunction'
tensile_testeDisplay.DataAxesGrid = 'GridAxesRepresentation'
tensile_testeDisplay.PolarAxes = 'PolarAxesRepresentation'
tensile_testeDisplay.ScalarOpacityUnitDistance = 0.00012612856299355803
tensile_testeDisplay.ExtractedBlockIndex = 2

# init the 'PiecewiseFunction' selected for 'ScaleTransferFunction'
tensile_testeDisplay.ScaleTransferFunction.Points = [-0.00247539408305662, 0.0, 0.5, 0.0,
0.0025000246167497897, 1.0, 0.5, 0.0]

# init the 'PiecewiseFunction' selected for 'OpacityTransferFunction'
tensile_testeDisplay.OpacityTransferFunction.Points = [-0.00247539408305662, 0.0, 0.5, 0.0,
0.0025000246167497897, 1.0, 0.5, 0.0]

# init the 'GridAxesRepresentation' selected for 'DataAxesGrid'
tensile_testeDisplay.DataAxesGrid.XTitleColor = [0.0, 0.0, 0.0]
tensile_testeDisplay.DataAxesGrid.YTitleColor = [0.0, 0.0, 0.0]
tensile_testeDisplay.DataAxesGrid.ZTitleColor = [0.0, 0.0, 0.0]
tensile_testeDisplay.DataAxesGrid.XLabelColor = [0.0, 0.0, 0.0]
tensile_testeDisplay.DataAxesGrid.YLabelColor = [0.0, 0.0, 0.0]
tensile_testeDisplay.DataAxesGrid.ZLabelColor = [0.0, 0.0, 0.0]

# init the 'PolarAxesRepresentation' selected for 'PolarAxes'
tensile_testeDisplay.PolarAxes.PolarAxisTitleColor = [0.0, 0.0, 0.0]
tensile_testeDisplay.PolarAxes.PolarAxisLabelColor = [0.0, 0.0, 0.0]
tensile_testeDisplay.PolarAxes.LastRadialAxisTextColor = [0.0, 0.0, 0.0]
tensile_testeDisplay.PolarAxes.SecondaryRadialAxesTextColor = [0.0, 0.0, 0.0]

# reset view to fit data
renderView1.ResetCamera()

# get the material library
materialLibrary1 = GetMaterialLibrary()

# update the view to ensure updated data information
renderView1.Update()

# set scalar coloring
ColorBy(tensile_testeDisplay, ('FIELD', 'vtkBlockColors'))

# show color bar/color legend
tensile_testeDisplay.SetScalarBarVisibility(renderView1, True)

# get color transfer function/color map for 'vtkBlockColors'
vtkBlockColorsLUT = GetColorTransferFunction('vtkBlockColors')
vtkBlockColorsLUT.InterpretValuesAsCategories = 1
vtkBlockColorsLUT.AnnotationsInitialized = 1
vtkBlockColorsLUT.Annotations = ['0', '0', '1', '1', '2', '2', '3', '3', '4', '4', '5', '5', '6', '6',
'7', '7', '8', '8', '9', '9', '10', '10', '11', '11']
vtkBlockColorsLUT.ActiveAnnotatedValues = ['0', '1']
vtkBlockColorsLUT.IndexedColors = [1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0,
0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.63, 0.63, 1.0, 0.67, 0.5, 0.33, 1.0, 0.5, 0.75, 0.53, 0.35, 0.7, 1.0,
0.75, 0.5]

# get opacity transfer function/opacity map for 'vtkBlockColors'
vtkBlockColorsPWF = GetOpacityTransferFunction('vtkBlockColors')

# create a new 'Merge Blocks'
mergeBlocks1 = MergeBlocks(Input=tensile_teste)

# show data in view
mergeBlocks1Display = Show(mergeBlocks1, renderView1)

# trace defaults for the display properties.
mergeBlocks1Display.Representation = 'Surface'
mergeBlocks1Display.ColorArrayName = [None, '']
mergeBlocks1Display.OSPRayScaleArray = 'Coordinates'
mergeBlocks1Display.OSPRayScaleFunction = 'PiecewiseFunction'
mergeBlocks1Display.SelectOrientationVectors = 'Coordinates'
mergeBlocks1Display.ScaleFactor = 0.0004975418793037534
mergeBlocks1Display.SelectScaleArray = 'Coordinates'
mergeBlocks1Display.GlyphType = 'Arrow'
mergeBlocks1Display.GlyphTableIndexArray = 'Coordinates'
mergeBlocks1Display.GaussianRadius = 2.487709396518767e-05
mergeBlocks1Display.SetScaleArray = ['POINTS', 'Coordinates']
mergeBlocks1Display.ScaleTransferFunction = 'PiecewiseFunction'
mergeBlocks1Display.OpacityArray = ['POINTS', 'Coordinates']
mergeBlocks1Display.OpacityTransferFunction = 'PiecewiseFunction'
mergeBlocks1Display.DataAxesGrid = 'GridAxesRepresentation'
mergeBlocks1Display.PolarAxes = 'PolarAxesRepresentation'
mergeBlocks1Display.ScalarOpacityUnitDistance = 0.00012612856299355803

# init the 'PiecewiseFunction' selected for 'ScaleTransferFunction'
mergeBlocks1Display.ScaleTransferFunction.Points = [-0.00247539408305662, 0.0, 0.5, 0.0,
0.0025000246167497897, 1.0, 0.5, 0.0]

# init the 'PiecewiseFunction' selected for 'OpacityTransferFunction'
mergeBlocks1Display.OpacityTransferFunction.Points = [-0.00247539408305662, 0.0, 0.5, 0.0,
0.0025000246167497897, 1.0, 0.5, 0.0]

# init the 'GridAxesRepresentation' selected for 'DataAxesGrid'
mergeBlocks1Display.DataAxesGrid.XTitleColor = [0.0, 0.0, 0.0]
mergeBlocks1Display.DataAxesGrid.YTitleColor = [0.0, 0.0, 0.0]
mergeBlocks1Display.DataAxesGrid.ZTitleColor = [0.0, 0.0, 0.0]
mergeBlocks1Display.DataAxesGrid.XLabelColor = [0.0, 0.0, 0.0]
mergeBlocks1Display.DataAxesGrid.YLabelColor = [0.0, 0.0, 0.0]
mergeBlocks1Display.DataAxesGrid.ZLabelColor = [0.0, 0.0, 0.0]

# init the 'PolarAxesRepresentation' selected for 'PolarAxes'
mergeBlocks1Display.PolarAxes.PolarAxisTitleColor = [0.0, 0.0, 0.0]
mergeBlocks1Display.PolarAxes.PolarAxisLabelColor = [0.0, 0.0, 0.0]
mergeBlocks1Display.PolarAxes.LastRadialAxisTextColor = [0.0, 0.0, 0.0]
mergeBlocks1Display.PolarAxes.SecondaryRadialAxesTextColor = [0.0, 0.0, 0.0]

# hide data in view
Hide(tensile_teste, renderView1)

# update the view to ensure updated data information
renderView1.Update()
```

```
animationScene1.GoToLast()

# save data
SaveData('$pwd//FieldData.csv', proxy=mergeBlocks1, UseScientificNotation=1,
    FieldAssociation='Field Data')

# save data
SaveData('$pwd/CellData.csv', proxy=mergeBlocks1, WriteTimeSteps=1,
    UseScientificNotation=1,
    FieldAssociation='Cells')

# save data
SaveData('$pwd/PointData.csv', proxy=mergeBlocks1, WriteTimeSteps=1,
    UseScientificNotation=1)

# set scalar coloring
ColorBy(mergeBlocks1Display, ('CELLS', 'Damage'))

# rescale color and/or opacity maps used to include current data range
mergeBlocks1Display.RescaleTransferFunctionToDataRange(True, False)

# show color bar/color legend
mergeBlocks1Display.SetScalarBarVisibility(renderView1, True)

# get color transfer function/color map for 'Damage'
damageLUT = GetColorTransferFunction('Damage')
damageLUT.RGBPoints = [0.0, 0.231373, 0.298039, 0.752941, 0.039473684210526314, 0.865003, 0.865003,
0.865003, 0.07894736842105263, 0.705882, 0.0156863, 0.14902]
damageLUT.ScalarRangeInitialized = 1.0

# get opacity transfer function/opacity map for 'Damage'
damagePWF = GetOpacityTransferFunction('Damage')
damagePWF.Points = [0.0, 0.0, 0.5, 0.0, 0.07894736842105263, 1.0, 0.5, 0.0]
damagePWF.ScalarRangeInitialized = 1

# Properties modified on mergeBlocks1Display
mergeBlocks1Display.PointSize = 10.0

# Properties modified on mergeBlocks1Display
mergeBlocks1Display.RenderPointsAsSpheres = 1

# get color legend/bar for damageLUT in view renderView1
damageLUTColorBar = GetScalarBar(damageLUT, renderView1)
damageLUTColorBar.Title = 'Damage'
damageLUTColorBar.ComponentTitle = ''
damageLUTColorBar.TitleColor = [0.0, 0.0, 0.0]
damageLUTColorBar.LabelColor = [0.0, 0.0, 0.0]

# change scalar bar placement
damageLUTColorBar.Orientation = 'Horizontal'
damageLUTColorBar.WindowLocation = 'AnyLocation'
damageLUTColorBar.Position = [0.3422020018198361, 0.0]
damageLUTColorBar.ScalarBarLength = 0.32999999999999946

# current camera placement for renderView1
renderView1.CameraPosition = [1.2315227650105953e-05, 4.926128895021975e-06, 0.005539835654097468]
renderView1.CameraFocalPoint = [1.2315227650105953e-05, 4.926128895021975e-06, 1.0837444278877228e-05]
renderView1.CameraParallelScale = 0.0025351215722264845

# save screenshot
SaveScreenshot('/home/peridyna/Documents/Peridigm/Simulations/Verifications/Kluster/CG_FD_NC/DmgSS.png',
renderView1, ImageResolution=[1099, 562],
    TransparentBackground=1)

# set scalar coloring
ColorBy(mergeBlocks1Display, ('POINTS', 'Force', 'Magnitude'))

# Hide the scalar bar for this color map if no visible data is colored by it.
HideScalarBarIfNotNeeded(damageLUT, renderView1)

# rescale color and/or opacity maps used to include current data range
mergeBlocks1Display.RescaleTransferFunctionToDataRange(True, False)

# show color bar/color legend
mergeBlocks1Display.SetScalarBarVisibility(renderView1, True)

# get color transfer function/color map for 'Force'
forceLUT = GetColorTransferFunction('Force')
forceLUT.RGBPoints = [2.0218012240182936e-05, 0.231373, 0.298039, 0.752941, 4812.927562062904, 0.865003,
0.865003, 0.865003, 9625.855103907796, 0.705882, 0.0156863, 0.14902]
forceLUT.ScalarRangeInitialized = 1.0

# get opacity transfer function/opacity map for 'Force'
forcePWF = GetOpacityTransferFunction('Force')
forcePWF.Points = [2.0218012240182936e-05, 0.0, 0.5, 0.0, 9625.855103907796, 1.0, 0.5, 0.0]
forcePWF.ScalarRangeInitialized = 1

# set scalar coloring
```

```
ColorBy(mergeBlocks1Display, ('POINTS', 'Force_Density', 'Magnitude'))

# Hide the scalar bar for this color map if no visible data is colored by it.
HideScalarBarIfNotNeeded(forceLUT, renderView1)

# rescale color and/or opacity maps used to include current data range
mergeBlocks1Display.RescaleTransferFunctionToDataRange(True, False)

# show color bar/color legend
mergeBlocks1Display.SetScalarBarVisibility(renderView1, True)

# get color transfer function/color map for 'Force_Density'
force_DensityLUT = GetColorTransferFunction('Force_Density')
force_DensityLUT.RGBPoints = [33326.56265622793, 0.231373, 0.298039, 0.752941, 7933437276202.003,
0.865003, 0.865003, 0.865003, 15866874519077.443, 0.705882, 0.0156863, 0.14902]
force_DensityLUT.ScalarRangeInitialized = 1.0

# get opacity transfer function/opacity map for 'Force_Density'
force_DensityPWF = GetOpacityTransferFunction('Force_Density')
force_DensityPWF.Points = [33326.56265622793, 0.0, 0.5, 0.0, 15866874519077.443, 1.0, 0.5, 0.0]
force_DensityPWF.ScalarRangeInitialized = 1

# get color legend/bar for force_DensityLUT in view renderView1
force_DensityLUTColorBar = GetScalarBar(force_DensityLUT, renderView1)
force_DensityLUTColorBar.Title = 'Force_Density'
force_DensityLUTColorBar.ComponentTitle = 'Magnitude'
force_DensityLUTColorBar.TitleColor = [0.0, 0.0, 0.0]
force_DensityLUTColorBar.LabelColor = [0.0, 0.0, 0.0]

# change scalar bar placement
force_DensityLUTColorBar.Orientation = 'Horizontal'
force_DensityLUTColorBar.WindowLocation = 'AnyLocation'
force_DensityLUTColorBar.Position = [0.37586897179253853, 0.15149466192170827]
force_DensityLUTColorBar.ScalarBarLength = 0.3300000000000003

# change scalar bar placement
force_DensityLUTColorBar.Position = [0.36131028207461313, 0.0]

# current camera placement for renderView1
renderView1.CameraPosition = [1.2315227650105953e-05, 4.926128895021975e-06, 0.005539835654097468]
renderView1.CameraFocalPoint = [1.2315227650105953e-05, 4.926128895021975e-06, 1.0837444278877228e-05]
renderView1.CameraParallelScale = 0.0025351215722264845

# save screenshot
SaveScreenshot('/home/peridyna/Documents/Peridigm/Simulations/Verifications/Kluster/CG_FD_NC/FDSS.png',
renderView1, ImageResolution=[1099, 562],
    TransparentBackground=1)

#### saving camera placements for all active views

# current camera placement for renderView1
renderView1.CameraPosition = [1.2315227650105953e-05, 4.926128895021975e-06, 0.005539835654097468]
renderView1.CameraFocalPoint = [1.2315227650105953e-05, 4.926128895021975e-06, 1.0837444278877228e-05]
renderView1.CameraParallelScale = 0.0025351215722264845

#### uncomment the following to render all views
# RenderAllViews()
# alternatively, if you want to write images, you can use SaveScreenshot(...).">pv_SS_data_script.py
####################################################
#               natsortfiles.m                  ####################################################
####################################################
echo "function [X,ndx,dbg] = natsortfiles(X,rgx,varargin)
% Alphanumeric / Natural-Order sort of a cell array of filename/filepath strings (1xN char).
%
% (c) 2014-2019 Stephen Cobeldick
%
% Alphanumeric sort of a cell array of filenames or filepaths: sorts by
% character order and also by the values of any numbers that are within
% the names. Filenames, file-extensions, and directories (if supplied)
% are split apart and are sorted separately: this ensures that shorter
% filenames sort before longer ones (i.e. thus giving a dictionary sort).
%
%%% Example:
% P = 'C:\SomeDir\SubDir';
% S = dir(fullfile(P,'*.txt'));
% C = natsortfiles({S.name});
% for k = 1:numel(C)
%     fullfile(P,C{k})
% end
%
%%% Syntax:
%  Y = natsortfiles(X)
%  Y = natsortfiles(X,rgx)
%  Y = natsortfiles(X,rgx,<options>)
% [Y,ndx,dbg] = natsortfiles(X,...)
%
% To sort all of the strings in a cell array use NATSORT (File Exchange 34464).
% To sort the rows of a cell array of strings use NATSORTROWS (File Exchange 47433).
%
```

E Stand-alone Shell Script for Post-Processing an Exodus-II File

```
%% File Dependency %%
%
% NATSORTFILES requires the function NATSORT (File Exchange 34464). The optional
% arguments <options> are passed directly to NATSORT. See NATSORT for case
% sensitivity, sort direction, numeric substring matching, and other options.
%
%% Explanation %%
%
% Using SORT on filenames will sort any of char(0:45), including the printing
% characters ' !\"#$%&''()*+,-', before the file extension separator character '.'.
% Therefore this function splits the name and extension and sorts them separately.
%
% Similarly the file separator character within filepaths can cause longer
% directory names to sort before shorter ones, as char(0:46)<'/' and
% char(0:91)<'\'. Check this example to see why this matters:
%
% >> X = {'A1\B', 'A+/B', 'A\B', 'A=/B', 'A/B'};
% >> sort(X)
% ans =   'A+/B'  'A/B'  'A1\B'  'A=/B'  'A\B'
% >> natsortfiles(X)
% ans =   'A\B'  'A/B'  'A1\B'  'A+/B'  'A=/B'
%
% NATSORTFILES splits filepaths at each file separator character and sorts
% every level of the directory hierarchy separately, ensuring that shorter
% directory names sort before longer, regardless of the characters in the names.
%
%% Examples %%
%
% >> A = {'a2.txt', 'a10.txt', 'a1.txt'};
% >> sort(A)
% ans = 'a1.txt'  'a10.txt'  'a2.txt'
% >> natsortfiles(A)
% ans = 'a1.txt'  'a2.txt'  'a10.txt'
%
% >> B = {'test_new.m'; 'test-old.m'; 'test.m'};
% >> sort(B) % Note '-' sorts before '.':
% ans =
%    'test-old.m'
%    'test.m'
%    'test_new.m'
% >> natsortfiles(B) % Shorter names before longer (dictionary sort):
% ans =
%    'test.m'
%    'test-old.m'
%    'test_new.m'
%
% >> C = {'test2.m'; 'test10-old.m'; 'test.m'; 'test10.m'; 'test1.m'};
% >> sort(C) % Wrong numeric order:
% ans =
%    'test.m'
%    'test1.m'
%    'test10-old.m'
%    'test10.m'
%    'test2.m'
% >> natsortfiles(C) % Shorter names before longer:
% ans =
%    'test.m'
%    'test1.m'
%    'test2.m'
%    'test10.m'
%    'test10-old.m'
%
%%% Directory Names:
% >> D = {'A2-old\test.m';'A10\test.m';'A2\test.m';'A1archive.zip';'A1\test.m'};
% >> sort(D) % Wrong numeric order, and '-' sorts before '\':
% ans =
%    'A10\test.m'
%    'A1\test.m'
%    'A1archive.zip'
%    'A2-old\test.m'
%    'A2\test.m'
% >> natsortfiles(D) % Shorter names before longer (dictionary sort):
% ans =
%    'A1archive.zip'
%    'A1\test.m'
%    'A2\test.m'
%    'A2-old\test.m'
%    'A10\test.m'
%
%% Input and Output Arguments %%
%
%%% Inputs (*=default):
% X   = CellArrayOfCharRowVectors, with filenames or filepaths to be sorted.
% rgx = Regular expression to match number substrings, '\d+'*
%     = [] uses the default regular expression, which matches integers.
% <options> can be supplied in any order and are passed directly to NATSORT.
%
%%% Outputs:
% Y   = CellArrayOfCharRowVectors, filenames of <X> sorted into natural-order.
% ndx = NumericMatrix, same size as <X>. Indices such that Y = X(ndx).
% dbg = CellVectorOfCellArrays, size 1xMAX(2+NumberOfDirectoryLevels).
%       Each cell contains the debug cell array for directory names, filenames,
%       and file extensions. Helps debug the regular expression. See NATSORT.
%
% See also SORT NATSORT NATSORTROWS DIR FILEPARTS FULLFILE NEXTNAME CELLSTR REGEXP IREGEXP SSCANF

%% Input Wrangling %%
%
assert(iscell(X),'First input <X> must be a cell array.')
tmp = cellfun('isclass',X,'char') & cellfun('size',X,1)<2 & cellfun('ndims',X)<3;
assert(all(tmp(:)),'First input <X> must be a cell array of strings (1xN character).')
%
if nargin>1
    varargin = [{rgx},varargin];
end
%
%% Split and Sort File Names/Paths %%
%
% Split full filepaths into file [path,name,extension]:
[pth,fnm,ext] = cellfun(@fileparts,X(:),'UniformOutput',false);
% Split path into {dir,subdir,subsubdir,...}:
pth = regexp(pth,'[^/\]+','match'); % either / or \ as filesep.
len = cellfun('length',pth);
num = max(len);
vec = cell(numel(len),1);
%
% Natural-order sort of the file extensions and filenames:
if isempty(num)
    ndx = [];
    ids = [];
    dbg = {};
elseif nargout<3 % faster:
    [~,ndx] = natsort(ext,varargin{:});
    [~,ids] = natsort(fnm(ndx),varargin{:});
else % for debugging:
    [~,ndx,dbg{num+2}] = natsort(ext,varargin{:});
    [~,ids,tmp] = natsort(fnm(ndx),varargin{:});
    [~,idd] = sort(ndx);
    dbg{num+1} = tmp(idd,:);
end
ndx = ndx(ids);
%
% Natural-order sort of the directory names:
for k = num:-1:1
    idx = len>=k;
    vec(:) = {''};
    vec(idx) = cellfun(@(c)c(k),pth(idx));
    if nargout<3 % faster:
        [~,ids] = natsort(vec(ndx),varargin{:});
    else % for debugging:
        [~,ids,tmp] = natsort(vec(ndx),varargin{:});
        [~,idd] = sort(ndx);
        dbg{k} = tmp(idd,:);
    end
    ndx = ndx(ids);
end
%
% Return the sorted array and indices:
ndx = reshape(ndx,size(X));
X = X(ndx);
%
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%natsortfiles
">natsortfiles.m
echo "
function [X,ndx,dbg] = natsort(X,rgx,varargin)
% Alphanumeric / Natural-Order sort the strings in a cell array of strings (1xN char).
%
% (c) 2012-2019 Stephen Cobeldick
%
% Alphanumeric sort a cell array of strings: sorts by character order and
% also by the values of any number substrings. Default: match all integer
% number substrings and perform a case-insensitive ascending sort.
%
%%% Example:
% >> X = {'x2', 'x10', 'x1'};
% >> sort(X)
% ans =   'x1'  'x10'  'x2'
% >> natsort(X)
% ans =   'x1'  'x2'  'x10'
%
%%% Syntax:
%  Y = natsort(X)
%  Y = natsort(X,rgx)
%  Y = natsort(X,rgx,<options>)
% [Y,ndx,dbg] = natsort(X,...)
%
% To sort filenames or filepaths use NATSORTFILES (FEX 47434).
% To sort the rows of a cell array of strings use NATSORTROWS (FEX 47433).
%
%% Number Substrings %%
%
% By default consecutive digit characters are interpreted as an integer.
% Specifying the optional regular expression pattern allows the numbers to
% include a +/- sign, decimal digits, exponent E-notation, quantifiers,
% or look-around matching. For information on defining regular expressions:
% http://www.mathworks.com/help/matlab/matlab_prog/regular-expressions.html
%
% The number substrings are parsed by SSCANF into numeric values, using
% either the *default format '%f' or the user-supplied format specifier.
%
% This table shows examples of regular expression patterns for some common
% notations and ways of writing numbers, with suitable SSCANF formats:
%
% Regular       | Number Substring | Number Substring              | SSCANF
% Expression:   | Match Examples:  | Match Description:            | Format Specifier:
% ==============|==================|===============================|==================
% *         \d+ | 0, 123, 4, 56789 | unsigned integer              | %f  %i  %u  %lu
% --------------|------------------|-------------------------------|------------------
%      [-+]?\d+ | +1, 23, -45, 678 | integer with optional +/- sign| %f  %i  %d  %ld
% --------------|------------------|-------------------------------|------------------
%     \d+\.?\d* | 012, 3.45, 678.9 | integer or decimal            | %f
% (\d+|Inf|NaN) | 123, 4, NaN, Inf | integer, Inf, or NaN          | %f
%  \d+\.\d+e\d+ | 0.123e4, 5.67e08 | exponential notation          | %f
% --------------|------------------|-------------------------------|------------------
%  0[0-7]+      | 012, 03456, 0700 | octal notation & prefix       | %o  %i
%   [0-7]+      |  12,  3456,  700 | octal notation                | %o
% --------------|------------------|-------------------------------|------------------
%  0X[0-9A-F]+  | 0X0, 0X3E7, 0XFF | hexadecimal notation & prefix | %x  %i
%    [0-9A-F]+  |   0,   3E7,   FF | hexadecimal notation          | %x
% --------------|------------------|-------------------------------|------------------
%  0B[01]+      | 0B1, 0B101, 0B10 | binary notation & prefix      | %b   (not SSCANF)
%    [01]+      |   1,   101,   10 | binary notation               | %b   (not SSCANF)
% --------------|------------------|-------------------------------|------------------
%
%% Debugging Output Array %%
%
% The third output is a cell array <dbg>, to check if the numbers have
% been matched by the regular expression <rgx> and converted to numeric
% by the SSCANF format. The rows of <dbg> are linearly indexed from <X>,
% even columns contain numbers, odd columns contain split substrings:
%
% >> [~,~,dbg] = natsort(X)
% dbg =
%    'x'    [ 2]
%    'x'    [10]
%    'x'    [ 1]
%
%% Examples %%
%
%%% Multiple integers (e.g. release version numbers):
% >> A = {'v10.6', 'v9.10', 'v9.5', 'v10.10', 'v9.10.20', 'v9.10.8'};
% >> sort(A)
% ans =   'v10.10'  'v10.6'  'v9.10'  'v9.10.20'  'v9.10.8'  'v9.5'
% >> natsort(A)
% ans =   'v9.5'  'v9.10'  'v9.10.8'  'v9.10.20'  'v10.6'  'v10.10'
%
%%% Integer, decimal, NaN, or Inf numbers, possibly with +/- signs:
% >> B = {'test+NaN', 'test11.5', 'test-1.4', 'test', 'test-Inf', 'test+0.3'};
% >> sort(B)
% ans =   'test' 'test+0.3' 'test+NaN' 'test-1.4' 'test-Inf' 'test11.5'
% >> natsort(B, '[-+]?(NaN|Inf|\d+\.?\d*)')
% ans =   'test' 'test-Inf' 'test-1.4' 'test+0.3' 'test11.5' 'test+NaN'
%
%%% Integer or decimal numbers, possibly with an exponent:
% >> C = {'0.56e007', '', '43E-2', '10000', '9.8'};
% >> sort(C)
% ans =   ''  '0.56e007'  '10000'  '43E-2'  '9.8'
% >> natsort(C, '\d+\.?\d*([eE][-+]?\d+)?')
% ans =   ''  '43E-2'  '9.8'  '10000'  '0.56e007'
%
%%% Hexadecimal numbers (with '0X' prefix):
% >> D = {'a0X7C4z', 'a0X5z', 'a0X18z', 'a0XFz'};
% >> sort(D)
% ans =   'a0X18z'  'a0X5z'  'a0X7C4z'  'a0XFz'
% >> natsort(D, '0X[0-9A-F]+', '%i')
% ans =   'a0X5z'  'a0XFz'  'a0X18z'  'a0X7C4z'
%
%%% Binary numbers:
% >> E = {'a11111000100z', 'a101z', 'a000000000011000z', 'a1111z'};
% >> sort(E)
% ans =   'a000000000011000z'  'a101z'  'a11111000100z'  'a1111z'
% >> natsort(E, '[01]+', '%b')
% ans =   'a101z'  'a1111z'  'a000000000011000z'  'a11111000100z'
%
%%% Case sensitivity:
% >> F = {'a2', 'A20', 'A1', 'a10', 'A2', 'a1'};
% >> natsort(F, [], 'ignorecase') % default
% ans =   'A1'  'a1'  'a2'  'A2'  'a10'  'A20'
% >> natsort(F, [], 'matchcase')
% ans =   'A1'  'A2'  'A20'  'a1'  'a2'  'a10'
%
%%% Sort order:
% >> G = {'2', 'a', '', '3', 'B', '1'};
% >> natsort(G, [], 'ascend') % default
% ans =   ''   '1'  '2'  '3'  'a'  'B'
% >> natsort(G, [], 'descend')
% ans =   'B'  'a'  '3'  '2'  '1'  ''
% >> natsort(G, [], 'num<char') % default
% ans =   ''   '1'  '2'  '3'  'a'  'B'
% >> natsort(G, [], 'char<num')
% ans =   ''   'a'  'B'  '1'  '2'  '3'
%
%%% UINT64 numbers (with full precision):
% >> natsort({'a18446744073709551615z', 'a18446744073709551614z'}, [], '%lu')
% ans =       'a18446744073709551614z'  'a18446744073709551615z'
%
%% Input and Output Arguments %%
%
%%% Inputs (*==default):
% X   = CellArrayOfCharRowVectors, to be sorted into natural-order.
% rgx = Regular expression to match number substrings, '\d+'*
%     = [] uses the default regular expression, which matches integers.
% <options> can be entered in any order, as many as required:
%     = Sort direction: 'descend'/'ascend'*
%     = NaN/number order: 'NaN<num'/'num<NaN'*
%     = Character/number order: 'char<num'/'num<char'*
%     = Character case handling: 'matchcase'/'ignorecase'*
%     = SSCANF number conversion format, e.g.: '%f'*, '%x', '%li', '%b', etc.
%
%%% Outputs:
% Y   = CellArrayOfCharRowVectors, <X> sorted into natural-order.
% ndx = NumericArray, such that Y = X(ndx). The same size as <X>.
% dbg = CellArray of the parsed characters and number values.
%       Each row is one input char vector, linear-indexed from <X>.
%
% See also SORT NATSORTFILES NATSORTROWS CELLSTR REGEXP IREGEXP SSCANF

%% Input Wrangling %%
%
assert(iscell(X),'First input <X> must be a cell array.')
tmp = cellfun('isclass',X,'char') & cellfun('size',X,1)<2 & cellfun('ndims',X)<3;
assert(all(tmp(:)),'First input <X> must be a cell array of char row vectors (1xN char).')
%
if nargin<2 || isnumeric(rgx)&&isempty(rgx)
    rgx = '\d+';
else
    assert(ischar(rgx)&&ndims(rgx)<3&&size(rgx,1)==1,...
        'Second input <rgx> must be a regular expression (char row vector).') %#ok<ISMAT>
end
%
% Optional arguments:
tmp = cellfun('isclass',varargin,'char') & cellfun('size',varargin,1)<2 & cellfun('ndims',varargin)<3;
assert(all(tmp(:)),'All optional arguments must be char row vectors (1xN char).')
% Character case:
ccm = strcmpi(varargin,'matchcase');
ccx = strcmpi(varargin,'ignorecase')|ccm;
% Sort direction:
sdd = strcmpi(varargin,'descend');
sdx = strcmpi(varargin,'ascend')|sdd;
% Char/num order:
chb = strcmpi(varargin,'char<num');
chx = strcmpi(varargin,'num<char')|chb;
% NaN/num order:
nab = strcmpi(varargin,'NaN<num');
nax = strcmpi(varargin,'num<NaN')|nab;
% SSCANF format:
sfx = ~cellfun('isempty',regexp(varargin,'^%([bdiuoxfeg]|l[diuox])$'));
%
nsAssert(1,varargin,sdx,'Sort direction')
nsAssert(1,varargin,chx,'Char<->num')
nsAssert(1,varargin,nax,'NaN<->num')
nsAssert(1,varargin,sfx,'SSCANF format')
nsAssert(0,varargin,~(ccx|sdx|chx|nax|sfx))
%
% SSCANF format:
if nnz(sfx)
    fmt = varargin{sfx};
    if strcmpi(fmt,'%b')
        cls = 'double';
    else
        cls = class(sscanf('0',fmt));
    end
else
    fmt = '%f';
    cls = 'double';
```

```
end
%
%% Identify Numbers %%
%
[mat,spl] = regexpi(X(:),rgx,'match','split',varargin{ccx});
%
% Determine lengths:
nmx = numel(X);
nmn = cellfun('length',mat);
nms = cellfun('length',spl);
mxs = max(nms);
%
% Preallocate arrays:
bon = bsxfun(@le,1:mxs,nmn).';
bos = bsxfun(@le,1:mxs,nms).';
arn = zeros(mxs,nmx,cls);
ars =  cell(mxs,nmx);
ars(:) = {''};
ars(bos) = [spl{:}];
%
%% Convert Numbers to Numeric %%
%
if nmx
    tmp = [mat{:}];
    if strcmp(fmt,'%b')
        tmp = regexprep(tmp,'^0[Bb]','');
        vec = cellfun(@(s)sum(pow2(s-'0',numel(s)-1:-1:0)),tmp);
    else
        vec = sscanf(sprintf(' %s',tmp{:}),fmt);
    end
    assert(numel(vec)==numel(tmp),'The %s format must return one value for each input number.',fmt)
else
    vec = [];
end
%
%% Debugging Array %%
%
if nmx && nargout>2
    dbg = cell(mxs,nmx);
    dbg(:) = {''};
    dbg(bon) = num2cell(vec);
    dbg = reshape(permute(cat(3,ars,dbg),[3,1,2]),[],nmx).';
    idf = [find(~all(cellfun('isempty',dbg),1),1,'last'),1];
    dbg = dbg(:,1:idf(1));
else
    dbg = {};
end
%
%% Sort Columns %%
%
if ~any(ccm) % ignorecase
    ars = lower(ars);
end
%
if nmx && any(chb) % char<num
    boe = ~cellfun('isempty',ars(bon));
    for k = reshape(find(bon),1,[])
        ars{k}(end+1) = char(65535);
    end
    [idr,idc] = find(bon);
    idn = sub2ind(size(bon),boe(:)+idr(:),idc(:));
    bon(:) = false;
    bon(idn) = true;
    arn(idn) = vec;
    bon(isnan(arn)) = ~any(nab);
    ndx = 1:nmx;
    if any(sdd) % descending
        for k = mxs:-1:1
            [~,idx] = sort(nsGroup(ars(k,ndx)),'descend');
            ndx = ndx(idx);
            [~,idx] = sort(arn(k,ndx),'descend');
            ndx = ndx(idx);
            [~,idx] = sort(bon(k,ndx),'descend');
            ndx = ndx(idx);
        end
    else % ascending
        for k = mxs:-1:1
            [~,idx] = sort(ars(k,ndx));
            ndx = ndx(idx);
            [~,idx] = sort(arn(k,ndx),'ascend');
            ndx = ndx(idx);
            [~,idx] = sort(bon(k,ndx),'ascend');
            ndx = ndx(idx);
        end
    end
else % num<char
    arn(bon) = vec;
    bon(isnan(arn)) = ~any(nab);
    if any(sdd) % descending
```

```
        [~,ndx] = sort(nsGroup(ars(mxs,:)),'descend');
        for k = mxs-1:-1:1
            [~,idx] = sort(arn(k,ndx),'descend');
            ndx = ndx(idx);
            [~,idx] = sort(bon(k,ndx),'descend');
            ndx = ndx(idx);
            [~,idx] = sort(nsGroup(ars(k,ndx)),'descend');
            ndx = ndx(idx);
        end
    else % ascending
        [~,ndx] = sort(ars(mxs,:));
        for k = mxs-1:-1:1
            [~,idx] = sort(arn(k,ndx),'ascend');
            ndx = ndx(idx);
            [~,idx] = sort(bon(k,ndx),'ascend');
            ndx = ndx(idx);
            [~,idx] = sort(ars(k,ndx));
            ndx = ndx(idx);
        end
    end
end
%
ndx  = reshape(ndx,size(X));
X = X(ndx);
%
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%natsort
function nsAssert(val,inp,idx,varargin)
% Throw an error if an option is overspecified.
if nnz(idx)>val
    tmp = {'Unknown input arguments',' option may only be specified once. Provided inputs'};
    error('%s:%s',[varargin{:},tmp{1+val}],sprintf('\n''%s''',inp{idx}))
end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%nsAssert
function grp = nsGroup(vec)
% Groups of a cell array of strings, equivalent to [~,~,grp]=unique(vec);
[vec,idx] = sort(vec);
grp = cumsum([true,~strcmp(vec(1:end-1),vec(2:end))]);
grp(idx) = grp;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%nsGroup">natsort.m
#####################################################
#               dataInspector.m                   #####################################################
#####################################################
echo "
clear;
close;
clc;
csvfiles        = strcat(pwd, '/Field*.csv');
files           = dir(csvfiles);
Fieldfiles      = struct2cell(files);
sortedfilenames = natsortfiles(Fieldfiles(1,:));
filedata        = importdata(fullfile(pwd,sortedfilenames{1}));


% Create structure into which we load data
timeSortedDataField = struct();
for i = 1:numel(filedata.colheaders)
    structFieldName = strrep(filedata.colheaders{i},'\"','');
    structFieldName = strrep(structFieldName,':','');
    structFieldName = strrep(structFieldName,' ','_');
    timeSortedDataField.(structFieldName)=filedata.data(:,i);
end

% Load data into respective structure Fields
for i = 2:numel(sortedfilenames)
    disp(sortedfilenames(i))
    currfile = importdata(fullfile(pwd,sortedfilenames{i}));
    [length, height] = size(currfile.data);
    for j = 1:numel(currfile.colheaders)
        structFieldName = strrep(currfile.colheaders{j},'\"','');
        structFieldName = strrep(structFieldName,':','');
        structFieldName = strrep(structFieldName,' ','_');

        timeSortedDataField.(structFieldName) = [timeSortedDataField.(structFieldName),currfile.data(:,j)];
    end

end

% Save structure fields to mat file
save('timeSortedDataField.mat','timeSortedDataField', '-v7.3');
disp('Done')


clear;
close;
clc;
```

```
%% Cells
csvfiles        = strcat(pwd, '/Cell*.csv');
files           = dir(csvfiles);
cellfiles       = struct2cell(files);
sortedfilenames = natsortfiles(cellfiles(1,:));
filedata        = importdata(fullfile(pwd,sortedfilenames{1}));


% Create structure into which we load data
timeSortedDataCells = struct();
for i = 1:numel(filedata.colheaders)
    structFieldName = strrep(filedata.colheaders{i},'\"','');
    structFieldName = strrep(structFieldName,':','');
    structFieldName = strrep(structFieldName,' ','_');
    timeSortedDataCells.(structFieldName)=filedata.data(:,i);
end

% Load data into respective structure fields
for i = 2:numel(sortedfilenames)
    disp(sortedfilenames(i))
    currfile = importdata(fullfile(pwd,sortedfilenames{i}));
    [length, height] = size(currfile.data);
    for j = 1:numel(currfile.colheaders)
        structFieldName = strrep(currfile.colheaders{j},'\"','');
        structFieldName = strrep(structFieldName,':','');
        structFieldName = strrep(structFieldName,' ','_');

        timeSortedDataCells.(structFieldName) = [timeSortedDataCells.(structFieldName),currfile.data(:,j)];
    end

end

% Save structure fields to mat file
save('timeSortedDataCells.mat','timeSortedDataCells', '-v7.3');
clear

%% Points
csvfiles        = strcat(pwd, '/Point*.csv');
files           = dir(csvfiles);
cellfiles       = struct2cell(files);
sortedfilenames = natsortfiles(cellfiles(1,:));
filedata        = importdata(fullfile(pwd,sortedfilenames{1}));


timeSortedDataPoints = struct();
for i = 1:numel(filedata.colheaders)
    structFieldName = strrep(filedata.colheaders{i},'\"','');
    structFieldName = strrep(structFieldName,':','');
    structFieldName = strrep(structFieldName,' ','_');
    timeSortedDataPoints.(structFieldName)=filedata.data(:,i);
end

for i = 2:numel(sortedfilenames)
    disp(sortedfilenames(i))
    currfile = importdata(fullfile(pwd,sortedfilenames{i}));
    [length, height] = size(currfile.data);
    for j = 1:numel(currfile.colheaders)

        structFieldName = strrep(currfile.colheaders{j},'\"','');
        structFieldName = strrep(structFieldName,':','');
        structFieldName = strrep(structFieldName,' ','_');

        timeSortedDataPoints.(structFieldName) = [timeSortedDataPoints.
(structFieldName),currfile.data(:,j)];
    end

end

% Save structure fields to mat file
save('timeSortedDataPoints.mat','timeSortedDataPoints', '-v7.3');
clear

disp('Done')
exit()
">dataInspectorTT.m
####################################################
#               DataAnalyserTT.m                #########################################################
####################################################
echo "clear all;
close all;
clc;
format long


%% File Availability check
tic
disp('File Availability')
fieldMatfiles   = strcat(pwd, '/*Field*.mat');
fieldFiles      = dir(fieldMatfiles);
```

```
fieldCellfiles  = struct2cell(fieldFiles);
sortedFieldfilenames = natsortfiles(fieldCellfiles(1,:));

disp('File Availability')
pointMatfiles   = strcat(pwd, '/*Point*.mat');
pointFiles      = dir(pointMatfiles);
pointCellfiles  = struct2cell(pointFiles);
sortedPointfilenames = natsortfiles(pointCellfiles(1,:));

cellMatfiles   = strcat(pwd, '/*Cell*.mat');
cellFiles      = dir(cellMatfiles);
cellCellfiles  = struct2cell(cellFiles);
sortedCellfilenames = natsortfiles(cellCellfiles(1,:));

toc
%% Loading mat-files
tic
disp('Loading MAT-files')
finalTime   = 10;
figIndx     = 0;
k = 1; %Ignore loop since there is only one field file and one cell file.
%for i = 1:length(sortedPointfilenames)

pointfilename      =  sortedPointfilenames{k};        %Load current mat-file
pointfile          = load (pointfilename);
pointfilename      = strrep(pointfilename,'.mat',''); %Remove .mat to use for variable

cellfilename      =  sortedCellfilenames{k};        %Load current mat-file
cellfile          = load (cellfilename);
cellfilename      = strrep(cellfilename,'.mat',''); %Remove .mat to use for variable

fieldfilename      = sortedFieldfilenames{k};        %Load current mat-file
fieldfile          = load (fieldfilename);
fieldfilename      = strrep(fieldfilename,'.mat',''); %Remove .mat to use for variable
toc
%% Calculation-related variables
tic
disp('Variables')
timeIncr    = finalTime/length(fieldfile.(fieldfilename).Left_Reaction_Force0(:));
time        = 0:timeIncr:finalTime-timeIncr;
x_section   = 1e-3*2e-4; %Cross section of domain.
nodeLeftInitial   = 1218;
nodeRightInitial  = 11565;
nodeLeftDisp      = 1218;
nodeRightDisp     = 11565;

%Interval node set
elemIndxLeft =find(pointfile.(pointfilename).PedigreeNodeId(:,1)==nodeLeftInitial);
elemIndxRight =find(pointfile.(pointfilename).PedigreeNodeId(:,1)==nodeRightInitial);

LeftX = pointfile.(pointfilename).Coordinates0(elemIndxLeft,1);
RightX = pointfile.(pointfilename).Coordinates0(elemIndxRight,1);

LeftXDisp = pointfile.(pointfilename).Displacement0(elemIndxLeft,:);
RightXDisp = pointfile.(pointfilename).Displacement0(elemIndxRight,:);

LeftXForce = pointfile.(pointfilename).Force0(elemIndxLeft,:);
RightXForce = pointfile.(pointfilename).Force0(elemIndxRight,:);


intervalNodes1 = find( (pointfile.(pointfilename).Coordinates0(:,1)>=LeftX));
intervalNodes = find(pointfile.(pointfilename).Coordinates0(intervalNodes1,1) <= RightX);
interval = intervalNodes;

L0          = RightX - LeftX;
elongation  = RightXDisp - LeftXDisp;
L           = elongation + L0;

cvNodes = find (abs(pointfile.timeSortedDataPoints.Coordinates0(:,1)) < 0.0000318);

%Retrieve the node index among the cvNodes, which are located on the ends
%of the domain.
%The node is the one to be used for calculations for the rest of the time
%period
[~, yMaxIndex]= max(pointfile.(pointfilename).Coordinates1(cvNodes,1));
[~, yMinIndex] = min(pointfile.(pointfilename).Coordinates1(cvNodes,1));
yCalcMaxNode = cvNodes(yMaxIndex);
yCalcMinNode = cvNodes(yMinIndex);
yWidth = pointfile.(pointfilename).Coordinates1(yCalcMaxNode,:) - pointfile.
(pointfilename).Coordinates1(yCalcMinNode,:);

[~, zMaxIndex ]= max(pointfile.(pointfilename).Coordinates2(cvNodes,1));
[~, zMinIndex]= min(pointfile.(pointfilename).Coordinates2(cvNodes,1));
zCalcMaxNode = cvNodes(zMaxIndex);
zCalcMinNode = cvNodes(zMinIndex);
zWidth = pointfile.(pointfilename).Coordinates2(zCalcMaxNode,:) - pointfile.
(pointfilename).Coordinates2(zCalcMinNode,:);

varyingCrossSection = (yWidth).*(zWidth);
```

```
diff = (varyingCrossSection(1)-x_section);
varyingCrossSection = varyingCrossSection - diff;


%% Engineering Stress/Strain
eStrain = elongation./L0; %Dimensionless, no conversion needed
eStress = ((fieldfile.(fieldfilename).Left_Reaction_Force0(:))./varyingCrossSection(1);
tStress = ((fieldfile.(fieldfilename).Left_Reaction_Force0(:))./varyingCrossSection';
tStrain = log(L/L0);

figIndx=+1;
figure(figIndx)
plot((eStrain),(eStress*10^(-6)),'-o');
xlabel('Engineering Strain')
ylabel('Engineering Stress[MPa]')
%MODIFIERA DENNA
legend('Stress-Strain','Location','southeast')
grid on
name = strcat('eStressStrain','.png');
saveas(figure(figIndx), name);

figIndx=figIndx+1;
figure(figIndx)
plot((tStrain),(tStress*10^(-6)),'-o');
xlabel('True Strain')
ylabel('True Stress[MPa]')
%MODIFIERA DENNA
legend('Stress-Strain','Location','southeast')
grid on
name = strcat('tStressStrain','.png');
saveas(figure(figIndx), name);

figIndx=figIndx+1;
figure(figIndx)
plot((eStrain),(eStress*10^(-6)),'-o');
hold on
plot((tStrain),(tStress*10^(-6)),'-o');
hold off
xlabel('Strain')
ylabel('Stress[MPa]')
%MODIFIERA DENNA
legend('Engineering Stress-Strain','True Stress-Strain','Location','southeast')
grid on
name = strcat('StressStrain','.png');
saveas(figure(figIndx), name);


exit()

">dataAnalyserTT.m
####################################################
#             Running the scripts                ##
####################################################

echo "Running paraview script."

# The file path needs to be full, since when using sudo, it can't seem to find any alias in the .bashrc.
Modify this to each computer.
if sudo /home/peridyna/peri/ParaView-5.7.0-MPI-Linux-Python3.7-64bit/bin/pvbatch pv_SS_data_script.py
#if sudo /home/theodor/src/ParaView-5.7.0-MPI-Linux-Python3.7-64bit/bin/pvbatch pv_SS_data_script.py
then
    #Remove the unneccesary rows. Must be modified if the trace generated py-file is changed. "Title"
column in csv-files breaks the dataInspector.
    sudo cut -d, -f20-111 --complement FieldData.csv>FieldData2.csv
    sudo rm FieldData.csv
    matlab -nodisplay -nosplash -nodesktop -r "run('dataInspectorTT.m')"
    matlab -nodisplay -nosplash -nodesktop -r "run('dataAnalyserTT.m')"
fi

#Cleanup since .csv have already been exported to .mat-files to save data
rm -f *.csv
```

F MATLAB code: Harmonic Structure Generation

```
%%Node generation of harmonic structure domain
%Theodor de Sousa   theodor.desousa@gmail.com
%Last edit: 2020 05 28
%
%This code generates evenly spaced nodes, assigns them two kinds of
    block ID
%depending on distance to a sphere center, as well as boundary
    condition
%node sets for Peridigm.
%
%To be fully functional, they should be converted to exodusII input
    using the
%script "text_to_genesis.py" provided in the source code to Peridigm.
%
%The script adjusts the sphere center iteratively to ensure a
    percentage of
%40 % of nodes outside of spheres and 60 % inside sphere, within the
    domain
%excluding the boundary sides on the left and right ends.

close all;
clear all;
clc;
%% User defined values
domainX          = 5.0e 3;     % Length
domainY      = 1.e 3;      % Height
domainZ      = domainY/5; % Depth
resolutionX = 200;        % Amount of nodes in x direction

%% Initial values

domainA = domainX*domainY;           %m2

commonRes        = resolutionX/domainX;

pointsX = resolutionX+3;
pointsY = (commonRes*domainY);
pointsZ  = (commonRes*domainZ);

pointsA = pointsX*pointsY;
pointsV = pointsX*pointsY*pointsZ;



elementSize      = domainX/pointsX;          % The spacing between
    material nodes
delta        = elementSize;
domainV      = domainX*domainY*domainZ;  % m3

totalPoints1     = pointsV;
elementVolume    = elementSize^2;
halfElemSize     = elementSize/2;
horizon                 = 3.0*delta;         %Could be 3.015 to ensure
     full family.
```

```
%% Generate grains
disp('Placing grain centers')
%   To reduce computational time if run often, adjust the intial
    guess of
%   grainlength to yield an initial guess.
grainDist   = domainY/10;  %Amount of grains in y direction decides.
grainLength = 1.663589216265020*grainDist;
xGrains     = 1/grainDist*domainX;
yGrains     = 1/grainDist*domainY;
zGrains     = 1/grainDist*domainZ;

grainCenters = [];
grain = 0;
for i =  1:xGrains
    for j =  1:yGrains
        for k =  1:zGrains
            if xor(mod(i,2),mod(j,2))
                z = k*grainDist + grainDist;
            else
                z = k*grainDist;
            end

            if xor(mod(j,2),mod(k,2))
                x = i*grainDist + grainDist;
            else
                x = i*grainDist;
            end

            if xor(mod(i,2),mod(k,2))
                y = j*grainDist + grainDist;
            else
                y = j*grainDist;
            end

            if x > (horizon + grainDist) && x < domainX   (horizon +
               grainDist)
                grainCenters = [grainCenters;[x,y,z]];
            end

        end
    end
end


%% Create node files
id = 1;
data = zeros(totalPoints1,5);
nodelist_1  = [];
nodelist_2  = [];
nodelist_3  = [];
nodelist_4  = [];
nodelist_5  = [];
nodelist_6  = [];
nodelist_7  = [];
nodelist_8  = [];
nodelist_9  = [];
nodelist_10 = [];
```

```
nodelist_11 = [];
z = 0.0;
ratio = 0.4002;
soughtUFGPoints = 0;
totalPoints = 0;

%% Points and boundary conditions
disp('Placing points')
for i = 1:pointsX
    for j = 1:pointsY
        for k = 1:pointsZ
            % Coordinates
            x = i*elementSize   halfElemSize;
            y = j*elementSize   halfElemSize;
            z = k*elementSize   halfElemSize;
            data(id,[1,2,3,5]) = [x, y, z, elementVolume];

            % Boundary conditions
            if x <= horizon % Left side clamped
                nodelist_1 = [nodelist_1; id];
            elseif x >= domainX   horizon   % Right side elongated
                nodelist_2 = [nodelist_2; id];
            else
                nodelist_3 = [nodelist_3; id]; % The bulk
            end
            if (x <= horizon && y <= horizon && z < horizon)
                                            ...     % Low Left
               Front
                    || (x <= horizon && y <= horizon && z > domainZ
                        horizon)             ...     % Low Left  Back
                    || (x <= horizon && y >= domainY   horizon && z <
                        horizon)             ...     % Up  Left
                       Front
                    || (x <= horizon && y >= domainY   horizon && z >
                        domainZ   horizon)  ...     % Up  Left  Back
                    || (x >= domainX   horizon && y <= horizon && z <
                        horizon)             ...     % Low Right
                       Front
                    || (x >= domainX   horizon && y <= horizon && z >
                        domainZ   horizon)  ...     % Low Right Back
                    || (x >= domainX   horizon && y >= domainY
                       horizon && z < horizon)  ...     % Up  Right
                       Front
                    || (x >= domainX   horizon && y >= domainY
                       horizon && z > domainZ   horizon)% Up  Right
                       Back
                nodelist_4 = [nodelist_4; id];
            end
            if y <= horizon
                nodelist_5 = [nodelist_5; id];
            end
            if abs(y) <= horizon && ( x <= horizon || x >= (domainX
                horizon) )
                nodelist_6 = [nodelist_6; id];
            end
            if y >= domainY   horizon   halfElemSize
                nodelist_7 = [nodelist_7; id];
            end
            if abs(y domainY/2) < horizon && ( x <= horizon || x >= (
               domainX   horizon) )
                nodelist_8 = [nodelist_8; id];
            end
            % Prevent rigid body motion in z
            if (z <= elementSize&& x <= elementSize) || (z <=
               elementSize && x >= domainX   elementSize  )
                nodelist_9 = [nodelist_9;id];
            end
            % Prevent rigid body motion in y
            if (y <= elementSize && x <= elementSize) || (y <=
               elementSize && x >= domainX   elementSize )
                nodelist_10 = [nodelist_10;id];
            end
            if ~(x > (horizon + grainDist) && x < domainX   (horizon
               + grainDist))
                nodelist_11 = [nodelist_11;id];
            end
            id = id+1;
        end
    end
end

%% Grain size
disp('Find grain size')
ratioCheck = 1;
loopIndx = 0;
%Eliminate the sides
dataRows = 1:length(data);
sideIndex = ismember(dataRows,[nodelist_11,]);
notSideIndex = ~sideIndex;
dataSideRows = dataRows(sideIndex);
dataNotSideRows = dataRows(notSideIndex);

while ratioCheck
    %The exponent dampens the iterative guesses, modify to adjust for
         overshoot.
    grainLengthChange = (ratio/0.4)^(sqrt(abs(ratio 0.4)));
    grainLength = grainLength*(grainLengthChange);

%   Assigning block IDs
    for i=1:length(data)
        x=data(i,1);
        y=data(i,2);
        z=data(i,3);
        for t = 1:length(grainCenters)
            if (norm([x,y,z] grainCenters(t,:)) <= (grainLength/2))
                %CG
                block_id = 1;
                break
            else
                %UFG
                block_id = 2;
            end

        end
        data(i,4) = block_id;
    end

   %Finding the ratios of the grain sizes.
```

```
    %Find amount of points with UFG and total amount of points
    UFGPoints       = length(find(data(dataNotSideRows,4)==2));
    soughtUFGPoints = totalPoints*0.4;
    totalPoints     = length(dataNotSideRows);

    ratio = UFGPoints/totalPoints;
    loopIndx = loopIndx+1;

    % Behaves like a damped
%    plot(loopIndx,ratio,'o')
%    drawnow
%    hold on

    ratioCheck = abs(ratio   0.4) > 0.001;
    if(~ratioCheck)
        data(id,4) = block_id;
    end
end


%% Post processing
% centering domain on 0
data(:,1) = data(:,1)   domainX/2 + halfElemSize;
data(:,2) = data(:,2)   domainY/2 + halfElemSize;
data(:,3) = data(:,3)   domainZ/2 + halfElemSize;

% Introducing perturbations in the nodes
%pert = 0.001*elementSize;
%for i = 1:length(data(:,1))
    %    data(i,1) = data(i,1) + (2*rand   1)*pert;
    %    data(i,2) = data(i,2) + (2*rand   1)*pert;
    %    data(i,3) = data(i,3) + (2*rand   1)*pert;
%end

%Compute class data:
xLeftCoord = 2*domainX/8  domainX/2;
yLeftCoord = 0.0;
zLeftCoord = 0.0;

xRightCoord = domainX   2*domainX/8  domainX/2;
yRightCoord = 0.0;
zRightCoord = 0.0;


disp('Writing matrices')

writematrix(data,'domainMeshNoPert.txt','Delimiter',' ');
writematrix(nodelist_1,'nodelist_1.txt','Delimiter',' ');
writematrix(nodelist_2,'nodelist_2.txt','Delimiter',' ');
writematrix(nodelist_3,'nodelist_3.txt','Delimiter',' ');
writematrix(nodelist_4,'nodelist_4.txt','Delimiter',' ');
writematrix(nodelist_5,'nodelist_5.txt','Delimiter',' ');
writematrix(nodelist_6,'nodelist_6.txt','Delimiter',' ');
writematrix(nodelist_7,'nodelist_7.txt','Delimiter',' ');
writematrix(nodelist_8,'nodelist_8.txt','Delimiter',' ');
writematrix(nodelist_9,'nodelist_9.txt','Delimiter',' ');
writematrix(nodelist_10,'nodelist_10.txt','Delimiter',' ');
writematrix(nodelist_11,'nodelist_11.txt','Delimiter',' ');
```

G MATLAB code: Random Structure Generation

```
%%Node generation of random rectangular domain
%Theodor de Sousa  theodor.desousa@gmail.com
%Last edit: 2020 05 28
%
%This code generates randomly spaced nodes, assigns them two kinds of
     block ID
%depending on distance to a sphere center, as well as boundary
    condition node sets for Peridigm.

close all;
clear all;
clc;
%% User defined values
domainX         = 2.5e 3;
domainY      = 1.0e 3;    %m
pointsZ      = 4;         % nodes
resolutionX  = 400;
strainrate   = 3.6e 3;    % m/s
E               = 210e9;
poissons        = 0.31;

soughtResolution = 1370706;

%% Initial values
rtext = num2str(resolutionX);

domainA = domainX*domainY;       %m2

commonRes       = resolutionX/domainX; % nodes/meter

pointsX = resolutionX+3;                        % nodes
pointsY = round(commonRes*domainY);      % nodes

pointsA = pointsX*pointsY;
pointsV = pointsX*pointsY*pointsZ;



elementSize     = domainX/pointsX;          % The spacing between
    material nodes
delta        = elementSize;
domainZ      = pointsZ*elementSize;              % m
domainV      = domainX*domainY*domainZ;  % m3

totalPoints             = pointsV;
elementVolume   = domainV/totalPoints;
halfElemSize    = elementSize/2;
horizon                 = 3.015*delta;
G               = E/(2*(1+poissons));
K               = E/(3*(1 2*poissons));
G0              = 40; % This is just an assumption to get something
scStateBased    = sqrt( G0/(horizon*( 3*G + ( 3/4 )^4*( K   2*G ) ))
    );
scBondBased     = sqrt(5*G0/(9*K*horizon));

%% Generera korn
disp('Placing grains')
```

```
grainDist   = domainY/11;
grainLength = 1.2*grainDist/1.8;        %m
xGrains     = 1/grainDist*domainX;
yGrains     = 1/grainDist*domainY;
zGrains     = 1/grainDist*domainZ; %Disregard

grainCenters = [];

%Calculate amount of grains
grain = 0;
for i =  1:xGrains+1
    for j =  1:yGrains+1
        grain = grain + 1;
    end
end
grainCenters = [];
grainLengths = [];
%Random normal distribution
%grainLengths = (randn(grain,1,'like',grainLength)+0.5)*grainLength;
%Random equal distribution
grainLengths = horizon + rand(grain,1)*grainLength*1.8;

%grainLengths = grainLength*rand(grain,1);
grainLengths = abs(grainLengths);

figure(1)
hold on
for i = 1:grain
    accepted = 0;
    xTest = 0;
    yTest = 0;


%% Single rectangle
%     disp('Grain')
%     disp(i);
%     while accepted ~= 1
%         %       disp('Refresh')
%         xTest   = ((2*horizon + grainDist) + domainX*rand);
%         yTest   = domainY*rand;
%         grainLengths(i) = horizon + rand*grainLength*1.8;
%         if xTest > (2*horizon + grainDist) && xTest < domainX   (2*
    horizon + grainDist)
%
%             if length(grainCenters) > 1
%                 for t = 1:length(grainCenters(:,1))
%                     %   disp('t: ')
%                     %   disp(t)
%                     if (norm([xTest,yTest] grainCenters(t,:)) < (
    grainLengths(t)/2+1/2*grainLengths(i)))
%                         accepted = 0;
%                         break;
%                     else
%                         accepted = 1;
%                     end
%                 end
%             else
%                 accepted = 1;
%             end
%          end
%      end
%      grainCenters(i,:) = [xTest,yTest];
%
%      plot(grainCenters(i,1),grainCenters(i,2),'o')
%      pause(0.01)
    %   grainLengths(i) = gLTest;

    %% Four rectangles
    if i<round(grain/4)
        disp('Section 1')
        disp('Grain')
        disp(i);
        while accepted ~= 1
            %        disp('Refresh')
            xTest   = ((2*horizon + grainDist) + domainX/2*rand);
            yTest   = domainY/2*rand;
            grainLengths(i) = horizon + rand*grainLength*1.8;

            if xTest > (2*horizon + grainDist) && xTest < domainX
                (2*horizon + grainDist)

                if length(grainCenters) > 1
                    for t = 1:length(grainCenters(:,1))
                        %    disp('t: ')
                        %    disp(t)
                        if (norm([xTest,yTest] grainCenters(t,:)) < (
                            grainLengths(t)/2+1/2*grainLengths(i)))
                            accepted = 0;
                            break;
                        else
                            accepted = 1;
                        end
                    end
                else
                    accepted = 1;
                end
            end
        end
        grainCenters(i,:) = [xTest,yTest];

        plot(grainCenters(i,1),grainCenters(i,2),'o')
        pause(0.01)
        %   grainLengths(i) = gLTest;

    elseif  i<round((2*1/4)*grain)

        disp('Section 2')
        disp('Grain')
        disp(i);
        accepted = 0;
        while accepted ~= 1
            %        disp('Refresh')
            xTest   = ((2*horizon + grainDist) + domainX/2*rand);
            yTest   = domainY/2 + domainY/2*rand;
             grainLengths(i) = horizon + rand*grainLength*1.8;
            if xTest > (2*horizon + grainDist) && xTest < domainX
                (2*horizon + grainDist)
            if length(grainCenters) > 1
                for t = 1:length(grainCenters(:,1))
                    %   disp('t: ')
                    %   disp(t)
                    if (norm([xTest,yTest] grainCenters(t,:)) < (
                       grainLengths(t)/2+1/2*grainLengths(i)))
                        accepted = 0;
                        break;
                    else
                        accepted = 1;
                    end
                end
            else
                accepted = 1;
            end
        end
    end
    grainCenters(i,:) = [xTest,yTest];

    plot(grainCenters(i,1),grainCenters(i,2),'o')
    pause(0.01)

elseif i < round((3*1/4)*grain)

    disp('Section 3')
    accepted = 0;
    disp('Grain')
    disp(i);
    while accepted ~= 1
        %       disp('Refresh')
        xTest   = domainX/2 + (domainX/2   (2*horizon + grainDist
           ))*rand;
        yTest   = domainY/2*rand;
         grainLengths(i) = horizon + rand*grainLength*1.8;
        if xTest > (2*horizon + grainDist) && xTest < domainX
            (2*horizon + grainDist)

            if length(grainCenters) > 1
                for t = 1:length(grainCenters(:,1))
                    %   disp('t: ')
                    %   disp(t)
                    if (norm([xTest,yTest] grainCenters(t,:)) < (
                       grainLengths(t)/2+1/2*grainLengths(i)))
                        accepted = 0;
                        break;
                    else
                        accepted = 1;
                    end
                end
            else
                accepted = 1;
            end
        end
    end
    grainCenters(i,:) = [xTest,yTest];

    plot(grainCenters(i,1),grainCenters(i,2),'o')
    pause(0.01)
else
```

```
        disp('Section 4')
        accepted = 0;
        disp('Grain')
        disp(i);
        while accepted ~= 1
            %       disp('Refresh')
            xTest   = domainX/2 + (domainX/2   (2*horizon + grainDist
                ))*rand;
            yTest   = domainY/2 + domainY/2*rand;
             grainLengths(i) = horizon + rand*grainLength*1.8;
            if xTest > (2*horizon + grainDist) && xTest < domainX
                (2*horizon + grainDist)

                if length(grainCenters) > 1
                    for t = 1:length(grainCenters(:,1))
                        %   disp('t: ')
                        %   disp(t)
                        if (norm([xTest,yTest] grainCenters(t,:)) < (
                            grainLengths(t)/2+1/2*grainLengths(i)))
                            accepted = 0;
                            break;
                        else
                            accepted = 1;
                        end
                    end
                else
                    accepted = 1;
                end
            end
        end
        grainCenters(i,:) = [xTest,yTest];

        plot(grainCenters(i,1),grainCenters(i,2),'o')
        pause(0.01)
        %   grainLengths(i) = gLTest;
    end
%
 end
grainInfo = [grainCenters, grainLengths];

%% Skapa nodfil
id = 1;
data = zeros(totalPoints,5);
nodelist_1 = [];
nodelist_2 = [];
nodelist_3 = [];
nodelist_4 = [];
nodelist_5 = [];
nodelist_6 = [];
z = 0.0;
disp('Placing points')
for i = 1:pointsX
    for j = 1:pointsY
        for k = 1:pointsZ
            x = i*elementSize   halfElemSize;
            y = j*elementSize   halfElemSize;
            z = k*elementSize   halfElemSize;
            %Check grain type
            for t = 1:length(grainCenters)
```

```
                if (norm([x,y] grainCenters(t,:)) <= (grainLengths(t)
                    /2))
                    %CG
                    block_id = 1;
                    break
                else
                    %UFG
                    block_id = 2;
                end
            end
            data(id,:) = [x, y, z, block_id, elementVolume];

            % Boundary conditions
            if x <= 2*horizon + 2*elementSize  % Left side clamped
                nodelist_1 = [nodelist_1; id];
            elseif x >= domainX   2*horizon    % Right side elongated
                nodelist_2 = [nodelist_2; id];
            else
                nodelist_3 = [nodelist_3; id]; % The bulk
            end
            if (x < horizon && y < horizon && z < horizon)
                                     ...      % Low Left  Front
                    || (x < horizon && y < horizon && z > domainZ
                        horizon)                ...      % Low Left  Back
                    ...               || (x < horizon && y > domainY
                        horizon && z < horizon)               ...      %
                        Up  Left  Front
                    ...               || (x < horizon && y > domainY
                        horizon && z > domainZ   horizon)  ...      %
                        Up  Left  Back
                    || (x > domainX   horizon && y < horizon && z <
                        horizon)                ...      % Low Right Front
                    || (x > domainX   horizon && y < horizon && z >
                        domainZ   horizon)  ...      % Low Right Back
                    ...               || (x > domainX   horizon && y >
                        domainY   horizon && z < horizon)  ...      %
                        Up  Right Front
                    ...               || (x > domainX   horizon && y >
                        domainY   horizon && z > domainZ   horizon)%
                        Up  Right Back
                    nodelist_4 = [nodelist_4; id];
            end
            if y < horizon
                nodelist_5 = [nodelist_5; id];
            end
            if abs(y   domainY/2) < horizon && ( x < horizon || x > (
               domainX   horizon) )
                nodelist_6 = [nodelist_6; id];
            end
            id = id+1;
        end
    end
end



disp('Drawing figures')
figure(2)
```

```
%plot(data(:,1),data(:,2),[data(:,4),1,0] 'o');
surface([data(:,1),data(:,1)],[data(:,2),data(:,2)],[data(:,3),data
    (:,3)],[data(:,4),data(:,4)],...
    'facecol','no','edgecol','interp','linew',2)
scatter(data(:,1),data(:,2),1,data(:,4));

figure(3)
plot3(data(:,1),data(:,2),data(:,3),'o');
disp('Writing Matrices')

writematrix(grainInfo,'grainInfo.txt','Delimiter',' ');


writematrix(data,'domainMesh.txt','Delimiter',' ');
writematrix(nodelist_1,'nodelist_1.txt','Delimiter',' ');
writematrix(nodelist_2,'nodelist_2.txt','Delimiter',' ');
writematrix(nodelist_3,'nodelist_3.txt','Delimiter',' ');
writematrix(nodelist_4,'nodelist_4.txt','Delimiter',' ');
writematrix(nodelist_5,'nodelist_5.txt','Delimiter',' ');
writematrix(nodelist_6,'nodelist_6.txt','Delimiter',' ');

%You can attach multiple files to an email.
%sendmail('recipient@someserver.com','Hello from MATLAB!', ...
%     'Thanks for using sendmail.',{'C:\yourFileSystem\message.txt',
    ...
%    'C:\yourFileSystem\message2.txt'});

%sendmail('lthperidigm@gmail.com','NodeSet2D','Here are some files
    ',...
%    {'/home/theodor/Documents/Skola/5_Examensarbete i mekanik
%    FMEM01/Peridigm/domainMesh.txt',...
%        '/home/theodor/Documents/Skola/5_Examensarbete i mekanik
    FMEM01/Peridigm/nodelist_1.txt',...
%        '/home/theodor/Documents/Skola/5_Examensarbete i mekanik
    FMEM01/Peridigm/nodelist_2.txt',...
%        '/home/theodor/Documents/Skola/5_Examensarbete i mekanik
    FMEM01/Peridigm/nodelist_3.txt'});
```

H MATLAB code: Data Conversion

```
clear;
close;
clc;
csvfiles        = strcat(pwd, '/Field*.csv');
files           = dir(csvfiles);
Fieldfiles      = struct2cell(files);
sortedfilenames = natsortfiles(Fieldfiles(1,:));
filedata        = importdata(fullfile(pwd,sortedfilenames{1}));

% Create structure into which we load data
timeSortedDataField = struct();
for i = 1:numel(filedata.colheaders)
    structFieldName = strrep(filedata.colheaders{i},'"','');
    structFieldName = strrep(structFieldName,':','');
    structFieldName = strrep(structFieldName,' ','_');
    timeSortedDataField.(structFieldName)=filedata.data(:,i);
end

% Load data into respective structure Fields
for i = 2:numel(sortedfilenames)
    disp(sortedfilenames(i))
    currfile = importdata(fullfile(pwd,sortedfilenames{i}));
    [length, height] = size(currfile.data);
    for j = 1:numel(currfile.colheaders)
        structFieldName = strrep(currfile.colheaders{j},'"','');
        structFieldName = strrep(structFieldName,':','');
        structFieldName = strrep(structFieldName,' ','_');

        timeSortedDataField.(structFieldName) = [timeSortedDataField
            .(structFieldName),currfile.data(:,j)];
    end
end

% Save structure fields to mat file
save('timeSortedDataField.mat','timeSortedDataField');
disp('Done')

clear;
close;
clc;

%% Cells
csvfiles        = strcat(pwd, '/Cell*.csv');
files           = dir(csvfiles);
cellfiles       = struct2cell(files);
sortedfilenames = natsortfiles(cellfiles(1,:));
filedata        = importdata(fullfile(pwd,sortedfilenames{1}));


% Create structure into which we load data
timeSortedDataCells = struct();
for i = 1:numel(filedata.colheaders)
    structFieldName = strrep(filedata.colheaders{i},'"','');
    structFieldName = strrep(structFieldName,':','');
    structFieldName = strrep(structFieldName,' ','_');
    timeSortedDataCells.(structFieldName)=filedata.data(:,i);
```

```
end

% Load data into respective structure fields
for i = 2:numel(sortedfilenames)
    disp(sortedfilenames(i))
    currfile = importdata(fullfile(pwd,sortedfilenames{i}));
    [length, height] = size(currfile.data);
    for j = 1:numel(currfile.colheaders)
        structFieldName = strrep(currfile.colheaders{j},'"','');
        structFieldName = strrep(structFieldName,':','');
        structFieldName = strrep(structFieldName,' ','_');

        timeSortedDataCells.(structFieldName) = [timeSortedDataCells
            .(structFieldName),currfile.data(:,j)];
    end
end

% Save structure fields to mat file
save('timeSortedDataCells.mat','timeSortedDataCells');
clear

%% Points
csvfiles        = strcat(pwd, '/Point*.csv');
files           = dir(csvfiles);
cellfiles       = struct2cell(files);
sortedfilenames = natsortfiles(cellfiles(1,:));
filedata        = importdata(fullfile(pwd,sortedfilenames{1}));


timeSortedDataPoints = struct();
for i = 1:numel(filedata.colheaders)
    structFieldName = strrep(filedata.colheaders{i},'"','');
    structFieldName = strrep(structFieldName,':','');
    structFieldName = strrep(structFieldName,' ','_');
    timeSortedDataPoints.(structFieldName)=filedata.data(:,i);
end

for i = 2:numel(sortedfilenames)
    disp(sortedfilenames(i))
    currfile = importdata(fullfile(pwd,sortedfilenames{i}));
    [length, height] = size(currfile.data);
    for j = 1:numel(currfile.colheaders)

        structFieldName = strrep(currfile.colheaders{j},'"','');
        structFieldName = strrep(structFieldName,':','');
        structFieldName = strrep(structFieldName,' ','_');

        timeSortedDataPoints.(structFieldName) = [
            timeSortedDataPoints.(structFieldName),currfile.data(:,j)
            ];
    end
end

% Save structure fields to mat file
save('timeSortedDataPoints.mat','timeSortedDataPoints');
clear

disp('Done')
exit()
```

I MATLAB code: Data Analysation

```
clear all;
close all;
clc;
format long

%% File Availability check
tic
disp('File Availability')
fieldMatfiles    = strcat(pwd, '/*Field*.mat');
fieldFiles       = dir(fieldMatfiles);
fieldCellfiles   = struct2cell(fieldFiles);
sortedFieldfilenames = natsortfiles(fieldCellfiles(1,:));

disp('File Availability')
pointMatfiles    = strcat(pwd, '/*Point*.mat');
pointFiles       = dir(pointMatfiles);
pointCellfiles   = struct2cell(pointFiles);
sortedPointfilenames = natsortfiles(pointCellfiles(1,:));

cellMatfiles    = strcat(pwd, '/*Cell*.mat');
cellFiles       = dir(cellMatfiles);
cellCellfiles   = struct2cell(cellFiles);
sortedCellfilenames = natsortfiles(cellCellfiles(1,:));

toc
%% Loading mat files
tic
disp('Loading MAT files')
finalTime    = 120;
figIndx      = 0;
k = 1; %Ignore loop since there is only one field file and one cell
    file.
%for i = 1:length(sortedPointfilenames)

pointfilename      =  sortedPointfilenames{k};        %Load current
    mat file
pointfile          = load (pointfilename);
pointfilename      = strrep(pointfilename,'.mat',''); %Remove .mat to
     use for variable

cellfilename      =  sortedCellfilenames{k};        %Load current mat
     file
cellfile          = load (cellfilename);
cellfilename      = strrep(cellfilename,'.mat',''); %Remove .mat to
    use for variable

fieldfilename      = sortedFieldfilenames{k};        %Load current
    mat file
fieldfile          = load (fieldfilename);
fieldfilename      = strrep(fieldfilename,'.mat',''); %Remove .mat to
     use for variable
toc
%% Calculation related variables
tic
disp('Variables')
timeIncr     = finalTime/length(fieldfile.(fieldfilename).
    Bottom_Reaction_Force1(:));
time          = 0:timeIncr:finalTime timeIncr;
x_section     = 1.999996e 05; %Cross section of the specific tensile
    test bar in m2.
nodeTopInitial   = 6715;
nodeBotInitial   = 6676;
nodeTopDisp      = 6715;
nodeBotDisp      = 6676;

%Interval node set
elemIndxTop =find(pointfile.(pointfilename).GlobalNodeId(:,1)==
    nodeTopInitial);
elemIndxBot =find(pointfile.(pointfilename).GlobalNodeId(:,1)==
    nodeBotInitial);
topY = pointfile.(pointfilename).Coordinates1(elemIndxTop,2);
botY = pointfile.(pointfilename).Coordinates1(elemIndxBot,2);
intervalNodes1 = find( (pointfile.(pointfilename).Coordinates1(:,2)<=
    topY));
intervalNodes = find(pointfile.(pointfilename).Coordinates1(
    intervalNodes1,2) >= botY);
interval = intervalNodes;

L0            = (fieldfile.(fieldfilename).Gage_Top_Initial_Position1
    (1))    (fieldfile.(fieldfilename).Gage_Bottom_Initial_Position1
    (1));
elongation   = (fieldfile.(fieldfilename).Gage_Top_Displacement1(:))
          (fieldfile.(fieldfilename).Gage_Bottom_Displacement1(:));
L             = elongation + L0;

cvNodes = find (abs(pointfile.timeSortedDataPoints.Coordinates1(:,1))
     < 0.0318);

%Retrieve the node index among the cvNodes, which are located on the
    ends
%of the domain.
%The node is the one to be used for calculations for the rest of the
    time
%peroid
[~, xMaxIndex]= max(pointfile.(pointfilename).Coordinates0(cvNodes,1)
    );
[~, xMinIndex] = min(pointfile.(pointfilename).Coordinates0(cvNodes
    ,1));
xCalcMaxNode = cvNodes(xMaxIndex);
xCalcMinNode = cvNodes(xMinIndex);
xWidth = pointfile.(pointfilename).Coordinates0(xCalcMaxNode,:)
    pointfile.(pointfilename).Coordinates0(xCalcMinNode,:);

[~, zMaxIndex ]= max(pointfile.(pointfilename).Coordinates2(cvNodes
    ,1));
[~, zMinIndex]= min(pointfile.(pointfilename).Coordinates2(cvNodes,1)
    );
zCalcMaxNode = cvNodes(zMaxIndex);
zCalcMinNode = cvNodes(zMinIndex);
zWidth = pointfile.(pointfilename).Coordinates2(zCalcMaxNode,:)
    pointfile.(pointfilename).Coordinates2(zCalcMinNode,:);

varyingCrossSection = (0.01*xWidth).*(0.01*zWidth);
diff = (varyingCrossSection(1) x_section);
varyingCrossSection = varyingCrossSection   diff;
```

```
%% Engineering Stress/Strain
eStrain = elongation./L0; %Dimensionless, no conversion needed
eStress = (fieldfile.(fieldfilename).Bottom_Reaction_Force1(:))
    .*0.00001./varyingCrossSection(1); %Conversion 1 dyne = 0.00001
    Newton and F/A = sigma
tStress = ((fieldfile.(fieldfilename).Bottom_Reaction_Force1(:))
    .*0.00001)./varyingCrossSection';
tStrain = log(L/L0);

figIndx=+1;
figure(figIndx)
plot((eStrain),(eStress*10^(6)),' o');
xlabel('Engineering Strain')
ylabel('Engineering Stress[MPa]')
%MODIFIERA DENNA
legend('Stress Strain','Location','southeast')
grid on
name = strcat('eStressStrain','.png');
saveas(figure(figIndx), name);

figIndx=figIndx+1;
figure(figIndx)
plot((tStrain),(tStress*10^(6)),' o');
xlabel('True Strain')
ylabel('True Stress[MPa]')
%MODIFIERA DENNA
legend('Stress Strain','Location','southeast')
grid on
name = strcat('tStressStrain','.png');
saveas(figure(figIndx), name);

figIndx=figIndx+1;
figure(figIndx)
plot((eStrain),(eStress*10^(6)),' o');
hold on
plot((tStrain),(tStress*10^(6)),' o');
hold off
xlabel('Strain')
ylabel('Stress[MPa]')
%MODIFIERA DENNA
legend('Engineering Stress Strain','True Stress Strain','Location','
    southeast')
grid on
name = strcat('StressStrain','.png');
saveas(figure(figIndx), name);


%% Force calculated stress

fStresszMax = (abs(pointfile.(pointfilename).Force1(zCalcMaxNode,:))
    .*0.00001)./varyingCrossSection;
figIndx=figIndx+1;
figure(figIndx)
plot((eStrain),(fStresszMax),' o');
xlabel('Strain')
ylabel('Force calculated Stress[MPa]')
hold on
```

```
fStresszMin = (abs(pointfile.(pointfilename).Force1(zCalcMinNode,:))
    .*0.00001)./varyingCrossSection;
plot((eStrain),(fStresszMin),' o');

fStressxMax = (abs(pointfile.(pointfilename).Force1(xCalcMaxNode,:))
    .*0.00001)./varyingCrossSection;
plot((eStrain),(fStressxMax),' o');

fStressxMin = (abs(pointfile.(pointfilename).Force1(xCalcMinNode,:))
    .*0.00001)./varyingCrossSection;
plot((eStrain),(fStressxMin),' o');

fStress = (fStresszMax + fStresszMin + fStressxMax + fStressxMin)./4;
plot((eStrain),(fStress),' o');

legend()
legend('Stress Strain','Location','southeast')
grid on
name = strcat('fStressStrain','.png');
saveas(figure(figIndx), name);

%% Von Mises Stress
%Displaying time varied average Von Mises
figIndx=figIndx+1;
figure(figIndx)
avgVonMises = mean(cellfile.(cellfilename).Von_Mises_Stress(interval
    ,:));
%                Plots                    %
plot(eStrain,avgVonMises);
title('Average interval Von Mises as function of strain')
xlabel('Strain')
ylabel('{\fontsize{16}\sigma_{v}} [MPa]')
grid on
name = strcat('VonMisesStressStrain','.png');
saveas(figure(figIndx), name);


%% Cauchy Stress
%Displaying time varied average Cauchy
figIndx=figIndx+1;
figure(figIndx)
intAvgCauchyY1 = mean(cellfile.(cellfilename).Cauchy_StressY1(
    interval,:));
%                Plots                    %
plot(eStrain,intAvgCauchyY1);
title('Average interval Cauchy YY as function of strain')
xlabel('Strain')
ylabel('{\fontsize{16}\sigma_{yy}} [MPa]')
grid on
name = strcat('CauchyXXStressStrain','.png');
saveas(figure(figIndx), name);

exit()
```

J MATLAB code: Stress-Strain Curve Discretisation

```
%Theodor de Sousa
% theodor.desousa@gmail.com
% Last edit: 2020 05 28
% Converts data points taken from analog plots and discretise these
    into true
% stress strain curves.
clear all
close all
clc

%% CG
%   Matrix generator of tensile curves from ASM Ni 200 Annealed
    nickel sheet
%        Two images, full, and "elastic" part

% "Elastic" part
c_elast_horisontal = 0.001*2/228;
c_elast_vertical = 70/300;

CG_Elastic_Strain = c_elast_horisontal
    *[13,24,34,44,54,64,74,84,94,104,114,124,134,144,154          ...
     164,194,224,284,344,404,454,504,554,604,664];
CG_Elastic_Stress = c_elast_vertical
    *[98,174,249,302,361,408,453,498,532,558,586,610,630,648,664,
            ...
     679,710,734,767, 787, 801, 810, 818, 824,829,833];

% Full part
c_full_horisontal = 0.05/149;
c_full_vertical = 140/226;

CG_Full_Strain = c_full_horisontal
    *[20,25,35,45,65,85,105,155,205,255,305,355,405,455,505,555,605,655,705,749];

%Adjusted for systematic error of 10 pixels
CG_Full_Stress = c_full_vertical
    *([327,334,347,358,382,404,424,473,512,548,576,601, 622, 640,
    655, 670,682, 691, 699, 705] 10);

% Complete Part
CG_Engineering_Strain = [CG_Elastic_Strain,CG_Full_Strain];
CG_Engineering_Stress = [CG_Elastic_Stress,CG_Full_Stress];


%Reverse get the gauge length from maximum strain and knowledge that
    total
%elongation is 39,5 percent
elongationPercentage_CG = 0.395;
gaugeLengthCG = elongationPercentage_CG/CG_Engineering_Strain(end);
%convert to millimeters
gaugeLengthCG = gaugeLengthCG*10^(3);

%Find the area change per step.
A0CG = 1;
```

```
VCG = A0CG*gaugeLengthCG;
elongationPerStepCG = gaugeLengthCG.*CG_Engineering_Strain;
AincrCG = VCG./(gaugeLengthCG+elongationPerStepCG);

%Ensure we don't use post necking calculations.
[~,maxIndexCG] = max(CG_Engineering_Stress);

% Calculate true stresses and strains.
%CG_True_Stress = CG_Engineering_Stress(1:maxIndexCG)./AincrCG(1:
    maxIndexCG);

CG_True_Stress = CG_Engineering_Stress(1:maxIndexCG).*(1+
    CG_Engineering_Strain(1:maxIndexCG));
CG_True_Strain = log((elongationPerStepCG(1:maxIndexCG)+gaugeLengthCG
    )/gaugeLengthCG);


%Power Law index:
logCG_TStrain = log(CG_True_Strain(35:end));
logCG_TStress = log(CG_True_Stress(35:end));

p = polyfit(logCG_TStrain,logCG_TStress,1);

figure(500)
plot(logCG_TStrain,logCG_TStress,'bo');
axis equal square
grid
xlabel('log(strain)');
ylabel('log(stress)');
k=p(1);
loga=p(2);
a=exp(loga);
hold on;
plot(logCG_TStrain,k*logCG_TStrain+loga,'g');
legend('Data',sprintf('stress=%.3f{}log(strain)+log(%.3f)',k,a));

figure(600)
plot(CG_True_Strain(35:end),CG_True_Stress(35:end),'bo');
xlabel('Strain');
ylabel('Stress [MPa]');
grid
hold on;
plot(CG_True_Strain(35:end),a*CG_True_Strain(35:end).^k,'g')
plot((linspace(0,1.05)),a*(linspace(0,1.05)).^k,'o ')
plot((linspace(0,1.05)),a*(linspace(0,1.05)).^1,'o ')
legend('Plastic data','Corresponding power law',sprintf('stress=%.3f
    {}strain^{%.3f}',a,k),sprintf('stress=%.3f{}strain^{%i}',a,1)');

hold off




figure(1)
plot(CG_Engineering_Strain,CG_Engineering_Stress,'o ');
xlabel('Engineering Strain')
ylabel('Engineering Stress [MPa]')
```

```
figure(2)
plot(CG_True_Strain,CG_True_Stress,' o')
xlabel('True Strain')
ylabel('True Stress [MPa]')


figure(12)
plot(CG_Engineering_Strain,CG_Engineering_Stress,'o ');
hold on
plot(CG_True_Strain,CG_True_Stress,' o')
hold off
%Enhance!
xlabel('Strain')
ylabel('Stress [MPa]')
legend('Engineering Stress strain','True Stress strain','Location','
    southeast')
title('CG Stress Strain Curves')
grid on

name = strcat('CG_facit','.png');
saveas(figure(12), name);
hold off

%% UFG
%   Matrix generator of tensile curves from figure 5 in Ni99.79@25
    degrees C
%        "Influence of impurities and deformation temperature on
%         the saturation microstructure and ductility of HPT deformed
    nickel."

%   Measurment Error + 0.5
c = 200/4.5;
UFG_Engineering_Stress = c.*[8.85;
    13.55;18.7;23.85;28.35;32.55;35.75;38;39.25; ...
     39.75;39.5;39.05;38.3;37.45;36.6;35.35;34.2;32.7;30.95];
c2 = 50/3.15;
UFGdispl = c2.*1e
    6.*[2.35;3.15;4.05;4.9;5.85;6.7;7.6;8.45;9.25;10.2;11.1;11.95;
    ...
     12.9;13.75;14.65;15.55;16.4;17.3;18.2];

gaugeLength = 2.5e 3;
UFG_Engineering_Strain = UFGdispl./gaugeLength;


A0 = 1;
V = A0*gaugeLength;
Aincr = V./(gaugeLength+UFGdispl);

[~,maxIndex] = max(UFG_Engineering_Stress);
UFG_True_Stress = UFG_Engineering_Stress(1:maxIndex)./Aincr(1:
    maxIndex);
UFG_True_Strain = log((UFGdispl(1:maxIndex)+gaugeLength)/gaugeLength)
    ;


%Power Law index:
logUFG_TStrain = log(UFG_True_Strain(end 2:end));
logUFG_TStress = log(UFG_True_Stress(end 2:end));
```

```
p = polyfit(logUFG_TStrain,logUFG_TStress,1);

figure(700)
plot(logUFG_TStrain,logUFG_TStress,'bo');
axis equal square
grid
xlabel('log(strain)');
ylabel('log(stress)');
k=p(1);
loga=p(2);
a=exp(loga);
hold on;
plot(logUFG_TStrain,k*logUFG_TStrain+loga,'g');
hold off
legend('Data',sprintf('stress=%.3f{}log(strain)+log(%.3f)',k,a));

figure(800)
plot(UFG_True_Strain(end 2:end),UFG_True_Stress(end 2:end),'bo');
xlabel('Strain');
ylabel('Stress [MPa]');
axis equal square
grid
hold on;
plot(UFG_True_Strain(end 2:end),a*UFG_True_Strain(end 2:end).^k,'g')
plot((linspace(0,1.05)),a*(linspace(0,1.05)).^k,'o ')
plot((linspace(0,1.05)),a*(linspace(0,1.05)).^1,'o ')
legend('Plastic data','Corresponding power law',sprintf('stress=%.3f
    {}strain^{%.3f}',a,k),sprintf('stress=%.3f{}strain^{%i}',a,1)');
hold off


%% Plots

figure(3)
plot(UFG_Engineering_Strain,UFG_Engineering_Stress,' o')
xlabel('Engineering Strain')
ylabel('Engineering Stress [MPa]')

figure(4)
plot(UFG_True_Strain,UFG_True_Stress,' o')
xlabel('True Strain')
ylabel('True Stress [MPa]')

figure(34)
plot(UFG_Engineering_Strain,UFG_Engineering_Stress,' o')
hold on
plot(UFG_True_Strain,UFG_True_Stress,' o')
xlabel('Strain')
ylabel('Stress [MPa]')
legend('Engineering Stress strain','True Stress strain','Location','
    southeast')
title('UFG Stress Strain Curves')
grid on

name = strcat('UFG_facit','.png');
saveas(figure(34), name);

hold off
```