NTNU

Norwegian University of
Science and Technology

Micromechanical Modelling and Simulation of Ductile Fracture in Metallic Materials

Sondre Bergo

Civil and Environmental Engineering (2 year)
Submission date: June 2016
Supervisor: Odd Sture Hopperstad, KT

Norwegian University of Science and Technology
Department of Structural Engineering

Department of Structural Engineering
Faculty of Engineering Science and Technology
NTNU- Norwegian University of Science and Technology

ACCESSIBILITY

MASTER THESIS 2016

SUBJECT AREA:	DATE:	NO. OF PAGES:
Materials Mechanics	10.06.2016	108 + 27

TITLE:

Micromechanical modelling and simulation of ductile fracture in metallic materials

Mikromekanisk modellering og simulering av duktilt brudd i metalliske materialer

BY:

Sondre Bergo

RESPONSIBLE TEACHER:	Prof. Odd Sture Hopperstad
SUPERVISOR(S)	Prof. Odd Sture Hopperstad and Ph.D. Lars Edvard Dæhli
CARRIED OUT AT:	NTNU

Department of Structural Engineering
FACULTY OF ENGINEERING SCIENCE AND TECHNOLOGY
NTNU – Norwegian University of Science and Technology

MASTER'S THESIS 2016

for

Sondre Bergo

Micromechanical modelling and simulation of ductile fracture in metallic materials

1. INTRODUCTION

In numerical simulations of structural components and structures in the ultimate limit load state or subjected to accidental loads, there is a need for constitutive models that describe the material behaviour at large deformations. In addition to modelling the stress-strain behaviour of the materials at large strains, the risk for ductile failure needs to be assessed in such simulations. The physical mechanisms governing ductile failure are nucleation, growth and coalescence of micro-voids and strain localization due to local bifurcation modes (loss-of-ellipticity). The topic of this thesis is micromechanical modelling and simulation of ductile fracture in metallic materials (aluminium alloys and steels) as well as analytical and computational methods for predicting evolution of damage and ductile failure in metallic structures.

2. OBJECTIVE

The objective is to formulate, implement and validate state-of-the art cellular automaton micro-mechanics based damage models for void nucleation, growth and coalescence in a stand-alone computer code. The computer code may then be used as a post-processing tool to assess the risk of ductile fracture in structural analysis based on the finite element method.

3. TASKS

The main topics in the research project will be as follows:

1. Literature study on ductile fracture in metallic materials: experiments, analytical methods, modelling and simulation.
2. Theoretical formulation of the cellular automaton micro-mechanics based damage models for void nucleation, growth and coalescence.
3. Numerical implementation of the cellular automaton model in a stand-along computer code.
4. Verification and validation of the implemented cellular automaton model using analytical solutions, experimental data from the literature, and finite element simulations of unit cells with distributed voids.
5. Numerical study on the influence of the model parameters on the predicted ductility.

Supervisors: Odd Sture Hopperstad, Lars Edvard Dæhli (NTNU)

The thesis must be written according to current requirements and submitted to the Department of Structural Engineering, NTNU, no later than June 11th, 2016.

NTNU, January 15th, 2016.

Odd Sture Hopperstad
Professor

Abstract

A Cellular Automaton Model for Calculation of Ductile Fracture

This thesis is concerned with the modeling of a cellular automaton model of ductile fracture in metals involving significant micro-structure heterogeneities, based on the growth equations of Rice&Tracey for the microvoids in metals.

The voids were given an initial size, orientation and position, which means that nucleation and growth were considered to be two separate phases of ductile growth. Further, the nucleation of all the microvoids were assumed completed at the moment the deformation started. Coalescence was defined to occur at the moment two voids were intersecting each other, and they got replaced by a new minimum volume enclosing ellipsoid, or with a scaled version of this new ellipsoid, as were thoroughly discussed. Elastic deformations were neglected. The ductility was defined as the value of the equivalent (plastic) strain at a given critical void volume fraction V_f^{crit}. $V_f^{crit} = 0.2$ was used in most of this thesis, but it became evident that the exact value of this parameter was of less importance.

Furthermore, it was seen that the probability distribution of the voids' positions introduced a variance into the fracture strain calculated by the model. The choice of probability distribution will evidently influence the results to a great extent.

Based on results obtained throughout the work with the model, it was concluded that to get converged results towards a physically meaningful value of the fracture strain without calibrating the model with results obtained from experiments, the biggest challenge is consequently to find a probability distribution that manages to generate a physical distribution of the microvoids as found in a given alloy.

Sondre Bergo

Sondre Bergo, Spring 2016, NTNU

Sammendrag

En cellulær automat modell for bestemmelse av duktilt brudd

Denne avhandlingen tar for seg modelleringen av en cellulær automat modell, for å bestemme duktilt brudd i materialer som har en betydelig andel mikrostrukturheterogeniteter, basert på likningen utviklet av Rice&Tracey for vekst av porene i metaller.

Porene ble gitt en opprinnelig størrelse, orientering og posisjon i rommet. Nukleering og vekst av porene ble betraktet som to separate faser i utviklingen av duktilt brudd, hvor all nukleeringen var antatt gjennomført før deformasjonen av materialet startet. Koalesens var definert til å inntreffe det øyeblikket to porer kom i kontakt med hverandre, hvorpå de ble erstattet av en minst mulig omsluttende ellipsoide.

Elastiske deformasjoner ble neglisjert. Duktilititeten ble definert som verdien på den ekvivalente plastiske tøyningen ved en gitt kritisk porevolumandel. Den kritiske volumandelen hulrom brukt i denne avhandlingen ble satt til $V_f^{crit} = 0.2$, men det kom klart fram i resultatene at den eksakte verdien på denne parameteren var av mindre betydning.

Videre ble det tydelig at sannsynlighetsfordelingen for porenes posisjoner introduserte en varians i bruddtøyningen kalkulert av modellen. Valg av sannsynlighetsfordeling vil altså påvirke resultatene i veldig stor grad.

Basert på resultater oppdaget gjennom arbeidet med modellen ble det konkludert med at for å få konvergerte resultater mot en fysisk meningsfull verdi på bruddtøyningen uten å måtte kalibrere modellen med resulateter fra forsøk, er den største utfordringen å finne en sannsynlighetsfordeling som klarer å generere en fysisk fordeling av porer som funnet i materialer.

Sondre Bergo

Sondre Bergo, Våren 2016, NTNU

Acknowledgements

The thesis presented herein was conducted at the Center for Research-based Innovation, Structural Impact Laboratory (SIMLab) which is housed at the Department of Structural Engineering at the Norwegian University of Science and Technology during the spring 2016.

I would like to express my gratitude to my supervisors:

Professor Odd Sture Hopperstad for his invaluable help. Whenever one of the algorithms or the behaviour of the model didn't work as expected, he was always there to guide me. Individual cases are too numerous to begin to mention, so I will content myself with saying that this thesis wouldn't have been done in time without his help.

PhD Candidate Lars Edvard Dæhli for his excellent understanding of how the microvoids behave, which really helped with how I should implement the model. The results from Abaqus are from him, and he also had suggestions for improvement of the layout of the thesis. He was always willing to help; several of the almost weekly meetings continued with him further explaining problems we had just discussed at the meeting.

Contents

1 Introduction

1.1 Motivation

Ductile fracture is preferred to brittle fracture in most applications, because of the extensive plastic deformation and energy absorption before fracture [1]. Through ductility, materials are able to absorb large deformations far beyond the elastic limit. The goal of plastic design is to utilize the reserve strength beyond the elastic limit due to the redistribution of internal forces [2]. If not being able to fully exploit this feature of ductility in materials, at worst a considerable sacrifice of economy is made [2].

Today there are many methods to try to calculate when materials will experience fracture. "Fracture Mechanics" is an entire field of mechanics, about the study of the propagation of cracks in materials. The concept of "fracture toughness" is a quantitative way of expressing a material's resistance to "brittle" fracture when a crack is present [3]. If a material has high fracture toughness, it will probably undergo "ductile" fracture. The ductility is a materials ability to deform under tensile stress, i.e to which extent it can be plastically deformed without fracture [3]. The ductility of a given material is often defined in terms of the equivalent plastic strain at fracture, which it also is defined as in this thesis.

There is a potential for improvement in the calculation of the ductility of alloys, see for instance the review by Benzerga and Leblond [4]. This thesis will try a relatively new approach in an attempt to determine the ductility of metallic materials.

1.2 Objective

The objective of this thesis is to understand and model the effect of microstructure heterogenities on damage accumulation in metallic alloys, through implementation of a cellular automaton model in Python. The implemented model should be verified and validated, and used to predict the ductility based on a given initial configuration. The sub-objectives are:

- A literature study
- The establishment of the mathematical foundation

- The implementation in Python
- A verification by comparing with analytical solutions
- A validation process through comparison with numerical solutions from Abaqus, and different case studies obtained from the model itself.
- A parameter study

The scope of this thesis is confined to:

- Only growth and coalescence of the microvoids are considered.
- The microvoids are assumed *initially nucleated*. This is a conservative simplification, which should yield a lower ductility than obtained from experiments.

1.3 Overview of Thesis

This thesis report is divided into chapters which presents the development of the cellular automaton model. A short description of each chapter is found below.

Chapter 2: Ductile Fracture Observations Chapter 2 presents how ductile fracture happens, by explaining in depth the process of nucleation, growth and coalescence of microvoids.

Chapter 3: Theory Chapter 3 contains a presentation of the theoretical foundation which is needed to fully understand the concepts and results presented in this thesis.

Chapter 4: The Cellular Automaton Model Chapter 4 presents the cellular automaton model starting with the overall framework, continuing with flowcharts, and ending with the pseudo-code of the primary parts of the program. The actual source code implementation is provided as an appendix.

Chapter 5 : Verification Chapter 5 presents the verification process of the implemented model. An outline of the preliminary controls made during the development are given, and a discussion of the main parts of the program, how they were solved, and possible sources of error is also conferred.

Chapter 6: Validation Chapter 6 compares some of the implemented functions to results obtained from Abaqus. It then continues by comparing how much the orientation of the local coordinate system introduces an error in the calculations. A parameterstudy and a sensitivity analysis is also conducted. At last, the model is used to calculate a fracture locus for the two outer scenarios; when coalescence is taken into account, and when it is not.

Chapter 7: Extension of Modeling Framework Chapter 7 presents concrete suggestions for future development of the model.

Chapter 8: Concluding Remarks Chapter 8 presents a summary of the results produced in this thesis along with concluding remarks.

2 Ductile Fracture Observations

2.1 Introduction

The most common fracture mehcanisms in metals and alloys are ductile fracture, cleavage and inter-granular fracture (and fatigue) [5]. Cleavage fracture involves separation along specific crystallographic planes, with a trans-granular fracture path. Inter-granular fracture occurs when the grain boundaries are the preferred fracture path in the material. This thesis is about ductile fracture, therefore only ductile fracture will be discussed further in the text.

Ductile fracture results from the nucleation, growth and coalescence of small internal cavities, which are called "microvoids". Microvoid nucleation is usually associated with the fracture of brittle particles or interface decohesion ([6] and [7]), and is the "creation" of cavities at positions without cavities just moments before. In short, microvoids may initiate at inclusions and second-phase particles. After nucleation, these microvoids grow by plastic deformation of the surrounding matrix [4].

The important role played by void nucleation, growth and coalescence in ductile fracture was understood in the late 1940s [8]. But, it was not until the year 1960 that the phenomenology of this process became well documented [9]. During the last approximately four decades, ductile fracture has been a subject of intense investigation [4]. The first micromechanical treatments of ductile fracture was due to McClintock [10] and Rice&Tracey [11], and considered isolated voids and how they grew. After the nucleation of voids, stable growth proceeds until plasticity localizes within the ligament between closely spaced microvoids setting the onset of the coalescence mechanism [12]. The accumulation of microvoid linkages leads to the formation of a dominant microscopic crack that finally propagates into the entire material, and fracture occurs.

The understanding and modeling of the different steps of the process of nucleation, growth, and coalescence of voids have been improved over the last five decades, as covered in several recent review papers, see f.ex [4], [13] and [14]. Many experimental investigations show for instance that the ductility depends considerably on the triaxiality of the stress state [15].

However, major questions remain about the understanding and proper modeling of ductile fracture in metals involving significant micro-structure heterogeneities, like several types of alloys. This introduction proceeds with a review of the main issues related to heterogeneities and how they affect damage evolu-

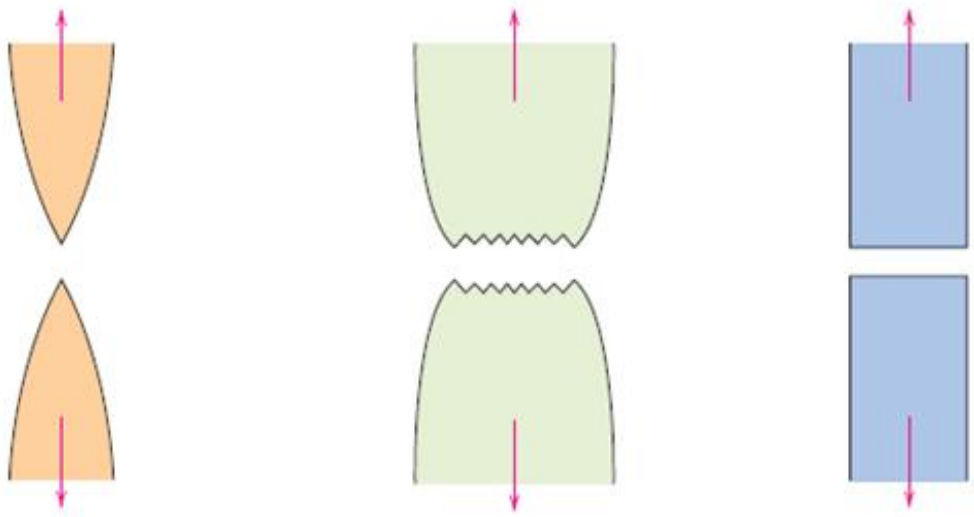
tion and complexifies the application of existing models.

Void nucleation is a discontinuous process of successions of discrete nucleation events [13]. The effect of particle size, and also the distribution of internal or inter-facial defects, can be a dominant factor of the inhomogeneity of the damage process. The particle shape relative to the loading orientation can also contribute to the heterogeneity of the nucleation process [16] and [17], which means microvoids can nucleate at very different times during the deformation. Voids nucleated by interface decohesion are initially somewhat like an ellipsoid, and tend to elongate in the principal loading direction. Heterogeneities in the void distribution do not significantly affect the void growth rates, which means that the interaction between the voids during the growth phase is weak when the volume fraction of constituent particles are low, as it usually is in industrial alloys ([18], [19],[20] and [21]). This means that the assumption of no interactions between the voids before coalescence may seem like a valid assumption, at least until the voids are quite close to each other.

The impact of heterogeneities on damage evolution until fracture can be modeled in many different ways, see the review [14]. In this thesis, a quite sophisticated approach was chosen, which consists of generating a large ensemble of microvoids, and simulate the latter two of the three steps of the damage process; growth and coalescence.

2.2 Ductile Fracture

Figure 2.1 shows the differences between ductile and brittle fracture. Ductile fracture is almost always the preferred form of fracture, since ductile materials generally are tougher, as mentioned in section 1.1. To easier explain the process of ductile fracture; consider a uniaxial tensile test. Fig 2.2 shows force vs dis-



(a) Highly ductile fracture (b) Moderately ductile fracture (c) Brittle fracture

Figure 2.1: Fracture may be divided in ductile or brittle fracture

placement, which means the plot is unaltered if engineering stress and strains

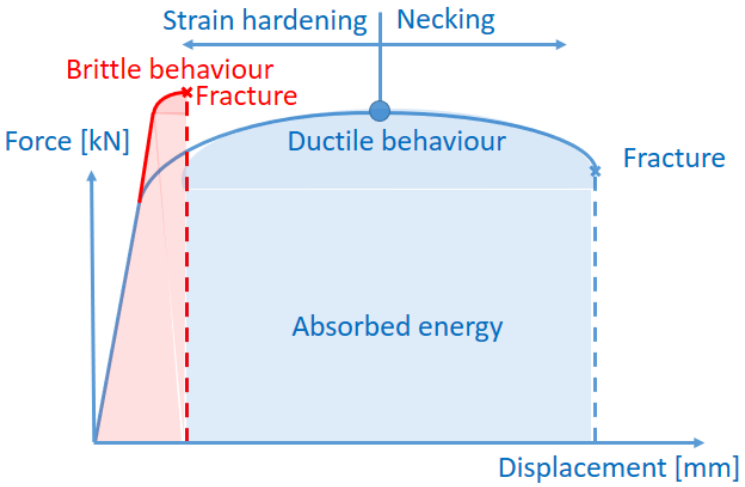


Figure 2.2: A perfectly plastic material

were measured instead. Engineering values means that stresses and strains are defined in terms of the initial cross sectional area and initial specimen length, respectively. The material eventually reaches an instability point, where strain hardening no longer can keep up with the loss of cross-sectional area, and a necked region forms beyond the maximum load. Depending on the purity of the material, the neck may go down to a sharp point (see Fig 2.1), which means the stress-strain curve approaches zero before fracture occurs.

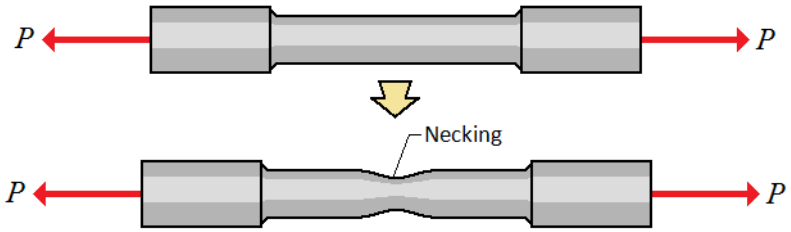


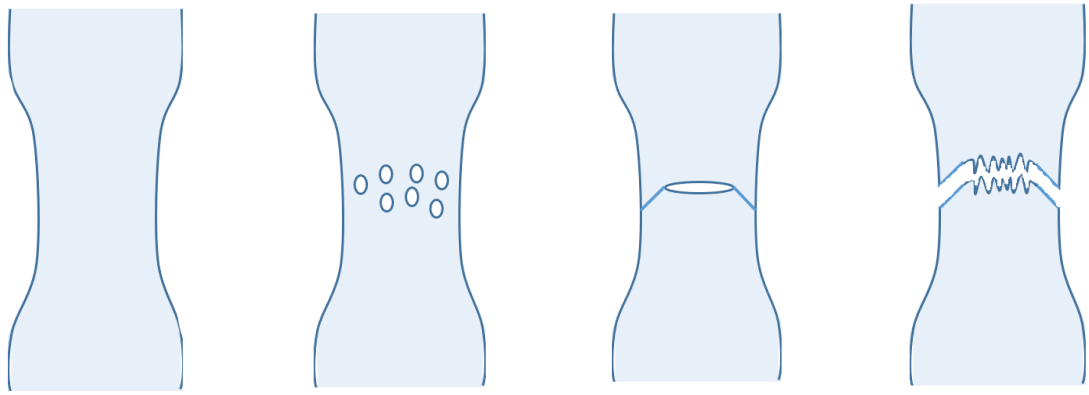
Figure 2.3: Necking in a material specimen, [22]

If the cross-sectional area are reduced to only a small fraction of the initial area, it results in extremely large local plastic strains. The less impurities, the more cross sectional area reduction will happen before fracture. But pure materials are seldom used in the industry. Steel for instance, is an alloy of iron and other elements, primarily carbon [23]. Carbon, other elements, and inclusions within iron acts as hardening agents that prevent the movement of dislocations that otherwise occur in the crystal lattices of iron atoms. These elements are what is called particles in the matrix. These particles are what gives steel alloys its hardness and excellent tensile strength, but they are also the source of the microvoids that nucleates [24]. Steel alloys are an excellent material to be used in structures, but the introduction of these particles is also the reason that predicting ductile fracture is considered a difficult task.

The commonly observed stages in ductile fracture are:

1. Necking (localization of plastic strain).
2. Formation of free surfaces at an inclusion/particle by either interface decohesion or particle cracking (nucleation of voids).
3. Growth of the voids around the particle, by means of plastic strain and hydrostatic stress.
4. Coalescence of some of the growing voids with adjacent voids.
5. Fracture.

These steps may be summarized by Figure 2.4. When the neck occurs, the stress state becomes triaxial, and the triaxiality increases from $\sigma^* = 1/3$ in a uniaxial stress state to maybe as much as $\sigma^* = 3$ at the most in the center of the neck. Due to this large elevation of the triaxiality, voids will nucleate much easier and grow much faster inside this neck compared to outside the neck. Closer to the outer edge of the test-specimen, the triaxiality is significantly lower, so in these locations there are both much fewer and smaller voids. When fracture occurs, this part of the cross-section will experience shear fracture, which results in the characteristic "cup-and-cone" fracture shown in Fig 2.5.



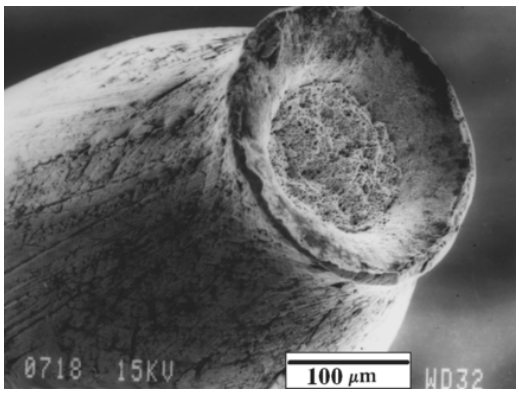
(a) Necking occurs (b) Nucleation and growth (c) Coalescence (d) Fracture

Figure 2.4: Ductile fracture process summarized

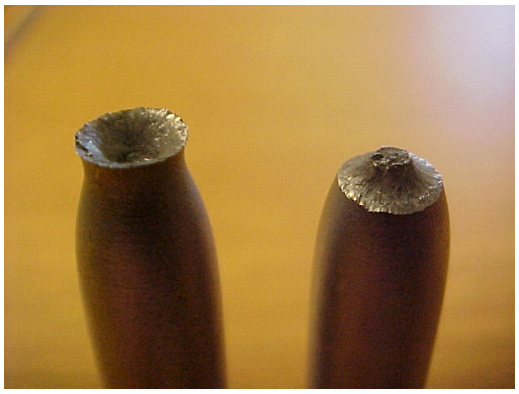
2.3 Void Nucleation

As mentioned in section 2.1, a void forms around a second-phase particle or inclusion when the sufficient stress is applied to break the interfacial bonds between the particle and the matrix [5]. The size of the microvoids usually varies between $0.01\mu m$ to $>> 1\mu m$ [15].

The energy released in the void nucleation must be sufficient to create the new surface that appears. Each particle may for instance be assigned an associated



(a) SEM fractograph of the fracture surface [25]



(b) Photograph of fracture surface [26]

Figure 2.5: "Cup and Cone" fracture

critical principal stress value, and when this value is reached, the particle will then nucleate into a microvoid [27].

A number of models for estimating void nucleation stress have been published. The most widely used continuum model for void nucleation is due to Argon et al. [7]. They argued that the inter facial stress at a cylindrical particle is approximately equal to the sum of the mean (hydrostatic) stress σ_m and the effective stress, defined as the vonMises stress: $\sigma_e = \sigma_{VM}$. The decohesion stress is defined as a critical combination of these two:

$$\sigma_c = \sigma_e + \sigma_m$$

The nucleation strain decreases as the hydrostatic stress increases. That is, void nucleation occurs more readily in a triaxial tensile stress field, a result that is consistent with experimental observations. In this thesis, the implemented model assumes already nucleated voids at the initial time step, so no implementation of nucleation of voids are included.

Theories of ductile fracture generally assume uniformity of particle size, shape and spacing, and simoultanous nucleation of microvoids at all particles [15]. This is the principle behind the "unit cell model", where it is possible to (and also limited to) consider only one microvoid. This is a simplification, since there may be significant variations, and nucleation and growth are not separate and sequential processes. Hannard et al. [27] concluded that the particle distribution is the most significant factor when trying to determine the fracture strain. In contrast to an ideal periodic distribution, a physical distribution (f.ex obtained as the result of a CT scan, also called X-ray computed tomography, of the matrix material) will always have some degree of clusters. A cluster is a group of independent voids that is located closer to each other somewhere in the matrix compared with other parts of the matrix. The spatial distribution of particles influences the void coalescence process due to its effect on the local void spacing. A periodic distribution is an idealized situation, and will result in higher ductility, since local clusters will not be present [5].

2.4 Void Growth

Once voids nucleate, further plastic strain and hydrostatic stress cause the voids to grow and eventually coalesce. At some point, the voids may interact, which will be discussed in section 2.5.

There are a number of mathematical models for void growth and coalescence, where the two most widely referenced models were published by Rice&Tracey [11], and Gurson [28]. Rice&Tracey considered a single void in an infinite solid. The void is subjected to remote normal stresses: σ_1, σ_2 and σ_3, and remote normal strain rates $\dot{\epsilon}_1$, $\dot{\epsilon}_2$ and $\dot{\epsilon}_3$. The initial void is assumed to be spherical, but it becomes ellipsoidal as it deforms. They developed a set of equations for the growth of a spherical void of radius R_0 in a remote strain field

$$\varepsilon_{ij} \tag{2.1}$$

and remote stress field

$$\sigma_{ij} = \sigma_{ij}^{'} + \sigma_m * \delta_{ij}. \tag{2.2}$$

The analysis was performed for a rigid-plastic non-hardening material, and an estimate for linear, isotropic hardening was also made. They differ only in a constant parameter, so in this thesis only rigid-perfectly plastic material behavior is considered. The void growth equations have the form:

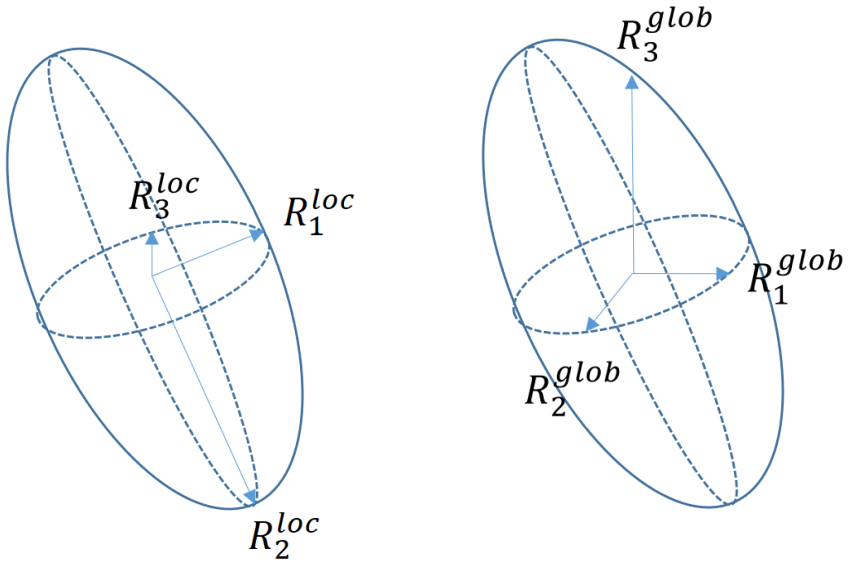
$$\dot{R_k}/R_0 = (1+E)\dot{\varepsilon_k} + D\sqrt{\frac{2}{3}\dot{\varepsilon_l}\dot{\varepsilon_l}} \qquad \text{k,l} = 1,2,3 \tag{2.3}$$

$\dot{R}_k$ are the rates of change in void radius in the directions x, y, and z defined by the principal strain rates of the remote strain field.

An heuristic procedure is proposed in this thesis (inspired by [27]) in order to estimate the void growth rates in the case of an ellipsoidal void under general loading conditions. The principal radii of the ellipsoidal void are replaced by the three intersections of the void with the principal stress (loading) directions, see Figure 4.1. This equivalent void is then assumed to follow the growth model given by Eq (2.3). In a general case, it is also necessary to replace R_0 by the mean void radius:

$$R_{mean} = \frac{R_1 + R_2 + R_3}{3} \tag{2.4}$$

which is the arithmetic mean. It is further assumed that the principal axes of the remote strain-rate field remain fixed in direction during the deformations, so the loading conditions are proportional and may be characterized by a constant Lode variable Θ_L (see Chapter 3 for definition). Since the Rice&Tracey model is based on a single void, it does not take into account interactions between voids, nor does it predict ultimate failure. A separate failure criterion must be applied to characterize microvoid coalescence [27]. An analytical solution is readily obtained if we can assume constant stress triaxiality σ^* and lode angle Θ_L, which



(a) Local/principal directions of the semi-axes

(b) Global directions of the semi-axes

Figure 2.6: Local and global semi-axes' directions

the numerical results are controlled against:

$$
\begin{aligned}
R_1 &= R_{mean}(A + \frac{3+\Theta_L}{2\sqrt{3+\Theta_L^2}}B) \\
R_2 &= R_{mean}(A - \frac{\Theta_L}{\sqrt{3+\Theta_L^2}}B) \\
R_3 &= R_{mean}(A + \frac{\Theta_L - 3}{2\sqrt{3+\Theta_L^2}}B)
\end{aligned}
\tag{2.5}
$$

where

$$
\begin{aligned}
A &= exp(\frac{2\sqrt{3+\Theta_L^2}}{3+\Theta_L}D\epsilon_1) \\
B &= (\frac{1+E}{D})(A-1) \\
D &= \begin{cases} \alpha exp(\frac{3}{2}\sigma^*) & \text{if } \sigma^* \geq 1.0 \\ \alpha \sigma^{*0.25} exp(\frac{3}{2}\sigma^*) & \text{if } \sigma^* < 1.0 \end{cases} \\
E &= \frac{2}{3} \\
\alpha &= 0.427
\end{aligned}
\tag{2.6}
$$

For cases when the lode parameter isn't constant, numerical integration must be used if the analytical solution is desired. The trapezoidal rule is a well known technique for approximating integrals like this, see Chapter 3.

2.5 Void Coalescence

This is the final stage of ductile fracture. At some point, neighboring voids interact. Plastic strain is concentrated between adjacent voids, and local necking instabilities develop. As a failure criterion, it is most often assumed that fracture occurs when a critical void volume fraction is reached [5]. The Gurson model [28] (modified by Tvergaard [29], who is behind one of the many expansions of the Gurson model) analyzes the plastic flow in a porous medium by assuming that the material behaves as a continuum. In the Gurson model, the effect of the voids is averaged through the material. The yield surface exhibits a hydrostatic stress dependence in the Gurson-Tvergaard model, in contrast to classical plasticity theory. This model also contains a fracture criterion; ductile fracture is assumed to occur as the result of a plastic instability that produces a band of localized deformation, which occurs more readily in a Gurson-Tvergaard material because of the strain softening induced by the hydrostatic stress. However, because the model does not consider discrete voids, it is unable to predict the necking instability between voids.

In this thesis, a finite number of voids in a matrix material is considered, and this way the interaction between voids are taken into account. To summarize, microvoids coalesce when adjacent microvoids link together or the material between microvoids experiences necking.

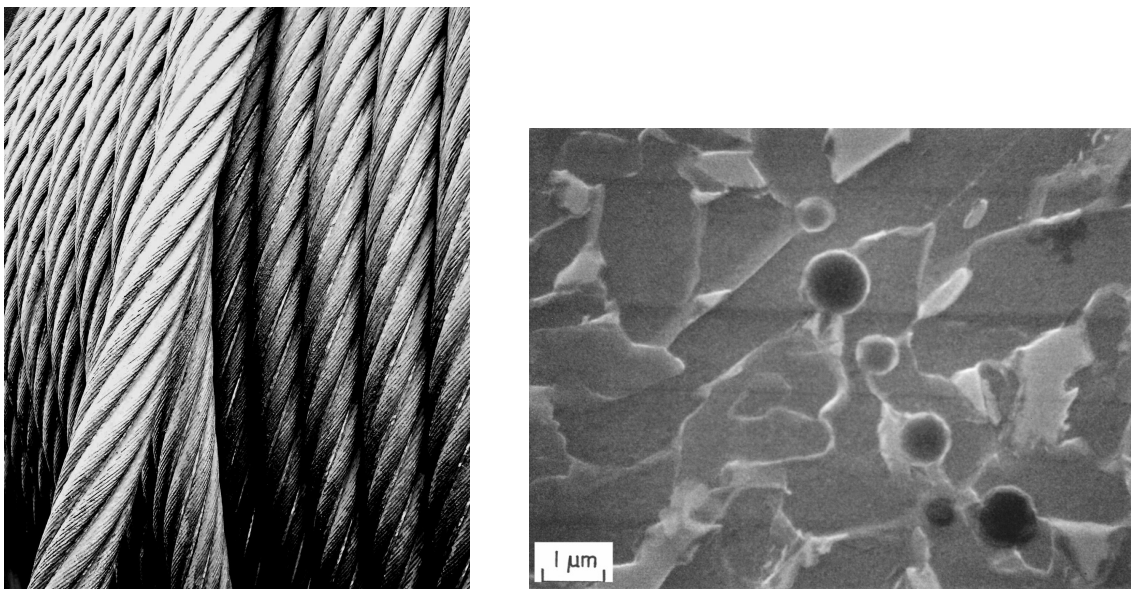
3 Essential Theory

Expected Prior Knowledge

When reading this thesis, the reader is presumed to have some prior knowledge in the fields: materials mechanics, continuum mechanics, computer programming and linear algebra. The following sections are meant as a refreshment of prior knowledge, to remind the reader of the theories which are needed to fully understand all the aspects of the model. The theory presented in the following sections are based upon the listed sources.

3.1 Materials Science

On Alloys as a Material



(a) Steel as our eyes see it [30] (b) Steel under a microscope [31]

Figure 3.1: Macroscopic and microscopic view of steel

An alloy is a mixture of ideally pure chemical elements, which forms an impure substance that keeps the characteristics of a metal. The added impurities are

usually desirable and will typically have some useful benefit. Alloys are made by mixing two or more elements, at least one being a metal, which is usually called the base metal for the alloy. When the molten elements are mixed, and cools into a solid, the mechanical properties will often be quite different from its individual components. Adding a small amount of non-metallic carbon to iron produces an alloy called steel. Due to its very high strength and toughness (which is much higher than pure iron), and its ability to be greatly altered by heat treatment, steel is one of the most common alloys in modern use.

The alloying elements (such as carbon, manganese, chromium, nickel, tungsten etc) acts as hardening agents that prevent the movement of dislocations that otherwise occur in the crystal lattices of iron atoms. Pure iron is very ductile and relatively weak, where steel is much less ductile but much stronger (higher tensile strength). A dislocation is a chrystallographic defect (irregularity) within a crystal structure (such as a metal, among others).

By varying the amount of the alloying elements, it is possible to control qualities such as hardness and ductility, by restricting the movement of these dislocations. These alloying elements is in this thesis referred to as *second-phase particles*, or just *particles*, which is the main reason for the nucleation of microvoids, see Chapter 2.

The other source of nucleation of voids, is *inclusions*. An inclusion in an alloy are either chemical compounds or present nonmetals. They are the product of physical effects or chemical reactions, and are categorized as either *endogenous* or *exogenous*. Endogenous inclusions occur within the material as a result of chemical reactions during cooling, and are typically very small. Exogenous inclusions are caused by entrapment of non-metals and may vary greatly in size, and their source can include slag, dross, flux, or even pieces of the mold. These inclusions/particles are the cause of the creation of the microvoids, through the nucleation process. The reader is referred to [23] and [24] for more on this topic.

3.2 Mathematical Foundation

Main source here is [32], unless stated otherwise.

3.2.1 Matrices (n-Dimensional Arrays)

A two-dimensional matrix is a table or rectangular array of elements arranged in rows and columns, and is herein expressed as a bold upper case Latin letter:

$$\begin{bmatrix} A_1 1 & A_1 2 & A_1 3 \\ A_2 1 & A_2 2 & A_2 3 \end{bmatrix} \equiv [A_{\alpha i}] \equiv \mathbf{A} \tag{3.1}$$

The elements may be numbers or functions. The symbol $A_{\alpha i}$ represent an arbitrary element for which the row number (α, the first subindex) here may take the value 1 or 2, while the column number (i, the second subindex) may take values 1,2 or 3. Lower case Greek letter indices represents the numbers 1 and 2, while lower case Latin letter indices represents the numbers 1, 2 and 3. A one-dimensional matrix is an array of elements arranged in a column (unless stated

otherwise), and is expressed as a bold lower case Latin letter:

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \equiv [a_i] \equiv \mathbf{a} \tag{3.2}$$

From a two-dimensional matrix $\mathbf{A} = [A_{ij}]$ a transposed matrix is constructed as $\mathbf{A^T} = [A_{ji}]$, by interchanging columns and rows:

$$A_{ij}^T = A_{ji} \tag{3.3}$$

An n-dimensional matrix is represented by a set of elements with n indices, f.ex C_{ijk} is a three dimensional matrix. Addition of matrices is defined only for matrices of same size, and are obtained by adding corresponding elements:

$$C_{ij} = A_{ij} + B_{ij} \tag{3.4}$$

The product of a matrix $\mathbf{A} = [A_{ij}]$ by a term α is a matrix $\alpha\mathbf{A} = [\alpha A_{ij}]$. The matrix product of two matrices **A** and **B** is a new matrix **C**:

$$\mathbf{AB} = \mathbf{C} \equiv \sum_{k=1}^{3} A_{ik}B_{kj} = C_{ij} \tag{3.5}$$

Note that in general $\mathbf{AB} \neq \mathbf{BA}$. From the definition in Eq 3.5 it follows that $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$. A symmetric n × n real matrix **A** is said to be positive definite if the scalar $\mathbf{z}^T\mathbf{Az}$ is positive for every non-zero column vector **z** of real numbers. Here $\mathbf{z^T}$ denotes the transpose of **z**. A positive definite matrix has some very interesting properties, where the most important is that:

$$\mathbf{A^T} = \mathbf{A} \tag{3.6}$$

The Kroenecker Delta

The kroenecker delta δ_{ij} represents the identity matrix, where:

$$\delta_{ij} = \delta_{ji} = \begin{cases} 1, & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{3.7}$$

Einsteins Summation Convention

An index repeated once and only once in a term implies a summation over the number region of that index. By this convention, the following are equivalent statements:

$$\sum_{k=1}^{3} A_{ik}B_{kj} \equiv A_{ik}B_{kj} \tag{3.8}$$

A summation index (as k above) is also called a dummy index because it may be replaced by a different letter without changing the result of the summation.

3.2.2 Quadratic Forms

A quadratic form is a homogeneous polynomial of 2^{nd} degree in a given number of variables. For example, $x^2+2y^2+4xyz+3z^2$ is a quadratic form of the variables x, y and z. A quadratic surface is given by the general equation:

$$ax^2 + by^2 + cz^2 + 2fyz + 2gzx + 2hxy + 2px + 2qy + 2rz + d = 0 \tag{3.9}$$

Ellipsoids

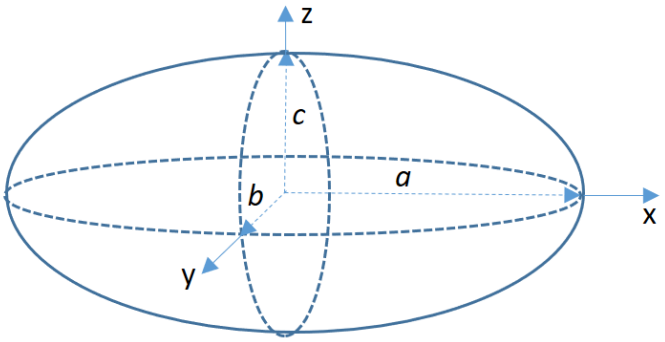


Figure 3.2: Ellipsoids with center in origo

An ellipsoid is an example of a quadratic surface. An ellipsoid centered in origo and with principal axes aligned with the coordinate system's axes, has a standard equation (a *quadratic form*) given as:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1 \tag{3.10}$$

Here a, b and c are the lengths of the principal semi-axes. This quadratic form may be written on a *generalized form*:

$$(\mathbf{x})^T \mathbf{A}\mathbf{x} = 1 \tag{3.11}$$

where $\mathbf{A}$ is a positive definite matrix. Its eigenvectors define the principal axes of the ellipsoid, and its eigenvalues are the reciprocals of the squares of the semi-axes: a^{-2}, b^{-2} and c^{-2} (see under subsection 3.2.4, *eigenvectors and eigenvalues*). If the ellipsoid is centered in $\mathbf{C}$, the generalized form becomes:

$$(\mathbf{x} - \mathbf{C})^{\mathrm{T}} \mathbf{A} (\mathbf{x} - \mathbf{C}) = 1 \tag{3.12}$$

The last thing that should be known about ellipsoids, is that the volume is given by

$$V = \frac{4\pi abc}{3}. \tag{3.13}$$

3.2.3 Discriminant

The discrimant of a polynomial is a function of its coefficients, typically denoted Δ, which gives information about its roots. In this thesis, since ellipsoids are

given on quadratic form, only discriminants of 2^{nd} order will be considered. For a general 2^{nd} order polynomial:

$$ax^2 + bx + c = 0 \tag{3.14}$$

the discriminant is given as:

$$\Delta = b^2 - 4ac \tag{3.15}$$

This is a very useful quantity to determine if the ellipsoids are intersecting planes, such as the box's 'walls'. In Fig 3.3, it is shown the three scenarios where the discriminant may be negative, equal to zero, or positive, respectively. For a negative discriminant, the solutions are imaginary numbers, which physically means no intersection. A discriminant equal to zero means one physical (real) solution, while a positive discriminant means two physical solutions as long as we are limited to 2^{nd} order polynomials and intersecting lines. When intersecting with planes, there will of course be a whole *set* of solutions. In Fig 3.3, the black points refers to imaginary solutions, while the red ones are meant to symbolize the real solutions.

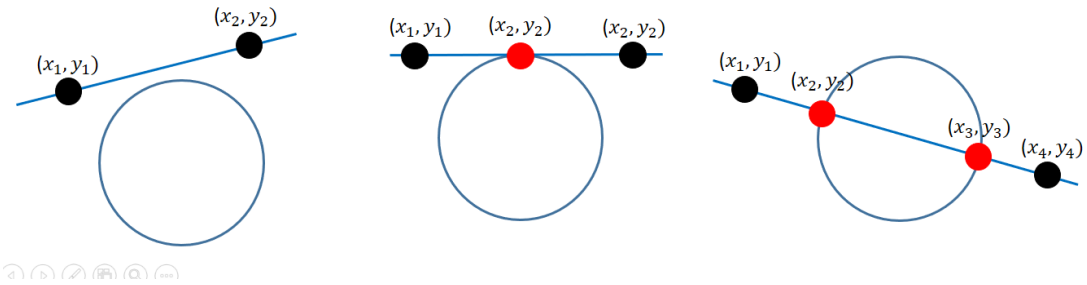


Figure 3.3: Physical interpretation of the discriminant

3.2.4 Tensors

Tensors are geometric objects that describe linear relations between geometric vectors (like for instance between the displacement vector $\mathbf{u}$ and $\bar{\mathbf{u}}$, described in different coordinate systems) , scalars and other tensors. A tensor can be represented as an multidimensional array of numerical values, where the order of the tensor is the dimensionality of the array needed to represent it. Tensors are invariant to the particular choice of coordinate system. In this thesis, 2^{nd} order tensors are extensively used to transform between a global and a local coordinate system. To establish how a vector transforms under the change of orthonormal basis, first the direction cosines between two sets of unit vectors $\mathbf{e_j}$ and $\bar{\mathbf{e}}_\mathbf{i}$ must be defined:

$$Q_{ij} = \bar{\mathbf{e}}_\mathbf{i} \cdot \mathbf{e_j} = \bar{\mathbf{e}}_\mathbf{i}\mathbf{e_j}\cos(\theta) = \cos(\theta) \tag{3.16}$$

Where θ is the angle between the coordinate axes $\bar{\mathbf{x}}_\mathbf{i}$ and $\mathbf{x_j}$. Q_{ij} are the components of the unit vectors $\bar{\mathbf{e}}_\mathbf{i}$ decomposed onto the unit vectors $\mathbf{e_j}$, and similarly are Q_{ji} the components of $\mathbf{e_i}$ onto $\bar{\mathbf{e}}_\mathbf{j}$, and we have the relations:

$$\bar{\mathbf{e}}_\mathbf{i} = Q_{ij}\mathbf{e_j} \quad , \quad \mathbf{e_j} = Q_{ij}\bar{\mathbf{e}}_\mathbf{i} \tag{3.17}$$

The matrix made up of the components Q_{ij} are called the transformation matrix $\mathbf{Q}$ for the coordinate transformation from Ox (spanned by the base vectors $\mathbf{e_j}$) to $\bar{O}\bar{x}$ (spanned by $\bar{\mathbf{e}}_\mathbf{i}$). The rows in $\mathbf{Q}$ represent the components of the orthogonal vectors $\bar{\mathbf{e}}_\mathbf{i}$ in the Ox system, and the columns in $\mathbf{Q}$ represents the components of $\mathbf{e_j}$ in the $\bar{O}\bar{x}$ system. The transformation matrix $\mathbf{Q}$ has the following properties:

$$Q_{ik}Q_{jk} = \delta_{ij} \quad \Leftrightarrow \quad \mathbf{Q}\mathbf{Q}^T = 1 \quad \Leftrightarrow \quad \mathbf{Q}^{-1} = \mathbf{Q}^T \tag{3.18}$$

$$\det\mathbf{Q} = 1 \tag{3.19}$$

A matrix with properties like this are called an orthogonal matrix. $\mathbf{Q}$ is a linear operator, and a tensor of 2^{nd} order. To transform between coordinate systems, just replace the coordinate variable according to Eq (3.17). The coordinate transformation formula becomes:

$$\bar{x}_i = \bar{c}_i + Q_{ij}x_j \quad \Leftrightarrow \quad \bar{\mathbf{x}} = \bar{\mathbf{c}} + \mathbf{Q}\mathbf{x} \tag{3.20}$$

The inverse transformation is easily found to be:

$$\mathbf{x} = -\mathbf{c} + \mathbf{Q^T}\mathbf{x} \tag{3.21}$$

To transform the ellipsoid surface given in generalized form (Eq (3.11)) between a global (Ox) and a local ($\bar{O}\bar{x}$) coordinate system, the relation between $\mathbf{A}$ and $\bar{\mathbf{A}}$ should be derived. Start with the generalized form:

$$(\mathbf{x})^T\mathbf{A}\mathbf{x} = 1$$

and replace $\mathbf{x}$ according to Eq (3.17):

$$\begin{aligned}(\mathbf{Q^{-1}}\bar{\mathbf{x}})^T\mathbf{A}\mathbf{Q^{-1}}\bar{\mathbf{x}} &= 1\\ (\bar{\mathbf{x}})^T\mathbf{Q^{-T}}\mathbf{A}\mathbf{Q^{-1}}\bar{\mathbf{x}} &= 1\\ (\bar{\mathbf{x}})^T\mathbf{Q}\mathbf{A}\mathbf{Q^T}\bar{\mathbf{x}} &= 1\end{aligned} \tag{3.22}$$

which means that:

$$\bar{\mathbf{A}} = \mathbf{Q}\mathbf{A}\mathbf{Q^T} \tag{3.23}$$

Eigenvectors and Eigenvalues

Also called principal vectors and principal values. In linear algebra, an eigenvector of a linear transformation is a non-zero vector that does not change its direction when that linear transformation is applied to it. I.e, if $\mathbf{v}$ is a vector that is not the zero vector, then it is an eigenvector of a linear transformation T if T($\mathbf{v}$) is a scalar multiple of $\mathbf{v}$. This can be written as the equation:

$$T(\mathbf{v}) = \lambda\mathbf{v}, \tag{3.24}$$

where λ is a scalar known as the eigenvalue (or principal value) associated with the eigenvector $\mathbf{v}$. If the linear transformation T is expressed as a square matrix $\mathbf{A}$, then the equation can be expressed as the matrix multiplication:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} \tag{3.25}$$

If this holds for a given direction $\mathbf{v}$, then $\mathbf{v}$ is an eigenvector and λ a corresponding eigenvalue to $\mathbf{A}$. This relation may be written on a form thats makes it easier to actually find these vectors and values:

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = \mathbf{0} \tag{3.26}$$

Where a non-trivial solution for $\mathbf{v}$ and λ requires that:

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0 \tag{3.27}$$

For a symmetric matrix with real components, the eigenvectors will be mutually orthogonal if they are all distinct. If the matrix $\mathbf{A}$ is positive definite, then all the eigenvectors will be distinct. An ellipsoid (see subsection 3.2.2) is uniquely determined by a positive definite matrix, which means that its three semi-axes are uniquely defined by its eigenvectors.

3.2.5 Numerical Integration - Trapezoidal Rule

The Trapezoidal rule is a technique for approximating definite integrals:

$$\lim_{\Delta x \to 0^+} \sum f(x)\Delta x \Rightarrow \int f(x)\,dx. \tag{3.28}$$

The trapezoidal rule approximates the area under some small part of the function as a trapezoid, calculates that small area, and adds all the contributions between the integral limits together, commonly known as the Riemann sum. When the limit $\Delta x \to 0^+$ is reached, the exact value of the integral is obtained. In numerical mathematics this is impossible, but *very* accurate results may be obtained by using many sample points.

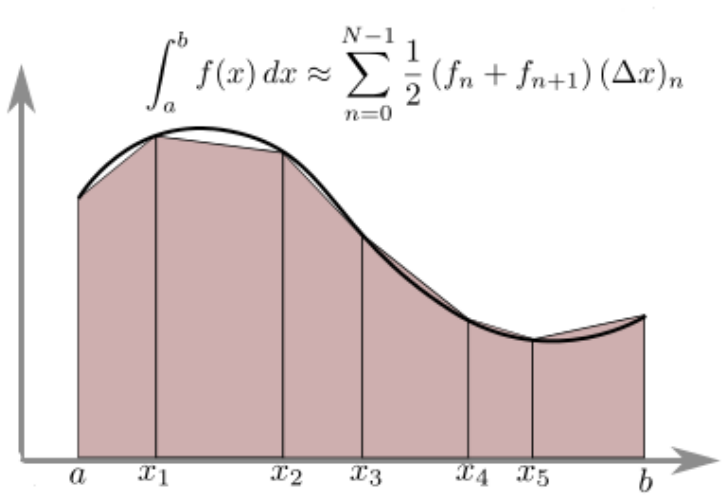


Figure 3.4: Trapezoidal Rule [33]

3.2.6 Numerical Solvers of Differential Equations

The Rice&Tracey growth equation, see 2.3, is solved numerically as a linear first order differential equation (an Ordinary Differential Equation, ODE) as an initial value problem. The initial boundary condition is given as the value of the semi axes in the global coordinate directions.

There exists a lot of different numerical procedures for solving ODEs, but the most common choice is usually an explicit method of some desired order. In this thesis, Euler (1. order), Heun (2.order) and Runge-Kutta 4 (4.order) was implemented to study the differences between some of the most common explicit methods for numerical integration of ODEs. All the explicit numerical integration methods discussed here only work on linear first order ODEs, where some tweaks must be made to get a correct solution for non-linear or higher order equations. This will not be discussed further, since Eq (2.3) is a 1. order ODE.

Explicit methods, in contrast to implicit methods, approximates the next value y_{n+1} based on the present value y_n. For higher order explicit methods, there is used a weighted average of n increments, where n is the order of the method. The weights is determined either by a Taylor expansion to the desired order, solve the Taylor expansion for y', and then solve the occurring linear algebra system for the weights a, b, and c, or by replacing the derivatives with the corresponding difference equation, where either forward-, backward- or central differences may be used. Its harder to obtain a desired order this way, but the order of the method is easily checked once it's weights are determined. The Euler scheme is given as:

$$y'(t) = f(t, y(t)), \qquad y_{n+1} = y_n + hf(t_n, y_n). \tag{3.29}$$

where h is the time step, $t_n = t_0 + nh$, and $y_n \approx y(x_n)$, for n = 0, 1, 2, 3, ...

Heuns method, also called the improved or modified Euler method is given as:

$$\begin{aligned} y'(t) = f(t, y(t)), \qquad & \tilde{y}_{n+1} = y_n + hf(t_n, y_n) \\ y_{n+1} = y_n + \frac{h}{2}[f(t_n, y_n) + f(t_{n+1}, \tilde{y}_{n+1})], & \end{aligned} \tag{3.30}$$

Runge-Kutta 4, for short called RK4, is given as:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{3.31}$$

using

$$\begin{aligned} k_1 &= f(t_n, y_n), \\ k_2 &= f(t_n + \tfrac{h}{2}, y_n + \tfrac{h}{2}k_1), \\ k_3 &= f(t_n + \tfrac{h}{2}, y_n + \tfrac{h}{2}k_2), \\ k_4 &= f(t_n + h, y_n + hk_3). \end{aligned} \tag{3.32}$$

3.2.7 QR Decomposition

In linear algebra, a QR decomposition of a matrix is a decomposition of a matrix **A** into a product **A** = **QR** of an orthogonal matrix **Q** and an upper triangular matrix **R**. The creation of **Q** is done by for instance the Gram-Schmidt process

(other methods are "Householder transformations" or "Givens rotations"), which is a procedure to make every column in a matrix, with basis in the first column, normal to each other with vector norm (the same as magnitude/length of a vector) equal to 1. This creates an orthonormal basis. **R** is created afterwards as $\mathbf{R} = \mathbf{Q}^{\mathbf{T}}\mathbf{A}$.

The first vector is chosen to remain fixed in its direction, while all the other vectors are projected into a plane which is perpendicular to this first vector, and all the vectors made up to this point. This is easy to envision in 2D or 3D, but not so much for higher dimensions.

Then, all the vectors are normalized, which makes the matrix **Q** orthonormal. To be able to allow the voids to rotate during the deformation process, the current largest semi axis in each ellipsoid is set as the basis (1^{st} column in its eigenvector matrix **T**) for the Gram-Schmidt process.

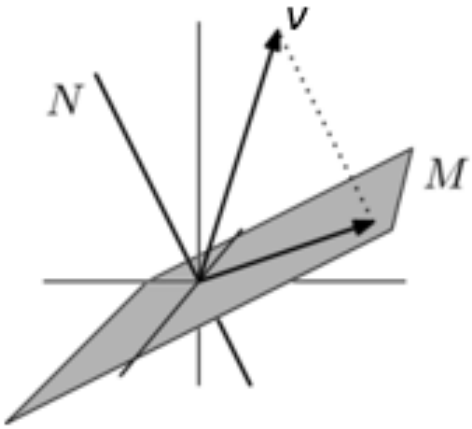


Figure 3.5: Projection of **v** into plane M, which is perpendicular to the vector **N** [34]

3.2.8 Constrained Optimization

This section is a summary of the relevant parts from the online source [35], in addition to [32]. In mathematical optimization, constrained optimization is the process of optimizing an objective function with respect to some variables in the presence of constraints on those variables.

Constraints can be either hard or soft constraints. Hard constraints set conditions for the variables that are required to be satisfied, while soft constraints have some variable values that are penalized in the objective function if, and based on the extent that, the conditions on the variables are not satisfied. A general constrained minimization problem may be written as follows:

$$\begin{array}{rlll} \min & f(\mathbf{x}) & \text{(the objective function)} & \\ \text{subject to} & g_i(\mathbf{x}) = c_i & \text{for } i = 1,\ldots,n & \text{(Equality constraints)} \\ & h_j(\mathbf{x}) \geq d_j & \text{for } j = 1,\ldots,m & \text{(Inequality constraints)} \end{array} \tag{3.33}$$

To solve problems like this, there exists several different strategies to choose from, depending on the properties of the problem. If the constrained problem has only equality constraints (the case for the problems in this thesis), the method of Lagrange multipliers may be used.

Lagrange multipliers

The method of Lagrange multipliers is a strategy for finding the local maxima and minima of a function subject to equality constraints. In mathematical analysis it is known that extremums must have the property that the set of partial derivatives of the objective function must be proportional to the corresponding partial derivatives of the constraint function. This can be expressed mathematically as:

$$\nabla_{x,y} f = -t\, \nabla_{x,y} g, \quad \text{for some t} \tag{3.34}$$

where f is the objective function, g the equality constraint function, and t the Lagrange multiplier. The Lagrange multiplier is a constant since the partial derivatives must be proportional in the maximum/minimum points.

3.3 Material Mechanics

The theory presented within this section (material mechanics) is taken from the references: [36], [37], [38], [39], and [15].

3.3.1 Stress Invariants

In this section, it is assumed isotropic materials. The stress state will be defined in terms of stress invariants.

Von Mises Stress

The first invariant is the Von Mises stress which is defined as:

$$\sigma_{VM} \equiv \sqrt{3J_2} \equiv \sqrt{\frac{3}{2}[(\sigma_I - \sigma_H)^2 + (\sigma_{II} - \sigma_H)^2 + (\sigma_{III} - \sigma_H)^2]} \tag{3.35}$$

where $\sigma_I \geq \sigma_{II} \geq \sigma_{III}$ are the principal stresses, $\sigma_H = \frac{1}{3}(\sigma_I + \sigma_{II} + \sigma_{III})$ the hydrostatic stress, and J_2 is the second principal invariant of the deviatoric stress tensor σ':

$$J_2 = \frac{1}{2}\sigma'_{ij}\sigma'_{ij} = \frac{1}{2}\left[(\sigma_I - \sigma_H)^2 + (\sigma_{II} - \sigma_H)^2 + (\sigma_{III} - \sigma_H)^2\right] \tag{3.36}$$

Stress Triaxiality

The second invariant employed to describe the stress state is the stress triaxiality σ^* which is defined by

$$\sigma^* \equiv \frac{I_\sigma}{3\sqrt{3J_2}} = \frac{\sigma_H}{\sigma_{VM}} \tag{3.37}$$

Lode Parameter

The third invariant adopted here is the Lode angle Θ_L which is defined as

$$\Theta_L = \frac{J_3}{2\sqrt{(J_2/3)^3}} = \frac{27J_3}{2\sigma_{VM}^3}, \quad -1 \leq \Theta_L \leq 1 \tag{3.38}$$

The lode variabel may also be defined in terms of the principal stresses as

$$\Theta_L = \frac{2\sigma_2 - \sigma_1 - \sigma_3}{\sigma_3 - \sigma_1}, \qquad \text{where} \quad -1 < \Theta_L < 1 \tag{3.39}$$

Owing to the normality rule, the strain field ε_{ij} is incompressible, i.e the volumetric strain is zero ($\varepsilon_V = 0$). This can be written on rate form:

$$\dot{\varepsilon}_1 + \dot{\varepsilon}_2 + \dot{\varepsilon}_3 = 0 \tag{3.40}$$

By invoking this theorem, the number of independent principal strain rates are reduced to two. These two unknowns can be expressed in terms of ε_1 and the lode parameter. The principal strain rate field may thus be expressed as

$$\dot{\epsilon}_2 = \frac{-2\Theta_L}{3+\Theta_L}\dot{\epsilon}_1 \qquad \dot{\epsilon}_3 = \frac{\Theta_L - 3}{3+\Theta_L}\dot{\epsilon}_1 \tag{3.41}$$

If proportional strain paths are considered, then the lode parameter Θ_L is constant, which implies that all $\dot{\varepsilon}_i$ are constant, which simplifies several calculations considerably.

Since the principal components of the strain rate field $\dot{\varepsilon}_k$ is parallel to the unit normal of the yield surface, it follows that the strain rate field can be characterized by the Lode parameter Θ_L as

$$\Theta_L = \frac{2\sigma_2 - \sigma_1 - \sigma_3}{\sigma_3 - \sigma_1} = \frac{2\dot{\varepsilon}_2 - \dot{\varepsilon}_1 - \dot{\varepsilon}_3}{\dot{\varepsilon}_3 - \dot{\varepsilon}_1} = -\frac{3\dot{\varepsilon}_2}{\dot{\varepsilon}_1 - \dot{\varepsilon}_3} \tag{3.42}$$

This is not valid in general, but the deformation of the matrix in the CA model implemented herein are limited to principal load directions parallel to the global axes. In such cases, Eq (3.42) holds.

3.3.2 Equivalent Plastic Strain

The definition of the equivalent strain most often used in the theory of plasticity is given as:

$$\varepsilon_{eq} = \sqrt{\frac{2}{3}\boldsymbol{\varepsilon}' : \boldsymbol{\varepsilon}'} = \sqrt{\frac{2}{3}\varepsilon'_{ij}\varepsilon'_{ij}}, \tag{3.43}$$

where

$$\boldsymbol{\varepsilon}' = \boldsymbol{\varepsilon} - \frac{1}{3}\mathrm{tr}(\boldsymbol{\varepsilon})\,\mathbf{1}, \tag{3.44}$$

or equivalently

$$\varepsilon'_{ij} = \varepsilon_{ij} - \frac{1}{3}\varepsilon_{kk}\delta_{ij}, \tag{3.45}$$

where $\text{tr}(\boldsymbol{\varepsilon}) \equiv \varepsilon_{kk}$, called the trace of $\boldsymbol{\varepsilon}$. For the case of a general extension deformation, where the elastic strains are neglected, this simplifies to

$$\varepsilon_{eq} = \sqrt{\frac{2}{3}(\varepsilon_1^2 + \varepsilon_2^2 + \varepsilon_3^2)}, \tag{3.46}$$

where it follows from the neglection of the elastic strains that

$$\varepsilon_i \equiv \varepsilon_i^p, \tag{3.47}$$

i.e the total strain is equal the plastic strain. The accumulated strain ε_{eq} is defined in rate form as

$$\dot{\varepsilon_{eq}} \equiv \sqrt{\frac{2}{3}\dot{\varepsilon}_{ij}^p \dot{\varepsilon}_{ij}^p} \tag{3.48}$$

where $\dot{\varepsilon}_{ij}$ are the components of the strain rate field.

3.3.3 Continuum Mechanics

Here the kinematics and kinetics of a continuum body loaded by quasi-static external forces are defined. That a body are a continuum body means that the material is modeled as a continuous mass rather than discrete particles. A body oc-

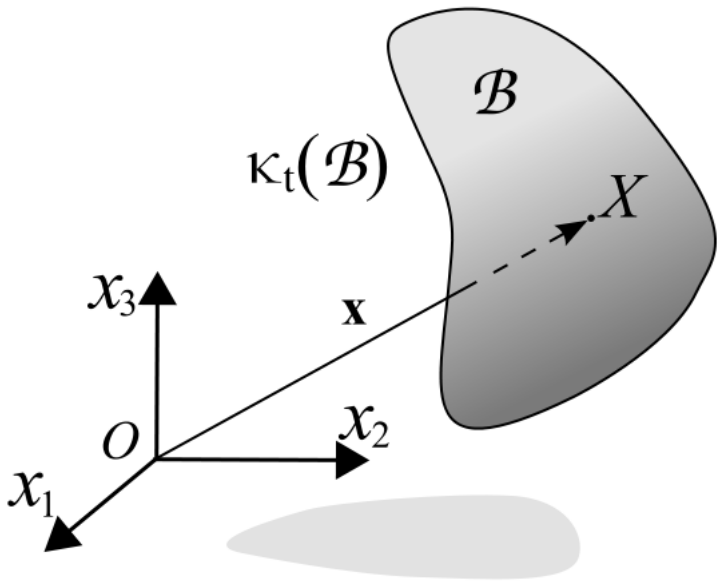


Figure 3.6: The configuration of a continuum body [40]

cupying a region (f.ex in Euclidean space) is called the continuum body $\mathcal{B}$ where the points inside this body are the material points. Every point in the body are occupied by a material point, and has a unique mapping (i.e no set of coordinates describes the same material point). In Figure 3.6, a continuum body in its reference configuration κ_t are shown.

The general case of deformation in the neighborhood of a material point P in a body that deforms from the reference configuration κ_0 at time t_0 to the present configuration κ_t at time t shall now be defined. The body is assumed to be undeformed in the reference configuration. Each particle i has its own coordinates

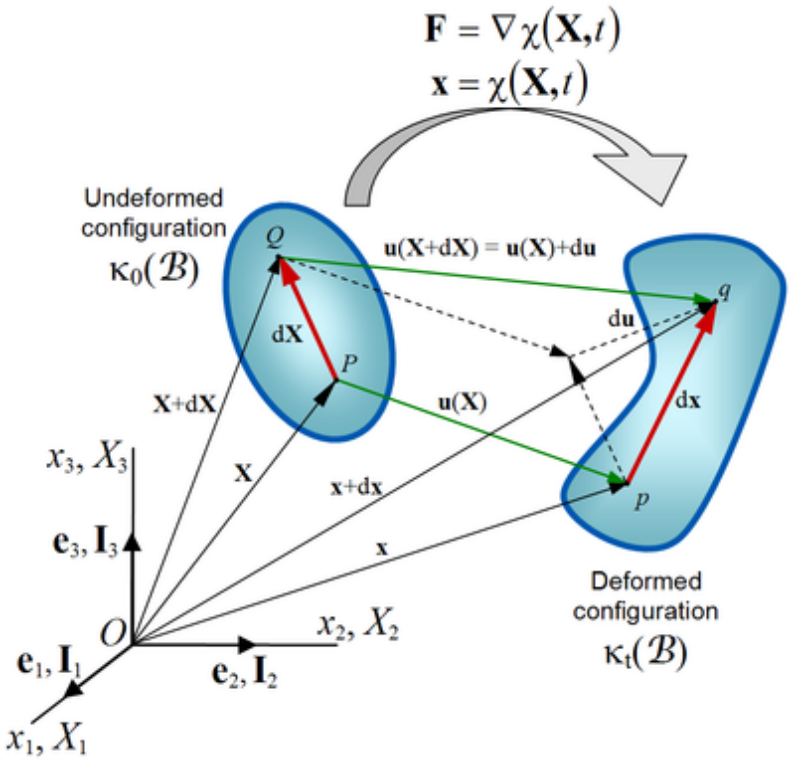


Figure 3.7: The deformation of a continuum body, [41]

X_i in the reference configuration κ_0, which may also be expressed as its *initial place vector* $\mathbf{X}$. The motion of the material point is given by its place vector $\mathbf{x}(\mathbf{X}, t)$, alternatively by the displacement vector $\mathbf{u}(\mathbf{X}, t)$:

$$\mathbf{u}(\mathbf{X}) = \mathbf{x}(\mathbf{X}, t) - \mathbf{X} \tag{3.49}$$

In Fig 3.7, the deformation is shown. The *deformation gradient* tensor is denoted $\mathbf{F}$ and is defined as:

$$\mathbf{F} \equiv \frac{\partial \mathbf{x}}{\partial \mathbf{X}} \equiv \text{Grad } \mathbf{x} = \nabla \mathbf{x} \tag{3.50}$$

The *displacement gradient* tensor $\mathbf{H}$ is defined by

$$\mathbf{H} \equiv \frac{\partial \mathbf{u}}{\partial \mathbf{X}} \quad \Leftrightarrow \quad H_{ij} \equiv \frac{\partial u_i}{\partial X_j} \tag{3.51}$$

The stretch ratio λ is related to the longitudinal strain, and is defined by:

$$\lambda = \frac{ds}{ds_0} \tag{3.52}$$

The relationship between the stretch ratio λ and the true strain ε_l is given by:

$$\varepsilon = \int_{l_0}^{l} \frac{dl}{l} = ln(\lambda) \tag{3.53}$$

Everything defined so far in this section, are strictly all that is necessary as long as infinitesimal deformations are assumed. The strain tensor for small deformations is easily defined in terms of either $\mathbf{F}$ or $\mathbf{H}$. But, to be able to describe

arbitrarily large deformations, several other interesting concepts must be introduced. To summarize the definitions above, the deformations can be described as:

$$x_i = x_i(X_i, t) = X_i + u_i(X_i, t) \tag{3.54}$$

In a homogeneous deformation of the body (the deformations are the same for all material points), the deformation may be described as:

$$\mathbf{x} = \mathbf{u}(\mathbf{X}) + \mathbf{F} \cdot \mathbf{X} \tag{3.55}$$

Where $\mathbf{u(X)}$ represents a translation, and $\mathbf{F} \cdot \mathbf{X}$ is $d\mathbf{x}$, see Fig 3.7. A *rigid-body motion* is an idealized situation where the deformation of the body is neglected during the displacement of the body. It is given by:

$$\mathbf{x} = \mathbf{u}(\mathbf{X}) + \mathbf{R} \cdot \mathbf{X} \tag{3.56}$$

Where $\mathbf{R}$ is the (orthogonal) rotation tensor. In the case of a rigid body motion, it is seen that $\mathbf{F} = \mathbf{R}$. A motion resulting in *homogeneous pure strain*, have that $\mathbf{F}$ is equal to a positive definite symmetric tensor $\mathbf{U}$.

$$\mathbf{x} = \mathbf{u}(\mathbf{X}) + \mathbf{U} \cdot \mathbf{X} \tag{3.57}$$

The principal values of $\mathbf{U}$ are $\lambda_i(t)$, and the eigenvectors of $\mathbf{U}$ are parallel to the principal deformation directions, represented by the unit vectors $\mathbf{a_i}$. A straight material line in the initial configuration can then be expressed as $d\mathbf{X} = ds_0\mathbf{a_i}$ If translations are set equal to zero for simplification, the following is obtained:

$$d\mathbf{x} = \mathbf{U} \cdot d\mathbf{X} = \mathbf{U} \cdot ds_0\mathbf{a_i} = \lambda_i ds_0\mathbf{a_i}$$

Where it is seen that $ds \equiv |d\mathbf{x}| = \lambda_i ds_0$, which means that λ_i are stretch ratios, called principal stretches of the deformation. In a coordinate system parallel to the principal deformation directions, the deformation may thus be given in terms of:

$$F_{ij} = U_{ij} = \lambda_i \delta_{ij}, \qquad \mathbf{x} = \mathbf{U} \cdot \mathbf{X} \equiv x_i = \lambda_i X_i \tag{3.58}$$

In this thesis, the coordinate system is chosen to have base vectors parallel to the principal directions of the loading. This means that the deformation is a motion of homogenous pure strain, with no translations or rotations, and it becomes much easier to define the deformation. A material point P at place $\mathbf{x}$ has the velocity $\mathbf{v}$(**x**,t) and is during the short time increment dt given the displacement $d\mathbf{u} = \mathbf{v} \cdot dt$. This displacement field leads to small deformations with a displacement gradient tensor $d\mathbf{H}$:

$$dH_{ij} = \frac{\partial v_i}{\partial x_j} dt \equiv v_{i,j} dt \tag{3.59}$$

Three new tensors are now defined: the *velocity gradient* tensor $\mathbf{L}$, the symmetric *rate of deformation* tensor $\mathbf{D}$ and the antisymmetric *rate of rotation* tensor $\mathbf{W}$, all defined at the time t:

$$\mathbf{L} = \text{grad}\mathbf{v} \equiv \frac{\partial \mathbf{v}}{\partial \mathbf{x}} \quad \Leftrightarrow \quad L_{ij} = v_{i,j} \tag{3.60}$$

$$\mathbf{D} = \frac{1}{2}(\mathbf{L} + \mathbf{L}^\mathbf{T}) \quad \Leftrightarrow \quad D_{ij} = \frac{1}{2}(v_{i,j} + v_{j,i}) \tag{3.61}$$

$$\mathbf{W} = \frac{1}{2}(\mathbf{L} - \mathbf{L}^\mathbf{T}) \quad \Leftrightarrow \quad W_{ij} = \frac{1}{2}(v_{i,j} - v_{j,i}) \tag{3.62}$$

From these definitions, and if small deformations is assumed, it follows that:

$$d\mathbf{H} = \mathbf{L}dt \tag{3.63}$$
$$d\mathbf{E} = \mathbf{D}dt \tag{3.64}$$
$$d\tilde{\mathbf{R}} = \mathbf{W}dt \tag{3.65}$$
$$\dot{\mathbf{E}} = \mathbf{D} \tag{3.66}$$
$$\dot{\tilde{\mathbf{R}}} = \mathbf{W} \tag{3.67}$$

In contrast to small deformations, where $H_{ij} << 1 \Leftrightarrow norm(\mathbf{H}) << 1$, it is more convenient to use the deformation gradient tensor $\mathbf{F}$ instead of the displacement gradient $\mathbf{H}$ when large deformations are considered. For large deformations, the equation $d\mathbf{E} = \mathbf{D}dt$ does not hold anymore. From the definitions of $\mathbf{F}$ in (3.50) and $\mathbf{L}$ in (3.60), the following relationship is derived:

$$\dot{\mathbf{F}} = \mathbf{L}\mathbf{F} \quad \Leftrightarrow \quad \mathbf{L} = \dot{\mathbf{F}}\mathbf{F}^{-1} \tag{3.68}$$

In large deformation, a state of pure strain has that $\mathbf{F}(\mathbf{X},t) = \mathbf{U}(\mathbf{X},t)$ and $\mathbf{R}(\mathbf{X},t) = \mathbf{1}$. In this thesis, a motion that is called *general extension* is considered, with the principal loading directions defined in the same direction as the global coordinate system. The motion is described through the relations:

$$x_i = \lambda_i(t)X_i \qquad \text{for } i = 1,2,3 \tag{3.69}$$

The deformation gradient $\mathbf{F}$ is now an diagonal matrix, and it is also equal to the stretch tensor $\mathbf{U}$:

$$\mathbf{F} \equiv \mathbf{U} = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} \tag{3.70}$$

The parameters λ_i are the principal stretches of the deformation. The material points velocity $\mathbf{v}$ is given as:

$$v_i = \frac{\partial x_i}{\partial t} = \dot{\lambda}_i X_i = \frac{\dot{\lambda}_i}{\lambda_i} x_i \tag{3.71}$$

Which gives the velocity gradient $\mathbf{L}$ as:

$$\mathbf{L} = \begin{bmatrix} \frac{\dot{\lambda}_1}{\lambda_1} & 0 & 0 \\ 0 & \frac{\dot{\lambda}_2}{\lambda_2} & 0 \\ 0 & 0 & \frac{\dot{\lambda}_i}{\lambda_i} \end{bmatrix} \tag{3.72}$$

In this case, it is seen that $\mathbf{D} = \mathbf{L}$, and that $\mathbf{W} = \mathbf{0}$, since $\mathbf{L}$ is symmetric. In this thesis, the deformation of the matrix material is defined through the global velocity gradient $\mathbf{L}$. The deformation of the voids are defined through a local velocity gradient $\mathbf{L}^*$, where the heuristic approach of defining global semi-axes for the ellipsoids were a necessity. This will be explained further in Chapter 4.

3.3.4 Elastic-Plastic Materials

Metallic materials exhibit linear elastic behavior for small stresses, while at a certain level (denoted the yield stress), the behaviour becomes elasto-plastic. The strain is divided into a recoverable part (the elastic strain) and an irrecoverable part (the plastic strain). The simplest possible material model is perfect plasticity, where the yield stress will be constant during further plastic deformation. If the elastic response is neglected, the material model exhibits what is called a *rigid - perfectly plastic* response.

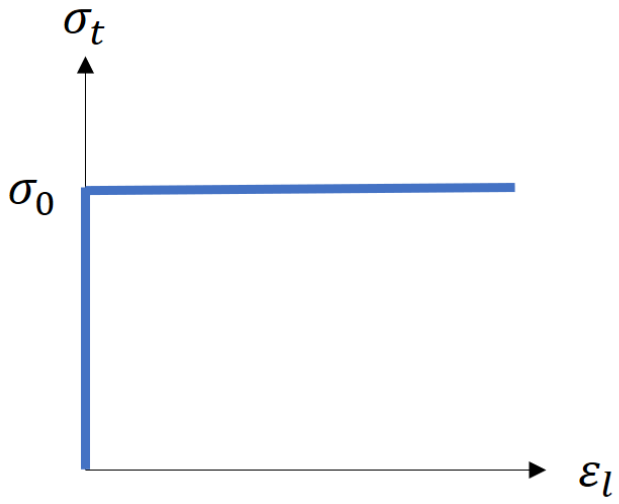


Figure 3.8: Stress-strain curve for a perfectly plastic material

The growth equations developed by Rice&Tracey assumes a rigid-perfectly plastic material response, where an extension was made to include rigid-linear hardening material response, where the only difference was a constant parameter. Therefore, the material assumed in this thesis are modeled as a rigid-perfectly plastic material, where the elastic response where neglected

$$\varepsilon^e \equiv 0 \quad \Rightarrow \quad \varepsilon = \varepsilon^p \tag{3.73}$$

When results obtained with the model are compared with analyses in Abaqus, the material model used in Abaqus had a very high value for Young's modulus in an attempt to neglect the elastic response.

3.4 Programming

3.4.1 Object Oriented Programming - OOP

Only a brief introduction is presented; it would be an advantage to have some experience from procedure-oriented programming. See [42], [43] and [44] for in depth information about all the subsections presented in this section (and a lot that hasn't been presented).

In Object-Oriented Programming (OOP for short), the term object means a collection of data with a set of functions for accessing and manipulating those data. OOP allow decomposition of a problem into a number of *objects* and then builds data (*attributes*) and functions (*methods*) around these objects. There are several reasons to use objects instead of global variables and functions, where the most beneficial is that problems may become unmanageable very fast for more complex things. A lot of "book-keeping" is taken care of behind the curtains with OOP. By "book-keeping", it is meant that it becomes easier to get, set and store calculated values. When for instance doing the same calculation several times, where the only difference is the arguments used for a function call, OOP really shines. Instead of explicitly changing the arguments, they are stored as attributes in different objects, and OOP just calls the same method for each object. This makes the code *much* more compact and easier to write/read. The basic ideas of OOP may be summarized through these three concepts:

Polymorphism: You can use the same operations on objects of different classes!

Encapsulation: You hide unimportant details of how the objects work from the outside world

Inheritance: You can create specialized classes of objects from general ones.

In this thesis, polymorphism and inheritance is of less importance, since the only class used is a class called *Void*, that is used to represent the voids in the matrix material. The basic idea of an object (a collection of data and functions) is nonetheless crucial to be able to manage a large number of voids that must interact with each other, and itself be the target of a lot of manipulation.

In OOP, the first step is to create a *Class* for representing data. A class is a blueprint of an object that contains variables for storing data and functions to perform operations on these data. An object is an instance of a class. An example may be the class Dog and the instance Fido. Every instance of a class gets its own unique set of data, and every method for the class works on these unique attributes. Objects provide a layer of abstraction, which is used to separate internal and external code.

The differences between class and objects are shown in Fig 3.9. The object Fido has its own unique attributes, and a set of methods that works on all instances of the class Dog. These methods may or may not take any of the instance's attributes as input (For example a method *eat* should probably take Fido's weight as input parameter, since large dogs usually eat more).

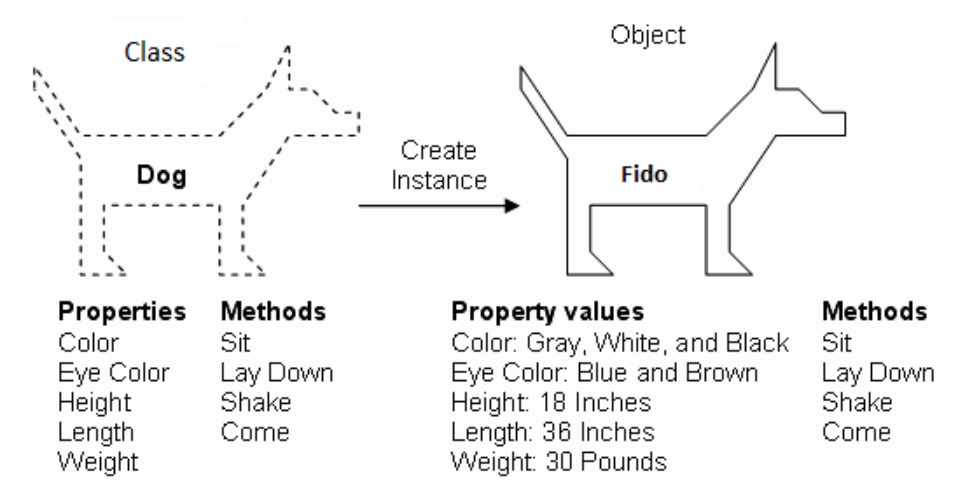


Figure 3.9: Class and Object [45]

An Object Oriented Model

When trying to solve a problem, write a description of what the desired program should do. Underline all nouns, verbs and adjectives, and go through the following checklist:

Nouns, looking for potential classes

Verbs, looking for potential methods

Adjectives, looking for potential attributes

This list was used to develop the basic idea of how the program should be written, based on the desired model behaviour. A void does according to this mindset classify as a class, while all the things the voids may do, like grow, rotate, translate, etc. classifies as methods.

If a very strict object oriented model is desired, every function should ideally be written as a method, but some potential methods are still written as functions in the program. OOP was used to get a much more intuitive program written in relatively few lines in Python, but it was not seen as a necessity to hide every function as a corresponding method.

3.4.2 Convex Hull

In mathematics, the convex hull of a set of points X in the Euclidean plane or Euclidean space is the smallest convex set that contains X. For instance, when X is a bounded subset of the plane, the convex hull may be visualized as the shape enclosed by a rubber band stretched around X.

The algorithmic problem of finding the convex hull of a finite sets of points in a Euclidean space is one of the fundamentals problems of computational geometry. In this thesis the concept of a convex hull was used to find the volume of the voids that grew outside the box's dimensions. This is a part of the program the user can decide to include or not. In the work with the model, it was ultimately decided that the voids that grew outside the box's dimension statistically should

be equalized by the voids that should have grown into the box from the outside. Therefore, this part of the program was not included in the calculations of the results presented herein.

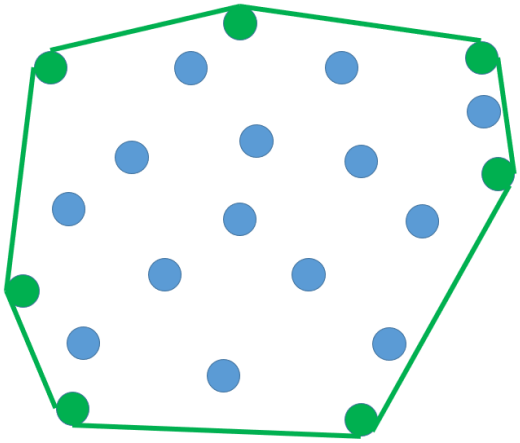
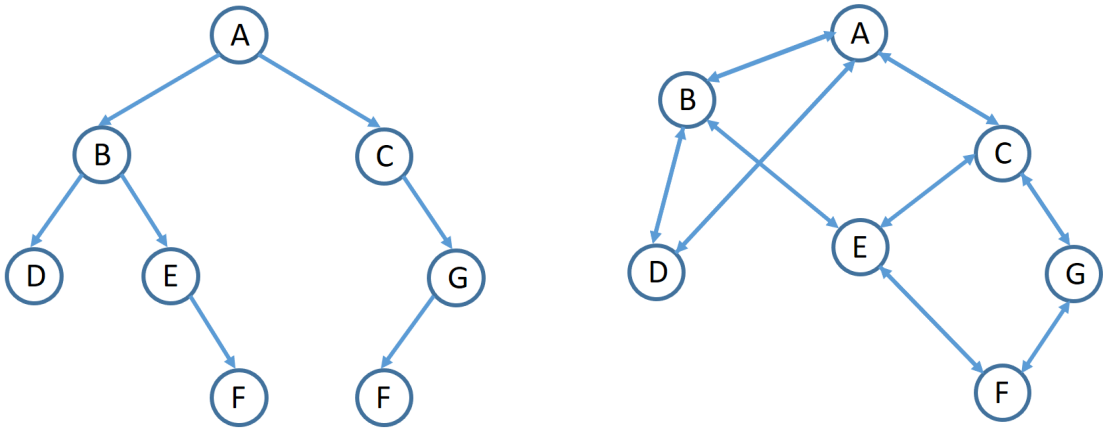


Figure 3.10: A convex hull in 2D

3.4.3 Data Structures

There exists many types of data structures. A data structure is a particular way of organizing data in a computer so it can be used efficiently. Linear data structures are sequences, such as array, lists, tuples, matrices etc. Trees have one root-node, and subtrees of children with a parent node. Graphs consists of a set of vertices (nodes) and a set of pairs of these vertices called edges.

Linear data structures is used to store values in a sequence, and is mostly used to perform calculations. The concept of a sequence is assumed known to the reader, so the rest of this subsection shall focus on graphs.



(a) The tree structure

(b) The graph structure

Figure 3.11: The tree and graph data structures

Graphs are a powerful mental (and mathematical) model of a structure in general. A graph $G = (V, E)$ consists of a set of nodes V and edges between them E. If

the edges have directions only one way, the graph is *directed*, otherwise *undirected*. Graphs can represent all kinds of structures/systems, and their expressibility can be increased by adding extra data such as weights or distances. A tree is just a special type of graph; they are connected and have no cycles.

A highly useful mental model for graph algorithms is 'traversal', i.e discovering and later visiting all the nodes in a graph. The two most well-known basic traversal strategies are depth-first search and breadth-first search. In classic graph traversal, each node in the graph should be visited, and exactly once. This is implemented by adding the nodes that have to be visited to some sort of collection, and adding the ones that are connected to the visited nodes through edges to a list of nodes that need to be visited. When a node becomes visited, it is removed from the 'must visit' list and added to a collection of discovered nodes.

The check neighbors algorithm implemented in this thesis is inspired by these traversal algorithms, but with some twists. In classical graph traversal, the goal is to reach each node in the graph, exactly once. In this thesis, the goal is to check every neighbor pair, exactly once. Every void is controlled against each of its neighbors, unless this exact pair of voids have been controlled before. Thats why a *predecessor list* is needed to keep track of which pairs that have been checked.

3.4.4 Asymptotic Notation

This is the last part of the theory chapter, and it may seem a bit unnecessary. But, it is quite useful to analyze the running complexity of the model, in order to be able to localize the bottle-necks of the program. By knowing how the program behaves, it is much easier to look for computational improvements. Coupled with the fact that this topic is unknown to most without a background in programming, an attempted brief introduction will conclude this chapter.

In computer science, in the analysis of algorithms, asymptotic analysis is a method of considering the performance of algorithms when applied to very big input datasets. The distinction between "constant factors" (related to more general things such as hardware or language performance) and the growth of the running time as problem size increases, is of crucial importance in the study of algorithms.

The core idea of asymptotic notion is to represent the resource that's being analyzed (usually time or memory) as a function, with the input size as the parameter, usually denoted n.

The asymptotic notion consists of O (Big Oh), Ω and Θ. The expression $O(g)$ for some function g represent a set of functions, and a function $f(n)$ is in this set if it satisfies the following condition: There exists a natural numbern_0 and a positive constant c such that:

$$f(n) \leq cg(n) \quad \text{for all} \quad n > n_0 \tag{3.74}$$

The constant c may be tweaked (f.ex by running on machines with different speed). Where O forms an asymptotic upper bound, Ω forms an asymptotic lower bound. Θ is the intersection of both, $\Theta(g) = O(g) \cap \Omega(g)$, it is both an upper and lower bound at the same time.

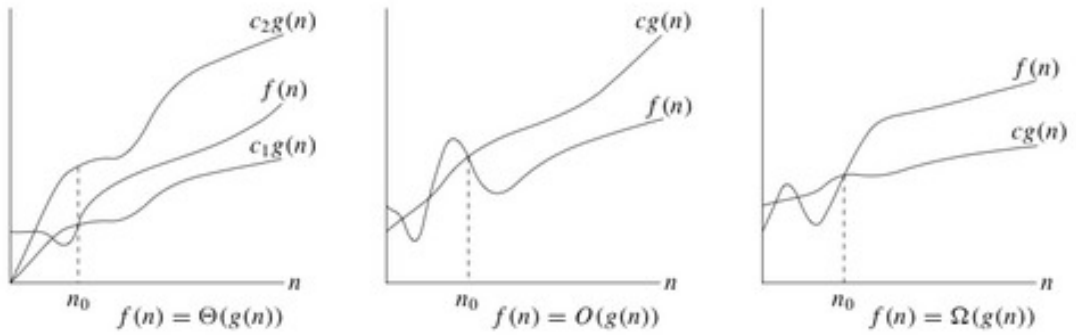


Figure 3.12: The three asymptotic notions [46]

When working with asymptotic analysis of algorithms, there is one important rule:

> Drop all multiplicative and additive constants, as well as other "small" parts of the function. All that matters is the largest order of n.

Algorithm design can be seen as a way of achieving low asymptotic running time by designing efficient algorithms. There has been paid quite a lot of attention throughout the development of this program on algorithmic design.

4 The Cellular Automaton Model

The overall framework of the Cellular Automaton model, "CA model" for short, will first be explained in order to give the reader a conceptual idea along with the mathematical definition of features related to the model. Following this, flowcharts and condensed pseudo formulations of the model will be presented.

In the model, the material is thought to exist of two phases, a matrix material and voids (which are just empty space). The two phases are decomposed, but managed separately. The matrix material is subject to a deformation field, defined by the velocity gradient **L**. The translation of the voids are based on the velocity gradient field for the entire box. Description of the two phases follows:

Matrix Phase
The box (the part of the material that are under consideration) is thought of as plastically incompressible. In reality, it is the matrix material that should be considered incompressible, but the void growth are considered uncoupled with the box. Based on this assumption the velocity gradient field is defined in terms of the lode parameter and the principle axis (which is defined as the x-axis in the model). By employing the incompressible theorem, the relations between the three different terms of the velocity gradient field is obtained.

Void Phase
The assumptions made in the Rice&Tracey growth equation is that the void under consideration is placed in an infinite matrix. No void interactions are assumed. A simplification of this model has been made, by allowing the matrix material's volume to decrease as the voids grow. See Chapter 6 for further discussion. As the voids grow, the void volume fraction keeps increasing, which means that the matrix volume is decreasing, since the box's volume is kept constant. The volume fraction of the voids V_f is between one and zero, i.e: $V_f \in [0,1]$, and the volume fraction of the matrix then follows as $V_m = 1 - V_f$.

4.1 Main Parts of the Program

In this section, the main parts of the program will be discussed in more detail. The program consists of some essential functions and methods that constitutes the basis of the program, and a lot of smaller functions that do the bookkeeping work. The four most important of these functions will be presented. After the reader is given a thorough understanding of the essential parts of the program, the flowcharts and pseudo code formulation is presented. With a understanding of the essential parts and the flow of the program, the reader is hopefully given a good basis for following the discussion of the validity, but also the flaws of the model in Chapter 5 and 6.

4.1.1 Simplifications / Limitations of the Model

The nucleation of voids are as mentioned neglected in this thesis. The material is modeled with an initial void volume fraction, spread among a desired number of voids. This is a conservative assumption, since all the particles that otherwise should have nucleated sometime into the deformation process are all assumed nucleated as the deformation starts. Depending on the user input, the voids are given an orientation and size, among several options. Since nucleation is neglected, it means that the total void number will only decrease, when coalescence occurs. For coalescence to occur, the coalescence criterion must be fulfilled. In this thesis, coalescence is defined to occur when two voids intersect each other. The two voids will then be replaced by a new ellipsoid, defined as the minimum volume enclosing ellipsoid of the two that collided, or in an implemented extension of the model, as a scaled version of this ellipsoid.

4.1.2 1^{st} Function: Grow, Rotate and Translate the Voids

This is perhaps the single most important function/method in the program. The procedure implemented here is credited my supervisors. The first step is to determine the local velocity gradient for each void, denoted $\mathbf{L}^*$. It is unique for each void, and may differ significantly from the global velocity gradient which is a result of the deformation of the box under consideration. The Rice&Tracey growth equation is used to calculate the rate of change of the global semi-axes of the voids, in the global coordinate system directions, see Fig 4.1. The term local will denote the coordinate system of each individual void (ellipsoids), as shown in Fig 2.6a. The local velocity gradient $\mathbf{L}^*$ is determined directly from the definition of the velocity gradient, as defined in Eq (3.60):

$$\mathbf{L}^* = \frac{\partial \mathbf{v}^*}{\partial \mathbf{x}^*} \tag{4.1}$$

The void growth model developed by Rice&Tracey was originally developed for voids with the principal axes aligned with the principal loading directions. An heuristic procedure was therefore followed in order to estimate the void growth rates in the case of an ellipsoidal void. The semi-axes of the voids are replaced by the three intersections of the void with the principal loading directions, again referring to Fig 4.1. This equivalent void is then assumed to follow the Rice&Tracey

growth equation. $\mathbf{L}^*$ is therefore developed as:

$$\mathbf{L}^* = \begin{bmatrix} \frac{\dot{R_1}^{glob}}{R_1^{glob}} & 0 & 0 \\ 0 & \frac{\dot{R_2}^{glob}}{R_2^{glob}} & 0 \\ 0 & 0 & \frac{\dot{R_3}^{glob}}{R_3^{glob}} \end{bmatrix} \tag{4.2}$$

where $\dot{R_i^{glob}}$ are found by solving the Rice&Tracey growth equation for this time step with either euler, heun or RK4 (see subsection 3.2.6), and R_i^{glob} are found as:

$$R_i^{glob} = \sqrt{\frac{1}{A_{ii}}} \tag{4.3}$$

The local velocity gradient field $\mathbf{L}^*$ is used to determine how the void will ro-

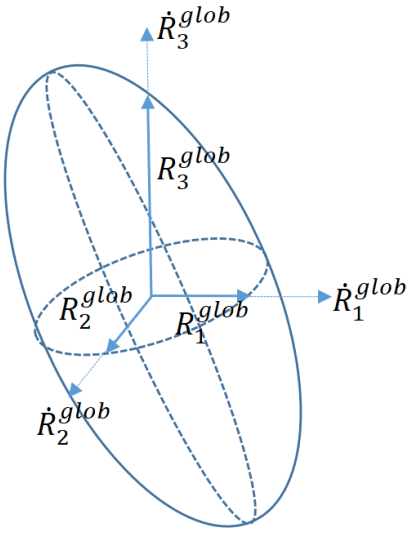
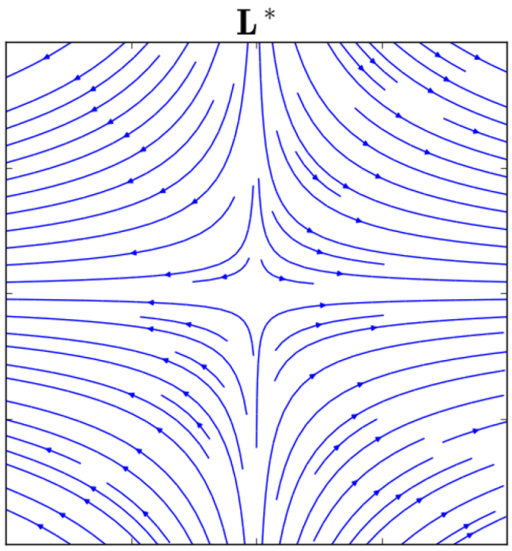


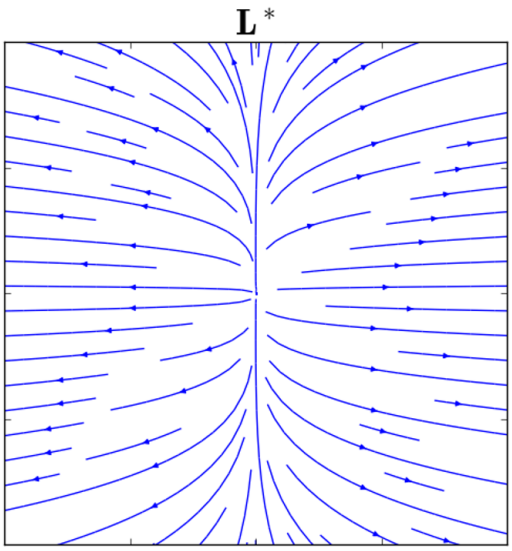
Figure 4.1: The global axes and "axes rates" of a void

tate. $\mathbf{L}^*$ acts upon the local (principal) semi-axes of the void during the timestep dt, which results in three no longer orthogonal semi-axes. The three semi-axes must then undergo a QR decomposition, to become orthogonal, as an ellipsoid's principal axes *must* be orthogonal to each other. The largest semi-axis in the ellipsoid is set to determine the rotation of the ellipsoid. This is done by setting it as the first column in its local coordinate basis, before the QR decomposition is performed.

In Fig 4.2, it is shown a graphical illustration in the form of a vector field, of how $\mathbf{L}^*$ looks like in 2D for low and high values of the triaxiality, respectively. It is seen how the void's semi-axes will decrease towards smaller y-values for small values of the triaxiality σ^* (the same is true for the z-values), so the volume of the void will actually decrease towards zero after some deformation. For higher values of σ^*, the y-values (and z-values) will increase, which means that the void will grow in every direction. The Rice&Tracey growth equation was first and foremost developed for these scenarios (higher triaxiality values).



(a) The vector field created by $\mathbf{L}^*$ for low triaxialities

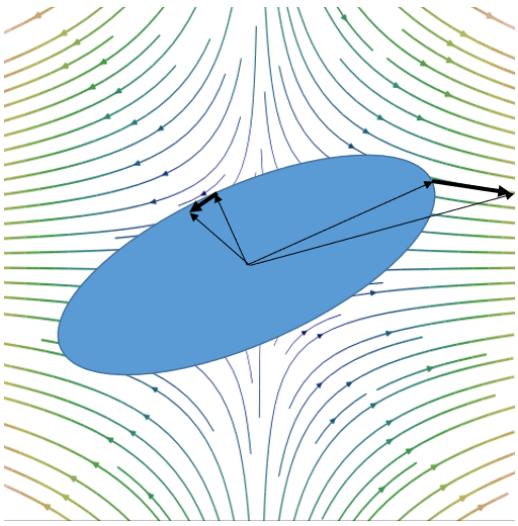


(b) The vector field created by $\mathbf{L}^*$ for larger triaxialities

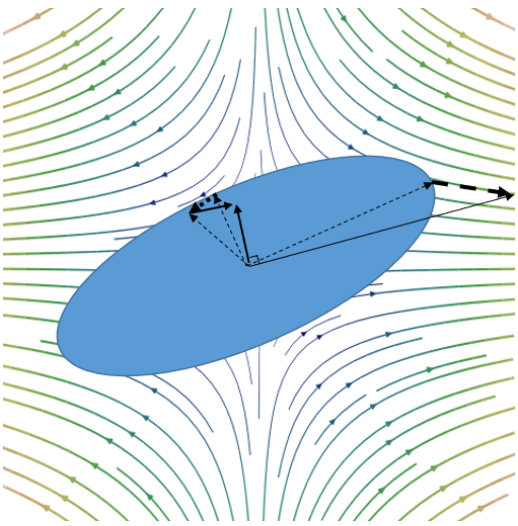
Figure 4.2: Local velocity tensor $\mathbf{L}^*$

The void's semi-axes deforms under the influence of $\mathbf{L}^*$ as shown in Fig 4.3a. This is the first step of the deformation process. The principal axes of the void has grown and rotated during the time increment, but the deformations have been greatly exaggerated to make a point. This step of the deformation process of the semi-axes are calculated as:

$$\mathbf{g_i}^{n+1} = \mathbf{g_i}^{n} + \Delta t \mathbf{L}^* \mathbf{g_i}^{n} \qquad \text{for } i = 1, 2, 3 \tag{4.4}$$



(a) Voids deforms under influence of $\mathbf{L}^*$



(b) The QR decomposition normalizes the set of vectors

Figure 4.3: Deformation of the void

By looking at the deformation of the void in Fig 4.3a, it is evident that the semi-axes of the void is no longer perpendicular to each other. Therefore, the

QR decomposition (which uses the Gram-Schmidt process) is used to orthonormalize the set of vectors, where the first vector's direction is leaved untouched, see Fig 4.3b. Since $\mathbf{L}^*$ tries to deform the semi-axes in opposing directions, this may introduce an error in the rotation of the voids, so very small time steps are recommended.

This resulting new basis is then set as the ellipsoids eigenvectors $\mathbf{T}$, and the norms of the vectors $\mathbf{g_i}$ are used to get the eigenvalues. With the eigenvectors and eigenvalues known, the $\mathbf{A}$ matrix which defines the ellipsoids in generalized form is calculated as:

$$\mathbf{A} = \mathbf{TDT^{-1}} \tag{4.5}$$

The voids were also translated, which means its position was updated to C^{new}. This is based on a linear interpolation of the box's deformation, given through the stretch ratios λ_i at the given time increment:

$$C^{new} = \begin{bmatrix} C_1^{orig}\lambda_1(t) \\ C_2^{orig}\lambda_2(t) \\ C_3^{orig}\lambda_3(t) \end{bmatrix} \tag{4.6}$$

where t is the current time in the deformation, and C^{orig} is the initial position of the voids, as given at $t = 0$. The stretch ratios are functions of time only.

Depending on whether Θ_L is constant or not, the values of $\mathbf{L}^*$ may either be constant during the deformation, or change. If Θ_L is set to vary, both Θ_L and $\mathbf{L}^*$ will nonetheless be a piecewise constant function, as the values within each time increment are held constant. Anyway, this means that λ_i should satisfy the equation:

$$\frac{\dot{\lambda_i}}{\lambda_i} = c_i \qquad \text{where } \; i = 1, 2, 3 \tag{4.7}$$

where c_i are constants. This is a 1^{st} order ordinary differential equation, and the solution is

$$\lambda_i = e^{c_i t} \qquad \text{where } \; i = 1, 2, 3 \tag{4.8}$$

The set of constants c_i are given in terms of the lode parameter Θ_L and the stretch ratio λ_i, since the deformation process were defined in terms of Θ_L and $\dot{\varepsilon}_1$.

4.2 Coalescence of Voids

The algorithm described in this section were briefly explained in "Algorithms for Ellipsoids" by Stephen B. Pope [47]. To determine when the ellipsoids are in contact, one of the two ellipsoids in question (E_1 and E_2) is transformed to a unit sphere in origin by the transformation:

$$\bar{\mathbf{x}} = \mathbf{R_1}(\mathbf{x} - \mathbf{c_1}), \tag{4.9}$$

and this transformed ellipsoid is denoted $E_1^{'}$. E_2 then undergoes the same transformation, and the new ellipsoid is denoted $E_2^{'}$. The point $x_2^{'}$ in $E_2^{'}$ which is closest to the origin is determined by the Lagrange multiplier method. If this point is closer to the origin than 1 (remember that $E_1^{'}$ is a unit sphere) the two ellipsoids intersect.

To determine the transformation matrix between the two coordinate systems, a transformation matrix $\mathbf{Q}$ with the following properties must be developed:

$$\mathbf{x} = \mathbf{Q}\bar{\mathbf{x}}, \tag{4.10}$$

so that the three unit vectors $\bar{x}_1$, $\bar{x}_2$ and $\bar{x}_3$ turns into the ellipsoid's principal axes by undergoing this transformation. The transformation matrix $\mathbf{Q}$ is therefore given by the length and orientation of the ellipsoids principal axes, which are the inverse of the root of the eigenvalues and the eigenvectors, respectively. The transformation matrix between the two coordinate systems is thus:

$$\mathbf{Q} = \left[\mathbf{v_1}\frac{1}{\sqrt{\lambda_1}}, \quad \mathbf{v_2}\frac{1}{\sqrt{\lambda_2}}, \quad \mathbf{v_3}\frac{1}{\sqrt{\lambda_3}}\right] \tag{4.11}$$

where λ_i are the three eigenvalues, and $\mathbf{v_i}$ the three eigenvectors. The matrix may therefore also be expressed as:

$$Q = \begin{bmatrix} \frac{v_1^{[1]}}{\sqrt{\lambda_1}} & \frac{v_2^{[1]}}{\sqrt{\lambda_2}} & \frac{v_3^{[1]}}{\sqrt{\lambda_3}} \\ \frac{v_1^{[2]}}{\sqrt{\lambda_1}} & \frac{v_2^{[2]}}{\sqrt{\lambda_2}} & \frac{v_3^{[2]}}{\sqrt{\lambda_3}} \\ \frac{v_1^{[3]}}{\sqrt{\lambda_1}} & \frac{v_2^{[3]}}{\sqrt{\lambda_2}} & \frac{v_3^{[3]}}{\sqrt{\lambda_3}} \end{bmatrix} \tag{4.12}$$

where $v_i^{[j]}$ means vector i's j^{th} component. The first ellipsoid is transformed according to:

$$\begin{aligned} \mathbf{x_1^T A_1 x} &= 1 \\ (\mathbf{Q}\bar{\mathbf{x}})^{\mathbf{T}}\mathbf{A_1}(\mathbf{Q}\bar{\mathbf{x}}) &= 1 \\ (\bar{\mathbf{x}})^{\mathbf{T}}(\mathbf{Q^T A_1 Q})\bar{\mathbf{x}} &= 1 \\ (\bar{\mathbf{x}})^{\mathbf{T}}(\bar{\mathbf{A}}_{\mathbf{1}})\bar{\mathbf{x}} &= 1 \end{aligned} \tag{4.13}$$

which shows that $\bar{\mathbf{A}}_{\mathbf{1}} = \mathbf{Q^T A_1 Q}$. Ellipsoid 2 undergoes the same transformation, i.e $\bar{\mathbf{A}}_{\mathbf{2}} = \mathbf{Q^T A_2 Q}$. The transformation is a linear one, so the transformation of the distance between the ellipsoids centers follows as:

$$\begin{aligned} d_0 &= \mathbf{C_2} - \mathbf{C_1} \\ d_1 &= \mathbf{Q^{-1}} \cdot d_0 \end{aligned} \tag{4.14}$$

where d_0 is the initial distance between the voids, and d_1 the distance between the centers of the transformed voids. The transformed ellipsoid 2, $\bar{E}_2$, expressed by $\bar{\mathbf{A}}_2$, has eigenvalues and eigenvectors denoted $\bar{\lambda}_2$ and $\bar{\mathbf{v}}_2$, respectively.

To find the shortest distance from origin to some point in the ellipsoid $\bar{E}_2$, the easiest approach is to change coordinate systems, so $\bar{E}_2$ can be expressed by its standard equation (see Eq (3.10)). Thus, $\bar{E}_2$ is transformed again into its own local coordinate system, and denoted $\bar{E}_2^{new}$, through the transformation:

$$\bar{\mathbf{A}}_{\mathbf{2}}^{\mathbf{new}} = \bar{\mathbf{v}}_{\mathbf{2}}^{\mathbf{-1}}\bar{\mathbf{A}}_{\mathbf{2}}\bar{\mathbf{v}}_{\mathbf{2}} \tag{4.15}$$

The transformed distance between the centers of the ellipsoids are transformed again, i.e the vector from origin to the center of ellipsoid $\bar{E}_2$, since the coordinate system has been changed:

$$d_2 = \bar{\mathbf{v}}_{\mathbf{2}}^{\mathbf{-1}} \cdot d_1 \tag{4.16}$$

The problem is now reduced to finding the minimum distance from d_2 to the surface of $\bar{E}_2^{new}$, and if this distance is smaller than or equal to unity, then the ellipsoids are intersecting. To find the shortest distance from a point to an ellipsoid expressed in its standard equation(again, Eq (3.10)), the following equation should be minimized:

$$(x-x_0)^2+(y-y_0)^2+(z-z_0)^2 \tag{4.17}$$

where (x_0, y_0, z_0) is the point d_2, and (x, y, z) are points on the surface of the ellipsoid, subject to the constraint of the standard equation of an ellipsoid (Eq (3.10)).

As described in 3.2.8, this is a problem that calls for finding a Lagrange multiplier. It is solved by setting

$$[2(x-x_0),\ 2(y-y_0),\ 2(z-z_0)] \tag{4.18}$$

proportional to

$$[2x/a,\ 2y/b,\ 2z/c] \tag{4.19}$$

The proportionality factor (the Lagrange multiplier) is here denoted t. This gives the following equations

$$t \ = \ a\frac{x-x_0}{x} \ = \ b\frac{y-y_0}{y} \ = \ c\frac{z-z_0}{z} \tag{4.20}$$

which leads to

$$x=\frac{a}{a-t}\,x_0, \quad y=\frac{b}{b-t}\,y_0, \quad z=\frac{c}{c-t}\,z_0 \tag{4.21}$$

Substituting these into the ellipsoid equation gives

$$\frac{a}{(a-t)^2}\,x_0^2+\frac{b}{(b-t)^2}\,y_0^2+\frac{c}{(c-t)^2}\,z_0^2 \ = \ 1, \tag{4.22}$$

from which it is obtained that

$$\begin{aligned}(a-t)^2(b-t)^2\cdot(c-t)^2 \ &= a(b-t)^2(c-t)^2\cdot x_0^2+b(c-t)^2(a-t)^2\cdot y_0^2 \\ &\quad + c(a-t)^2(b-t)^2\cdot z_0^2\end{aligned} \tag{4.23}$$

This is a sixth order polynomial equation in the multiplier t. Python were used to solve this, where the seven coefficients were worked out with symbolic calculations. A general polynomial may be written on the form:

$$c_1\cdot x^n+c_2\cdot x^{n-1}+\ldots+c_n=0, \tag{4.24}$$

where the seven coefficients were found to have the following values

$$\begin{aligned}
c_1 &= 1 \\
c_2 &= -2(a+b+c) \\
c_3 &= a^2+b^2+c^2+4(bc+ca+ab)-(ax_0^2+by_0^2+cz_0^2) \\
c_4 &= 2(-a^2(b+c)-b^2(c+a)-c^2(a+b) \\
&\quad -4abc+a(b+c)x_0^2+b(c+a)y_0^2+c(a+b)z_0^2) \\
c_5 &= b^2c^2+c^2a^2+a^2b^2+4abc(a+b+c)-a(b^2+c^2)x_0^2-b(c^2+a^2)y_0^2 \\
&\quad -c(a^2+b^2)z_0^2-4abc(x_0^2+y_0^2+z_0^2) \\
c_6 &= 2abc\cdot((b+c)x_0^2+(c+a)y_0^2+(a+b)z_0^2-bc-ca-ab) \\
c_7 &= abc(abc-bcx_0^2-cay_0^2-abz_0^2)
\end{aligned} \tag{4.25}$$

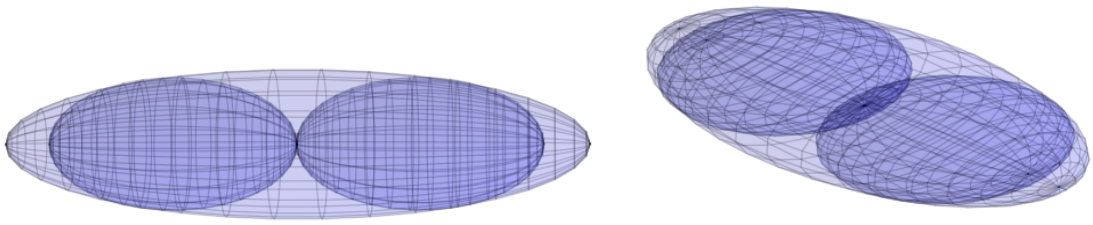
Only the real roots gives possible solutions, so for every real root of t the corresponding (x, y, z) coordinates are found from Eq (4.21). The distance d (Eq (4.17)) is calculated for these real solutions, and the smallest value among them is chosen. In this case, there is exactly two real roots; the maximum and minimum distance from the point to the ellipsoid's surface. As mentioned before, if this distance d is smaller than unity, $d \leq 1$, the ellipsoids intersects.

4.3 Minimum Volume Enclosing Ellipsoid

In mathematical optimization, the ellipsoid method is an iterative method for minimizing convex functions with inequality constraints. The minimum volume enclosing ellipsoid will for short be denoted MVEE, or MVE ellipsoid from here on. The ellipsoid method generates a sequence of ellipsoids whose volume uniformly decreases at every step, thus enclosing a minimizer of a convex function. Leonid Khachyan's [48] algorithm is implemented in this thesis, with help from [49]. See also [50] and [51] for a full explanation of the algorithm. The following optimization problem is solved

$$\begin{aligned} &\text{minimize} && log(\det(\mathbf{A})) \\ &\text{subject to} && (\mathbf{P_i} - \mathbf{c})' * \mathbf{A} * (\mathbf{P_i} - \mathbf{c}) \leq 1 \quad \text{for all points } \mathbf{P_i} \end{aligned} \tag{4.26}$$

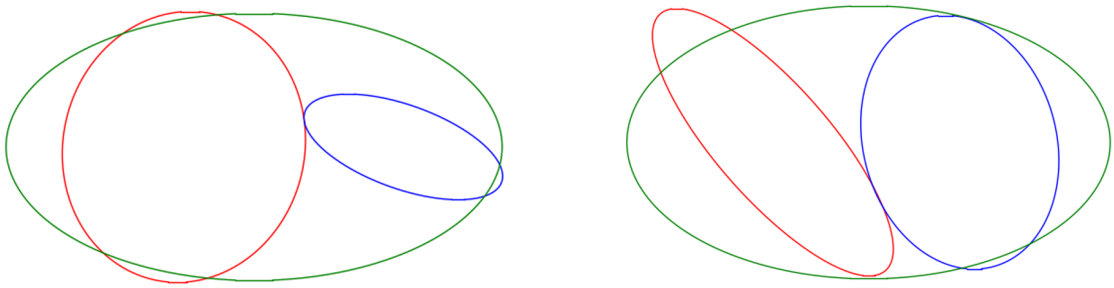
in variables $\mathbf{A}$ and $\mathbf{c}$, for a set of data points stored in $\mathbf{P}$, where $\mathbf{P_i}$ is the i^{th} point in $\mathbf{P}$. $\mathbf{A}$ describes as before the ellipsoid equation in its generalized form, and $\mathbf{c}$ is the ellipsoid's center. The final solution differs from the optimal value by the pre-specified amount of *tolerance*. This is the reason the new ellipsoid is not completely covering all the input points. The pseudo-code for this algorithm are shown in 4.3. Figures that combine the two algorithms MVEE and IntersectingVoids, are shown in Fig 4.4 and Fig 4.5.



(a) Two coalesced voids, from the front (b) Two coalesced voids, from an angle

Figure 4.4: Coalescence shown in 3D

As shown in Fig 4.4, when two aligned (and in this case identical) voids intersects, the resulting MVE ellipsoid is clearly very accurate. When two ellipsoids with very different size and/or orientations, the resulting MVE ellipsoid does not obtain quite as accurate results, see Fig 4.5 and Fig 4.6. Here it is shown that by varying the error tolerance, a visible effect on the resulting MVE-ellipsoid returned from the algorithm are obtained. Some parts of the two intersecting ellipsoids will be outside the resulting MVE-ellipsoid, because of the fact that the algorithm is truncated before reaching its optimal value, because of the TOL parameter.



(a) Tight error tolerance between random oriented voids

(b) A more relaxed error tolerance

Figure 4.5: Coalescence shown in 2D upon contact

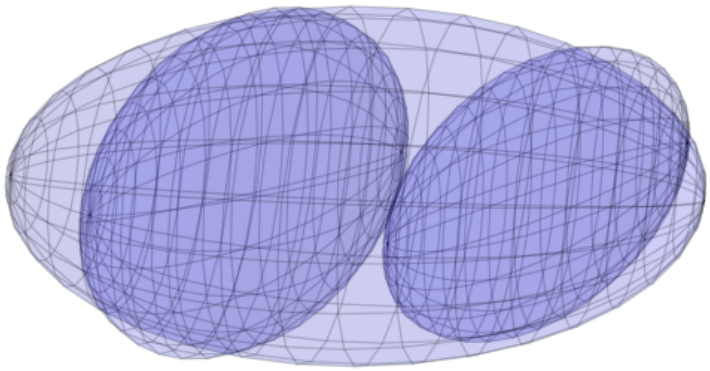


Figure 4.6: A more relaxed tolerance shown in 3D

When two voids are determined to be intersecting, MVEE are called with arguments being sample points from the two ellipsoids surfaces. The new ellipsoid that is created does have larger volume than the sum of the intersecting ellipsoids, so matrix is removed from the model for each coalescence. This is an non-physical phenomenon, since the matrix in reality is incompressible during plastic deformations. On the other hand, the voids would probably coalesce quite a while before they intersect. These two effects are at the moment assumed to cancel each other out, at least to some extent. The fact that the MVE-ellipsoid "eats" matrix, is quite suitably called the "Pac-Man effect".

In Fig 4.7, the "Pac-Man effect" is shown for the same scenario shown in Fig 4.4, i.e for two identical initially spherical voids, that experiences coalescence. The red graph is the "Pac-Man effect", and it makes a jump for each coalescence, and are constant in between. The "Pac-Man effect" denotes how much of the volume that is the result of "eaten" matrix of the total void volume. The total void volume is the sum of void growth and the "Pac-Man effect". These results will be further discussed in Section 5.

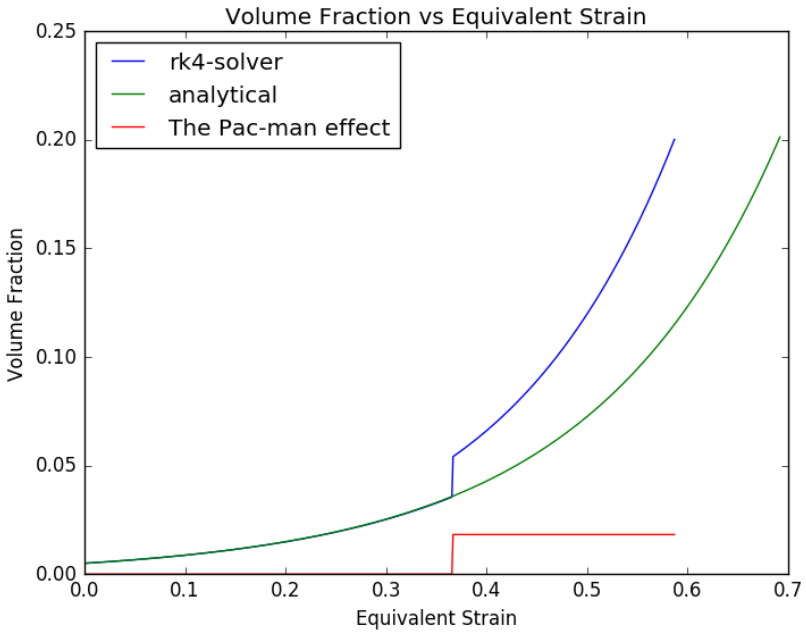


Figure 4.7: The Pac-Man Effect

4.4 Collision with Box

The last algorithm that will be shown. This is used for the periodic distribution of voids during the parameterstudy in Chapter 6, and are also used in order to subtract the void volume of the parts of the voids that is outside the box's dimensions, if the user so specifies.

In an attempt to model the behavior of the microvoids even more accurately, voids were allowed the option to be positioned outside the box's dimensions. This way, voids may grow into the box from the outside, and out of the box from the inside. Only the volume of the parts of the voids that are inside the voids are then added to the void volume. The disadvantage is that more voids, combined with the fact that only part of the void's volume should be added to the total void volume, of course will result in longer computation time. As mentioned in subsection 3.4.2, it was decided that the voids that grew outside the box's dimension statistically should be equalized by the voids that should have grown into the box from the outside. The benefits of this procedure was therefore rather uncertain, and while it is an option for the user, it was not used to produce any of the results presented herein.

To determine if a void is intersecting the box, the general equation of the ellipsoid (Eq (3.12)) is expanded, which results in

$$\begin{aligned} x(a_{00}x + a_{01}y + a_{02}z) + y(a_{01}x + a_{11}y + a_{12}z) \\ +z(a_{02}x + a_{12}y + a_{22}z) - 1 = 0 \end{aligned} \tag{4.27}$$

This expression is a 2^{nd} order polynomial in three variables. It is solved for the different scenarios where the ellipsoid may collide with the planes determined by a constant x, y or z value, i.e the expression is solved for x, y and z. To determine if the planes intersects the ellipsoid, the discriminants need to be established (see

Algorithm 1: MVE Ellipsoid: Pseudocode

Data: A matrix **P** storing points, and a tolerance for error TOL
Result: **A** and **c**
Initialization: N is the number of points, and d the dimension of the points
Establish **Q**: $\mathbf{Q} = [\mathbf{P}; ones(1, N)]$
Establish **u**: $\mathbf{u} = (1/N) * \text{ones}(N, 1)$
while *error > tolerance* **do**
  Establish **X**: $\mathbf{X} = \mathbf{Q}\ \text{diag}(\mathbf{u})\ \mathbf{Q}^T$
  Establish **M**: $\mathbf{M} = \text{diag}(\mathbf{Q}^T\ \text{inv}(\mathbf{X})\ \mathbf{Q})$
  Store the maximum value in **M** as *max*, and the location/index as j
  Calculate step size: *step-size* =
    (max - d -1)/((d+1)·(max-1))
  Calculate **new_u**: $\mathbf{new\_u} = (1 - \textit{step-size}) \cdot \mathbf{u}$
  Increase the j^{th} element of **new_u** : **new_u**(j) = **new_u**(j) + step-size
  Store an error-measure: *err = norm(***new_u** - **u***)*
  Increment count: *count = count + 1* Replace u: $\mathbf{u} = \mathbf{new\_u}$
end
Compute **U**: $\mathbf{U} = \text{diag}(\mathbf{u})$
Compute **A**: $\mathbf{A} = (1/d) * \text{inv}(\mathbf{P} * \mathbf{U} * \mathbf{P}^T - (\mathbf{P} * \mathbf{u}) * (\mathbf{P} * \mathbf{u})^T)$
Compute center: $\mathbf{c} = \mathbf{P} * \mathbf{u}$

subsection 3.2.3). The discriminants are given as

$$\begin{aligned} discr_x = &- a_{00}a_{11}y^2 - 2a_{00}a_{12}yz - a_{00}a_{22}z^2 + a_{00} \\ &+ a_{01}^2y^2 + 2a_{01}a_{02}yz + a_{02}^2z^2 \end{aligned} \tag{4.28}$$

$$\begin{aligned} discr_y = &- a_{00}a_{11}x^2 - 2a_{02}a_{11}xz - a_{11}a_{22}z^2 + a_{11} \\ &+ a_{01}^2x^2 + 2a_{01}a_{12}xz + a_{12}^2z^2 \end{aligned} \tag{4.29}$$

$$\begin{aligned} discr_z = &- a_{00}a_{22}x^2 - 2a_{01}a_{22}xy - a_{11}a_{22}y^2 + a_{22} \\ &+ a_{02}^2x^2 + 2a_{02}a_{12}xy + a_{12}^2y^2 \end{aligned} \tag{4.30}$$

The distance from the ellipsoids centroid to the plane is then substituted for the relevant variable. Now, the discriminants are functions of one variable only. When for instance controlling intersection with the y-plane, the discriminant for the equation solved with respect to x is found, and then the value for y is substituted. The resulting function is then only in z, i.e $f = f(z)$. The 2^{nd} derivative of $f(z)$ is negative for all values of z, which follows from the fact that the quadratic form is given by a definite positive matrix **A**. So if the maximum value of $f(z)$, found as $f(z_{crit})$, is ≥ 0, where z_{crit} is obtained from

$$\frac{\partial f(z_{crit})}{\partial z} = 0, \tag{4.31}$$

then the ellipsoid intersects the plane, since the discriminant is positive ($\Delta \geq 0$). Thus, a real solution exists. Otherwise, they are separated.

4.5 Varying Lode Θ_L or Triaxiality σ^*

Finally, it should also be mentioned that the triaxiality and lode must by no means be constant throughout the deformation. They may vary depending on the user input, through the function called *Make_linear_function()*. In Fig 4.8, it is shown how for instance the stress triaxiality may vary. The development of σ^* may for instance be taken from a FE-analysis, and the development may be given to the CA model. The deformation process is therefore not limited to proportional deformation, as long as the principal deformation directions aligns with the global coordinate system in the CA model. Proportional deformation means that Θ_L is constant, and it is also the most usual scenario in experiments. The results obtained in this thesis have all been for constant values of Θ_L and σ^*, but the choice to let these parameters vary have been implemented.

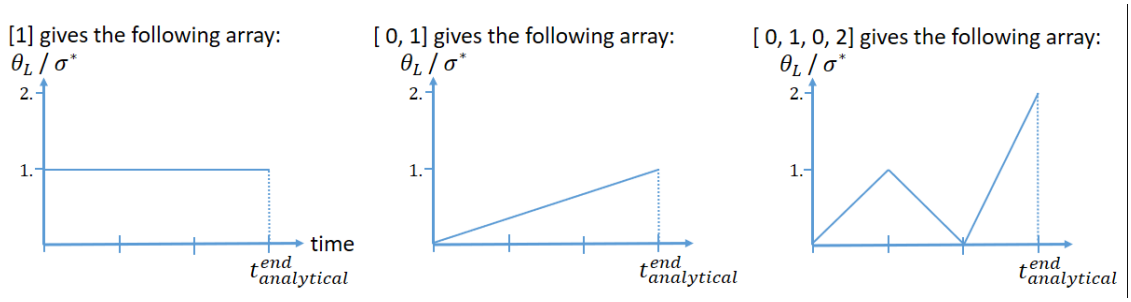


Figure 4.8: How Θ_L or σ^* may vary, defined by user

4.6 Flowchart and Pseudocode Formulation

This section will present flow charts and the corresponding pseudocode for the two main parts of the program. The reader is advised to maybe focus more on the flowcharts in Fig 4.9 and Fig 4.10 than the pseudocode shown in 4.11 and 4.12. When trying to formulate a program in a computer language, there is always a lot more unexpected commandos that need to be executed compared to the oral description of the problem, so to get an understanding of what happens, and in which order it happens, the flowcharts will give the necessary information.

The instantiation part (green flowchart, Fig 4.9) are executed in the beginning of the blue flowchart; see the green box in Fig 4.10.

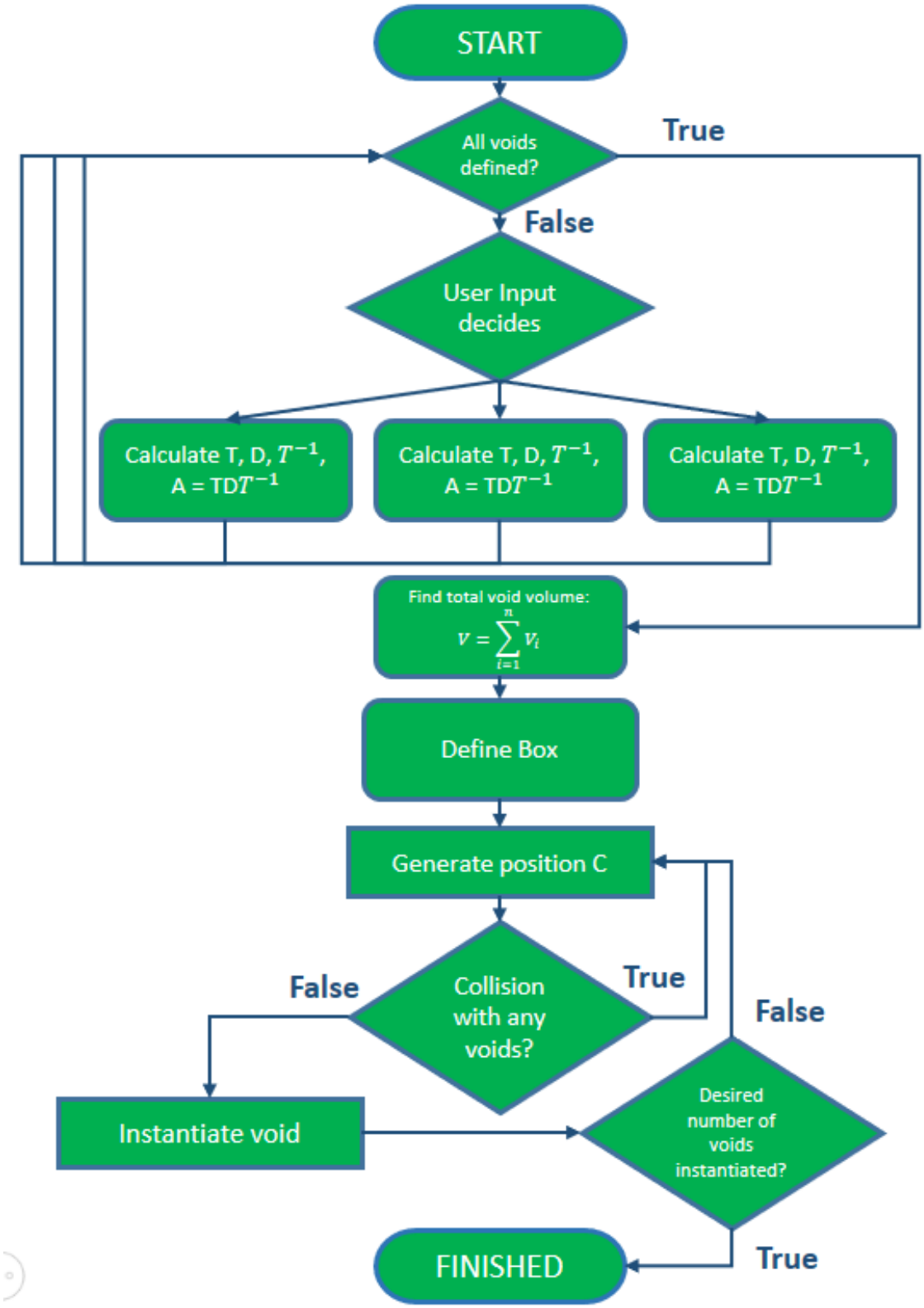


Figure 4.9: Flowchart that gives an overview of the instantiation part of the program

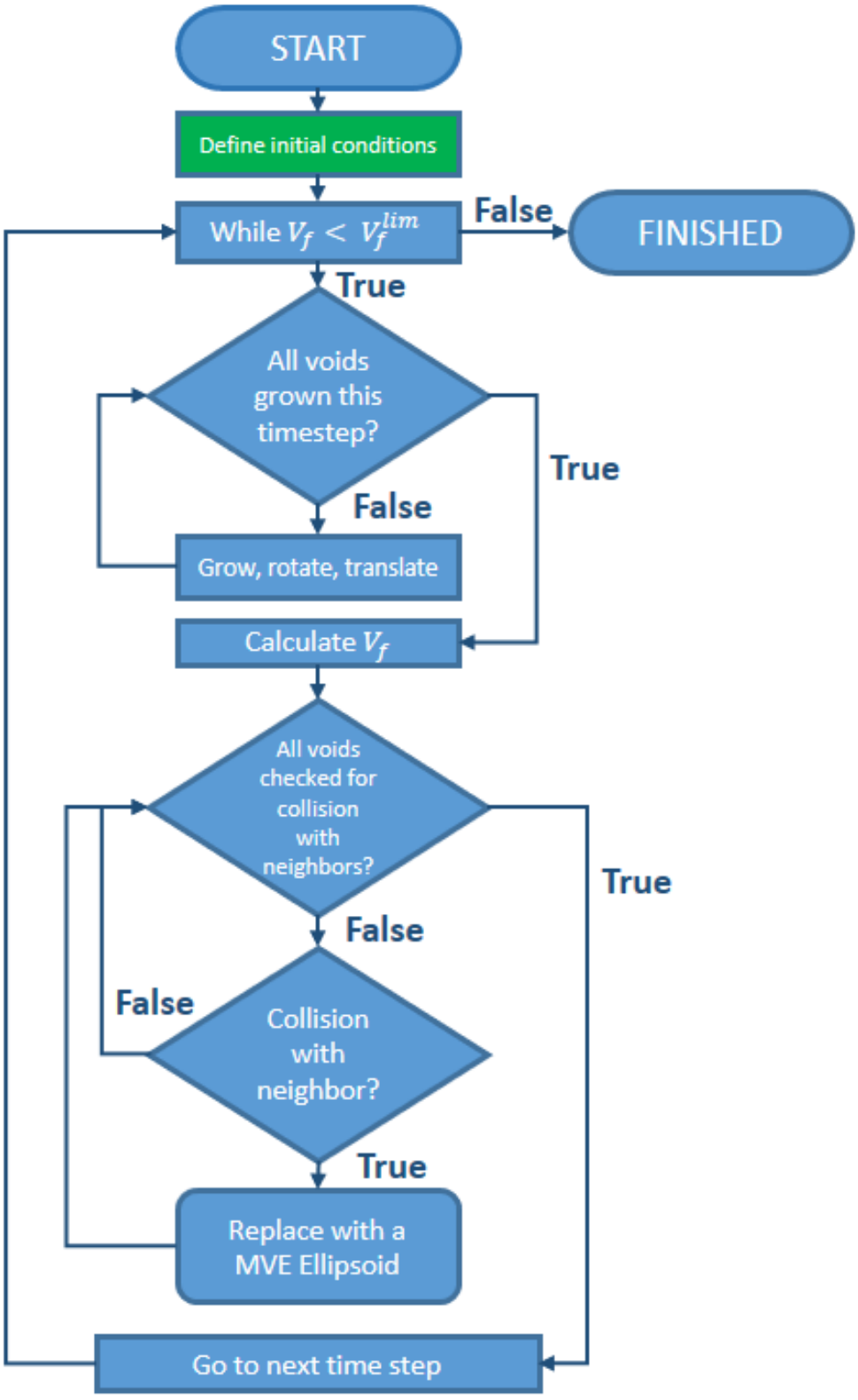


Figure 4.10: Flowchart that gives an overview of the main part of the program

Figure 4.11: Pseudocode for the instantiation part

Algorithm 2: Pseudocode - Instantiation part

```
for each void desired do
    if orientation == something then
        Generate desired orientation
    else
        Generate another type of void...
    end
end
Calculate void volume
Define box
for each void desired do
    Generate position
    for each void instantiated do
        if collision with new void then
            generate another position
        else
            instantiate void
        end
    end
end
```

Figure 4.12: Pseudocode for the main part of the program

Algorithm 3: Pseudocode - Main part

while $V_f \leq V_F^{lim}$ **do**
  Find neighbors every k^{th} increment
  $\sigma^* = \sigma^*_{vec}[t]$
  $\Theta_L = \Theta_L^{vec}[t]$
  if $sigma^* \leq 1.0$ **then**
    $D = \alpha e^{\frac{3\sigma^*}{2}}$
  else
    $D = \alpha \sigma^{\frac{1}{4}} e^{\frac{3\sigma^*}{2}}$
  end
  The constants c_i in $\lambda(t) = e^{c_i t}$ are determined from:
  $c_1 = user\text{-}specified\ value$
  $c_2 = (-\Theta_L * c_1 * 2)/(3 + \Theta_L)$
  $c_3 = c_1 * (\Theta_L - 3)/(\Theta_L + 3)$
  $L = \begin{bmatrix} c_1 & 0 & 0 \\ 0 & c_2 & 0 \\ 0 & 0 & c_3 \end{bmatrix}$
  Calculate $\dot{\varepsilon}_{eq}$
  for *each void* **do**
    Grow, rotate and translate
    Add all the voids volumes together, to get V_f
    if *user decides* **then**
      subtract the volumes of the voids thats outside the box
    end
  end
  for *each void* **do**
    for *each neighbor* **do**
      if *(void,neighbor) pair has been checked already* **then**
        skip it
      else
        check collision between the ellipsoids
      end
      if *collision* **then**
        Sample points from both voids
        Create a MinimumVolumeEllipsoid around these points
        Delete the two intersecting voids
        Instantiate the replacement MVEE void
        for *each void* **do**
          Delete these two voids from the neighbor list
        end
        Find all the neighbors for the new MVEE void
        Add the (void,neighbor) pair to a list of checked pairs
      end
    end
  end
  Increase t: $t+ = dt$
end

5 Verification

In the development of the model, there were possibly two major sources of error: Flaws in the mathematical foundation and mistakes in the software implementation of the mathematical foundation. In chapter 4, the mathematical foundation was described in detail for the most important algorithms. In this section, it will be assumed that these procedures were calculated correctly, and the focus will therefore be on the implementation in Python, and how the results compares to known solutions where these are available.

5.1 Preliminary Controls

In order to avoid bugs (flaws/errors) in a program, a continuous quality control of the code were performed. Examples are:

1. Symbolic operations were performed and controlled with Sympy, which is a Python module for symbolic math operations.

2. Comparison with the closed form solution of Rice&Tracey

3. Graphical plots were used all the time to see if the voids behaved as expected. When a large amount of random generated voids are shown in plots during the deformation process, it is quite easy to see if something does not behave as expected, where rotation and translation of the voids are obvious examples, but also coalescence and the MVE ellipsoid. A lot of errors where discovered this way; by simply considering the results graphically to see if the behavior made sense.

4. When the implemented algorithms didn't work as expected, further study of the problem was undertaken, which most often resulted in a change of the whole algorithmic idea. Especially the collision between voids was a problem of this type; where the procedure changed several times during the work with the thesis.

5.2 Comparison to Closed-Form Solution

To control the numerical solution, it is compared to the analytical solution of the growth equation. As mentioned before, as long as the stress triaxiality and lode parameter is held constant during the deformation, there exists a closed form solution. The Rice&Tracey void growth equation (Eq (2.3)) have a closed form solution when Θ_L and σ^* are held constant during the deformations:

$$R_i = R_0(A + \frac{3+\Theta_L}{2\sqrt{3+\Theta_L^2}}B \qquad \text{for} \quad i = 1,2,3 \tag{5.1}$$

where

$$\begin{aligned} A &= \exp(\frac{2\sqrt{3+\Theta_L^2}}{3+\Theta_L}D\varepsilon_1 \\ B &= \frac{1+E}{D}(A-1) \end{aligned} \tag{5.2}$$

To compare the results, an initial spherical void with principal axes in the same directions as the global axes was computed. The stress triaxiality and lode parameter was held constant, and the three semi-axes' magnitude were calculated. The differences between the analytical and the numerical solution was extremely small even for relatively large time steps.

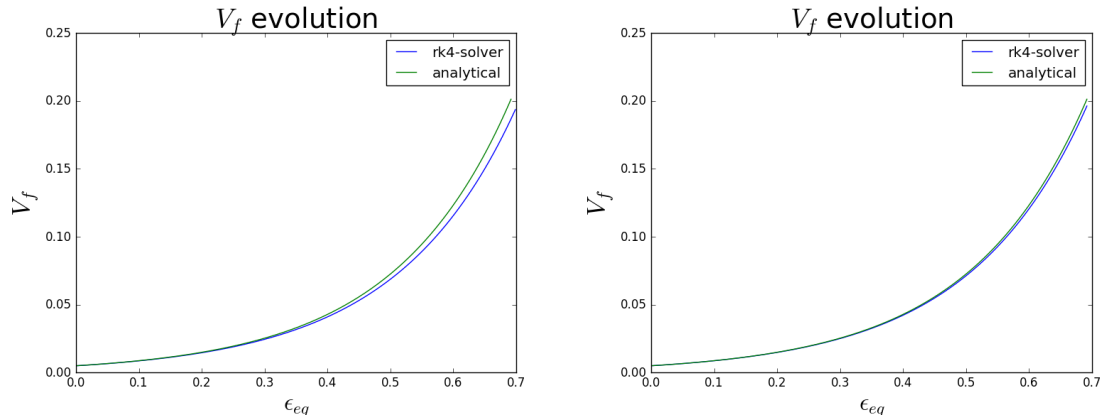
The figure Fig 5.1 shows how the volume developed both analytically and numerically during the deformation, according to Runge-Kutta 4 with the two different timesteps 100 and 300. The lode parameter used was $\Theta_L = 1$, and the triaxiality was $\sigma^* = 1$. This value of σ^* is chosen because the Rice&Tracey model is first and foremost developed for stress triaxialities at this value and higher.

Most of the simulations run in this thesis will use this value for σ^*. With this value, the voids will grow in all principal directions. Higher values will result in very fast void growth, while low values will result in a situation where one or two (depending on Θ_L) principal axes will start to shrink, and this is a bit outside the assumptions made in the development of the Rice&Tracey growth equation. $\sigma^* = 1$ is therefore a well suited value for the validation and verification of the model.

As shown in Fig 5.1 (which shows the volume of the void vs equivalent strain), the numerical and analytical solutions are very close to each other, even for as few as 100 time steps! The situations when $\Theta_L = 0$ and $\Theta = -1$ behaves exactly the same, so they will not be shown here. A plot of how the voids semi-axes develop is also shown, see Fig 5.2, where the length of the principal axes are plotted against the equivalent strain. Since the volume of an ellipsoid is

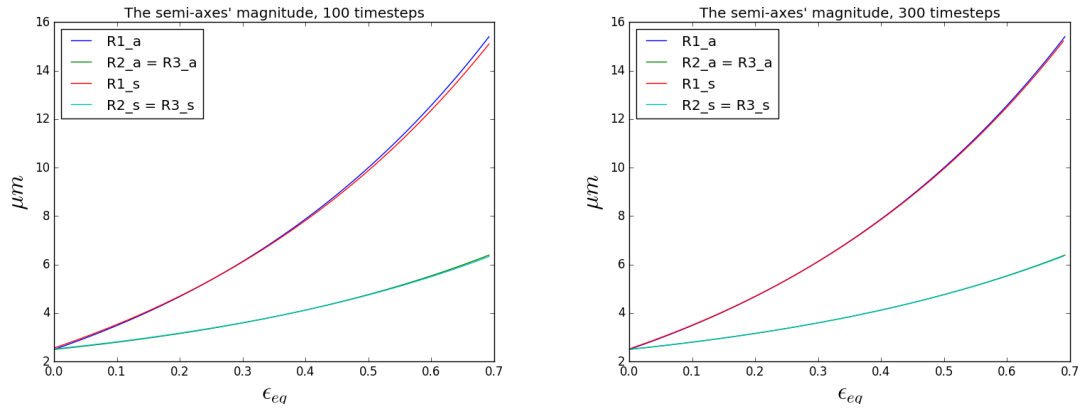
$$V_f = \frac{4\pi R_1 R_2 R_3}{3} \tag{5.3}$$

the error shown in the calculated V_f will in a worst case scenario be in the 3^{rd} power (the semi-axes calculated in the numerical solution is not always either all smaller or larger than the analytical solution, therefore only *worst case scenario*). Still, the differences between numerical and analytical solution is insignificant, and by increasing the timesteps up to f.ex 500, it is almost impossible to tell them



(a) 100 timesteps (b) 300 timesteps

Figure 5.1: Evolution of the void volume fraction

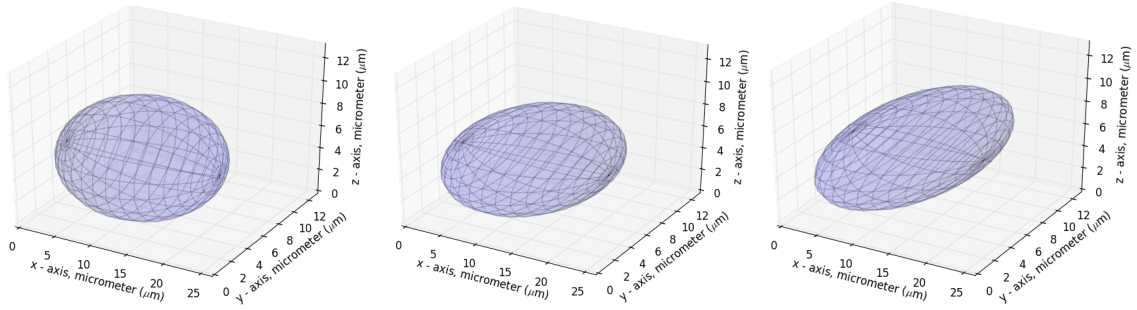


(a) 100 timesteps (b) 300 timesteps

Figure 5.2: Evolution of the void's semi-axes, $\Theta_L = 1$

apart. It is concluded that the numerical solution gives very accurate results even for relatively few timesteps (for instance 100 time steps).

In Fig 5.3, it is shown how the voids deforms differently for different values of the Lode parameter. The plots are taken from the same time increment, and all the voids were initially spherical and of identical size.



(a) $\Theta_L = 1$ (b) $\Theta_L = 0$ (c) $\Theta_L = -1$

Figure 5.3: The voids grows differently depending on Θ_L

There are extremely small differences in the void volumes for varying Θ_L (not shown here), but the shapes differs to some degree. In the case of generalized shear and generalized compression, the void will grow more towards a "pancake" geometry, where the generalized compression will be the flattest and widest. These differences in how the shape turns out may result in differences in when they coalesce (See Chapter 6 for further discussion).

Unfortunately, for the case of varying Θ_L or σ^*, there exists no analytical (closed-form) solution to the Rice&Tracey growth equations. It is therefore assumed that since the numerical solution for the case when triaxiality and lode are constant clearly converges, the program will also converge for varying parameters, which is no more than partially constant parameters. The lode and triaxiality may change between time increments, but are constant within each. The convergence would probably be slower, so more time steps may be advised, but it is concluded that since the solution converges for constant parameters, the solution should also converge when the parameters are partially constant. As long as the solution of independent subproblems are correct, compiling the results from these subproblems should yield a correct solution.

5.2.1 Stop Semi-Axes of the Void from Decreasing Below Zero

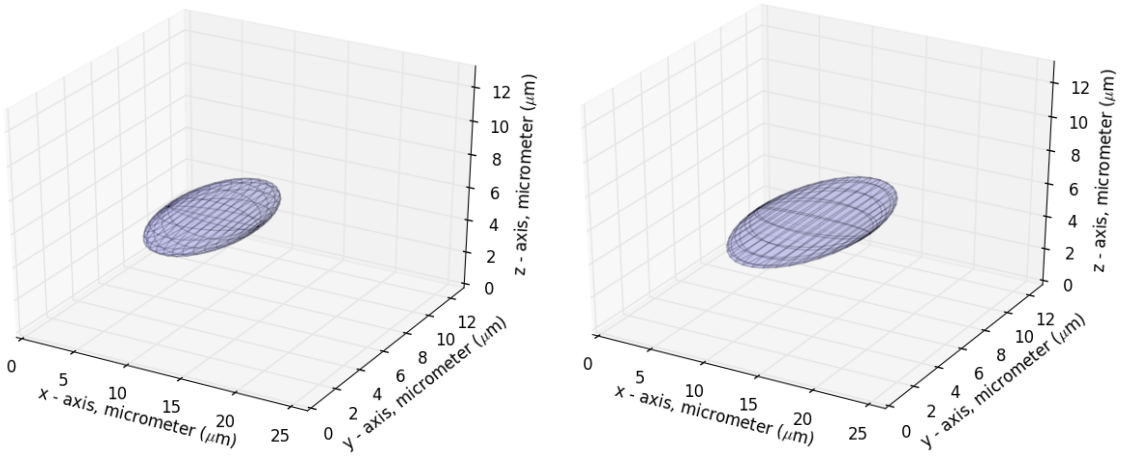
This may not be a very relevant case, since this may only occur for small values of the stress triaxiality. But it is made sure that instabilities are avoided for lower stress triaxialities, so it is possible to look at f.ex how the rotation develops during extremely large deformations (shown in the next chapter, see 6.1.1). During the method 'Rotate_and_grow()', the voids translates, grows and rotates. Each increment, the original orthonormal basis given by the **A** matrix' eigenvectors are adjusted as a result of the local velocity field $\mathbf{L}^*$ during the operation

$$\mathbf{g_1} = \mathbf{g_1} + \Delta t \ \mathbf{L}^* \dot{\mathbf{g}}_\mathbf{1} \tag{5.4}$$

where $\Delta t \ \mathbf{L}^* \dot{\mathbf{g}}_1$ from now on is denoted $\Delta \mathbf{g}$. To stop the semi-axes from getting negative values, the following procedure was chosen:

The ratio $\frac{|\Delta \mathbf{g}|}{|\mathbf{g}|}$ gets larger as $\mathbf{g}$ keeps decreasing in magnitude, which is determined by its corresponding eigenvalue. When this ratio reaches a critical value, the magnitude of $\mathbf{g}$ is kept unchanged during the following increments. For relatively small time steps, the semi-axes may reach values extremely close to zero without ever have to worry about the instability that may be caused from ill-conditioned equations. Note that this instability has nothing to do with numerical instability for the explicit solvers.

In Figure 5.4, it is shown how the voids will stop decreasing when the semi axis reaches a value close to 0. The corresponding evolution of the void's volume is shown in Fig 5.5, where the red dots show where in the deformation process the plots are taken from. Since only 40 timesteps are used, the void will not get a volume particularly close to zero before the semi-axis will stop decreasing. Notice that the void's volume will start to increase towards the end, because the semi-axis in the z-direction will have fixed magnitude from this point on, while the other semi-axes will keep increasing according to the Rice&Tracey growth equation.



(a) Decreasing void volume (b) Stopped decreasing semi-axis in z-direction

Figure 5.4: Growth of a void, $\Theta = -1$ and $\sigma^* = \frac{1}{3}$, only 40 timesteps

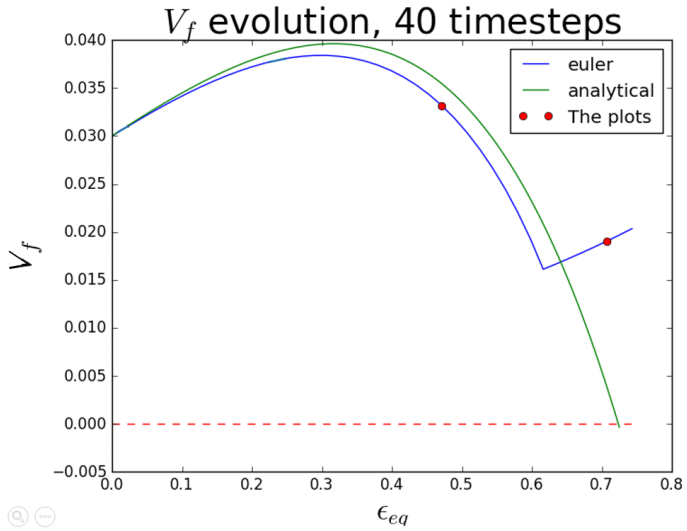
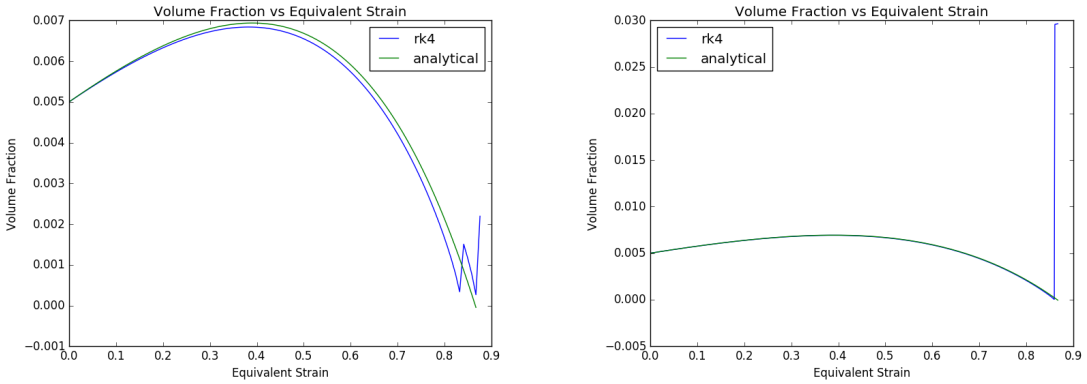


Figure 5.5: Numerical vs analytical solution

If more time steps are used, the voids will become practically completely flat, as the ratio $\frac{|\Delta \mathbf{g}|}{|\mathbf{g}|}$ will be below the critical ratio much longer, due to low values of $|\Delta \mathbf{g}|$. The volume of the voids will then get much closer to zero, see Fig 5.6b. In Fig 5.6, it is shown what will happen if this procedure hadn't been implemented. If it had been active, the void's volume would have stayed very close to zero during further deformation, instead of the instabilities shown.

5.3 Differences of Euler, Heun, RK4

Figure 5.7 clearly shows how small differences there is between the different solutions, even for as few timesteps as 40 (which nonetheless is *way* fewer than recommended, because of the desired accuracy). They are *very* close to each other, and all of them is as expected for 40 timesteps a bit different from the analytical solution.



(a) Instability for 35 timesteps

(b) Instability for 500 timesteps

Figure 5.6: Instability because of decreasing voids, 35 timesteps and 500 timesteps, RK4 solver

Based on the theory presented on the explicit solver methods in Section 5.3, much larger differences between Euler, Heun and RK4 were expected. But, the reason for the very similar results have an explanation:

The process of defining the local velocity tensor $\mathbf{L}^*$ based on the voids global semi-axes, is the source behind the rather unexpected results. The fraction $\dot{R}_i/R_i$ gets very similar values for all methods, since RK4 gets more correct values for R_i, which results in a "more correct" value for the next time increment than the other methods manages. But, since the Euler method results in a lower value for R_i, it therefore results in higher growth rate of the void through $\mathbf{L}^*$. Heun is somewhere in between of the results obtained from Euler and RK4. The results turned out to be *very* similar in the end, for all the explicit methods. They were in fact so similar that the author several times checked the numerical values to be sure that RK4 hadn't been used by an accident for all the simulations; but there are indeed small differences between the calculated values.

It may be concluded that the Euler method should be used in the simulations instead of Heun or RK4, since the results are so similar, but the Euler method is computationally more effective. But, every possible combination of void growth hasn't been checked, so there may yet exist undiscovered scenarios where the differences between Euler, Heun and RK4 are larger. In a "rather safe than sorry" mentality, the RK4 solver was therefore still used for all the results obtained in this thesis.

5.4 Asymptotic Running Time

When writing code, almost all problems one may stumble upon does have a 'brute force' solution. This simply means that every options is calculated and compared to find the solution one is searching for, or the 'optimal solution'. In this thesis, a lot of effort have been made to come up with better algorithms than this 'brute force' alternative. A 'brute-force' solution is usually pretty easy to implement, but also quite computationally demanding, and thereby setting a limit on the problem size.

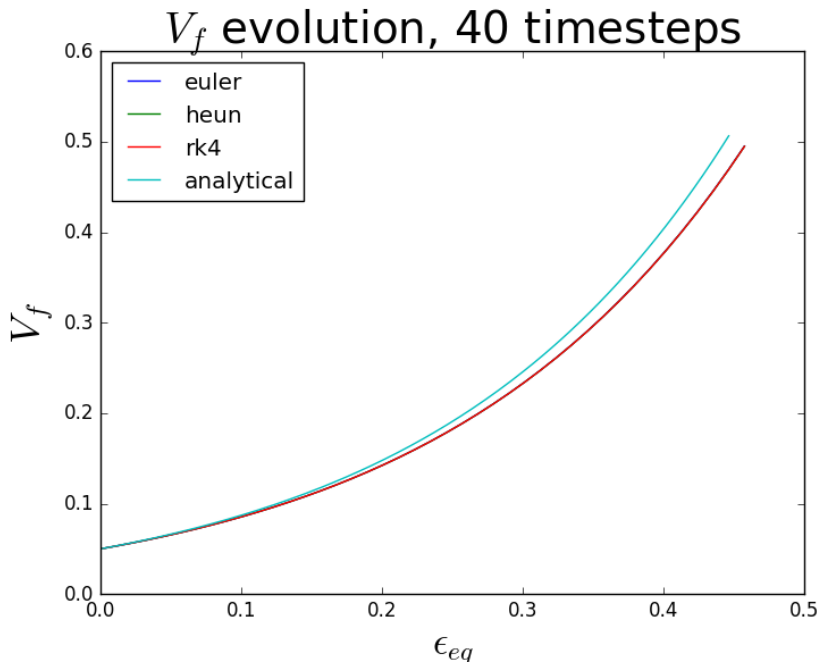


Figure 5.7: The numerical solvers compared to the analytical solution

In this section, the asymptotic running time of the program will be stated. The flowcharts (and of course the pseudo code) of the program is tightly coupled with this section.

To show the asymptotic running time as easily as possible, the pseudocode of the program is used, but this time all the commands have written their respective running times at the end of the same line. By remembering the principal rule when working with asymptotic analysis (see 3.4.4), that it is allowed to drop all the constant parts of our functions, the program turns out like shown in Fig 5.8. The total running time complexity for the initialization part of the program becomes $\Theta(n^2)$, which is because every void created must be compared with all the others to avoid initial collision, resulting in $n(n+1)/2$ comparisons.

The running time complexities for the main part of the program of the program is shown in Fig 5.9. Here, the running time is found to be $\Theta(N*n^3*m*TOL)$. N means the number of time steps, TOL is the tolerance given in the MVE ellipsoid algorithm, n is the total number of voids, and m the average number of neighbors for each void.

Asymptotic notion is used to tell how the program responds to changes in the input on the running time. In this thesis, it was also used to find the "correct" parts of the program to try to optimize.

The only part of the code (see Appendix A for the source code) that is not as good as desired, is the algorithm that updates the neighbor lists every k^{th} timestep. The algorithm used is a brute force algorithm, that checks the distances between all the voids' centers, and if the centers are closer than the sum of the two voids largest principal axis times a factor of 1.1, assigns them as neighbors. The value $k = 5$ was used to obtain the results herein. On the bright side, just computing the norm of a vector in 3D and comparing it to the voids' semi axes, is not a very computational expensive procedure. If all the voids had been controlled

Algorithm 4: Running Time Complexity - Initialization part

```
for each void wanted Θ(n) do
    if orientation == something Θ(1) then
        Generate desired orientation Θ(1)
    else
        Generate another type of void... Θ(1)
    end
end
Calculate void volume Θ(1)
Define box Θ(1)
for each void wanted Θ(n) do
    Generate position Θ(1)
    for each void instantiated Θ(n) do
        if collision with new void Θ(1) then
            generate another position Θ(1)
        else
            instantiate void Θ(1)
        end
    end
end
```

Figure 5.8: Running Time Complexity - Initialization part

for intersections with each other, the algorithm is still a brute force strategy, but it would have been *a lot* more expensive. By filtering out the voids that have no possibility of intersecting each other based on distance between each other and size, the computations becomes *much* faster. This process is called *pruning*, which in computer science means that subproblems (intersecting ellipsoids in this case) whose solution is known beforehand to give a "negative" result, are discarded without further consideration.

Nonetheless, the problem of finding current neighbors is possible to solve by a Divide&Conquer strategy, which would have resulted in a running time of $\Theta(Nn^2 \cdot lg(n)m \cdot TOL)$ instead of $\Theta(Nn^3m \cdot TOL)$. Unfortunately, it was not found enough time to implement a Divide&Conquer algorithm for this problem. This topic is therefore mentioned in the Chapter "Extensions of the Modeling Framework". Pay attention to the fact that the running time of finding the convex hull to remove part of the void's volume is not included in the calculated running time of the program. This functionality was as mentioned not used to obtain any of the results in this thesis, and therefore it was decided to not include it in the running time complexity for the CA model.

Algorithm 5: Running Time Complexity - Main part

```
while V_f ≤ V_F^lim Θ(N) do
    Find neighbors every k^th increment Θ(n^2)
    A lot of constant operations, Θ(1)
    for each void Θ(n) do
        Grow,rotate and translate Θ(1)
        Add all the voids volumes together, to get V_f Θ(1)
        if user decides then
            subtract the volumes of the voids thats outside the box Θ(nlg(n))
        end
    end
    for each void Θ(n) do
        for each neighbor Θ(m) do
            if (void,neighbor) pair has been checked already then
                skip it Θ(1)
            else
                check collision between the ellipsoids Θ(1)
            end
            if collision then
                Sample points from both voids Θ(1)
                Create a MinimumVolumeEllipsoid around these points
                 Θ(TOL)
                Delete the two intersecting voids Θ(1)
                Instantiate the replacement MVEE void Θ(1)
                for each void Θ(n) do
                    Delete these two voids from the neighbor list Θ(1)
                end
                Find all the neighbors for the new MVEE void Θ(n^2)
                Add the (void,neighbor) pair to a list of checked pairs Θ(1)
            end
        end
    end
    Increase t: t+ = dt
end
```

Figure 5.9: Running Time Complexity - Main part

6 Validation

This chapter will start off with mentioning the shortcomings of some assumptions made in the implementation of the model:

In the Rice&Tracey growth equation, Rice&Tracey looked at one void in an infinite medium, and in this case the void volume fraction is of course 0. The assumption of an incompressible matrix are not violated. On the other hand, to consider a finite matrix with a large amount of voids, does violate the incompressibility matrix theorem of plastic flow. This was a necessary simplification in order to be able to use the Rice&Tracey growth equation to model the void growth in the CA model. As the voids grow, the volume of the matrix is reduced correspondingly. This section will do its best to uncover and discuss any shortcomings as a result of the assumptions and simplifications made in this model.

6.1 Comparing Results to Abaqus

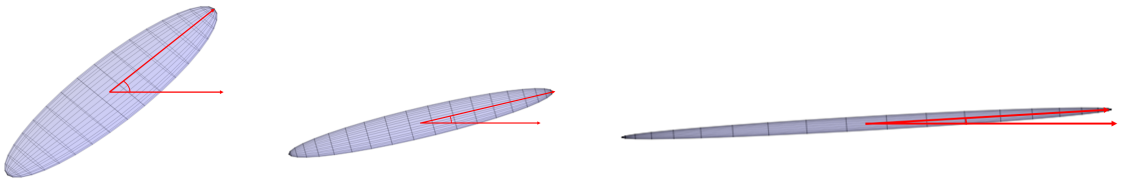
6.1.1 Rotation Compared to Abaqus

This section will compare how the voids rotates in the CA model compared to the exact same simulations in Abaqus. For all the following cases, the stress triaxiality is set as low as $\sigma^* = 1/3$, to avoid that the void will grow outside the box, which terminates the Abaqus simulations. This is to be able to fully focus on the rotation of the voids. Further, the lode parameter is equal to 1, which characterizes a load situation of generalized tension. The initial void volume fraction is 0.005.

1^{st} case - 45 degrees initial angle, R1=4R2=4R3

The void is initially rotated from the global coordinate system around the y-axis, an angle of 45 degrees. The semi-axes ratio is as mentioned in the title $R2 = R3, R1 = 4R2$, and the angle between the R1 axis and the x-axis is measured in both the CA model and in Abaqus. In Fig 6.1, it is shown angle vs principal strain along the x-axis (ε_{11}). The results must be said to be in very good agreement. It is seen that when ε_{11} exceeds 1.5, the angle from the CA model will keep decreasing towards 0, for so to be equal to 0 for further deformation. The angle in the Abaqus analysis will from $\varepsilon_{11} \geq 1.5$ on decrease more slowly.

It is worth mentioning that strain values of 1.5 is equal to a stretch ratio of $\lambda = e^{1.5} = 4.48$, which means ridiculously large deformations. No results of interest should be taken from such large strains in the CA model nonetheless, since the void will be so incredibly different from the spherical void that is assumed in the Rice&Tracey growth equations.



(a) Initial configuration (b) Early rotated configuration (c) Configuration at large deformation

Figure 6.1: 1^{st} case - 45 degree initial angle, R1=4R2=4R3

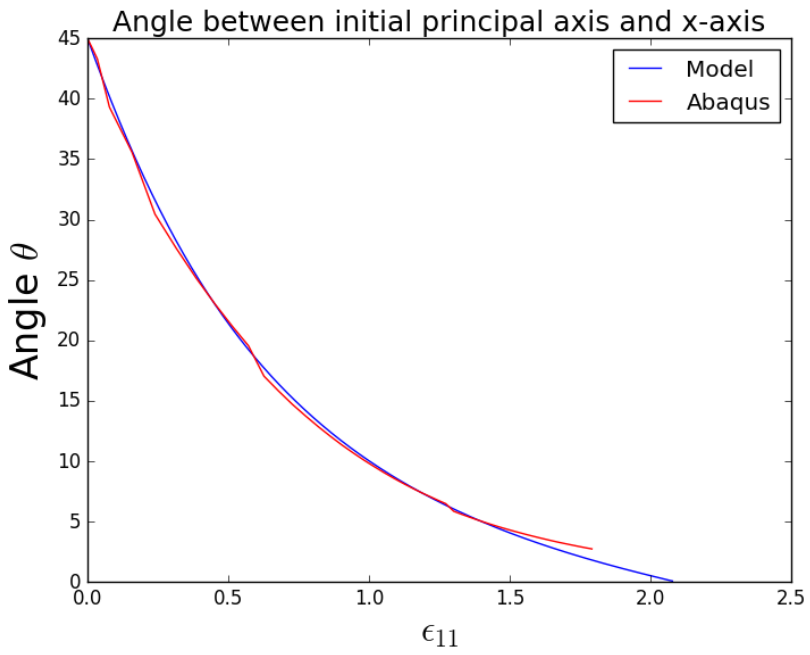


Figure 6.2: Comparison of the void's rotation; Abaqus and model

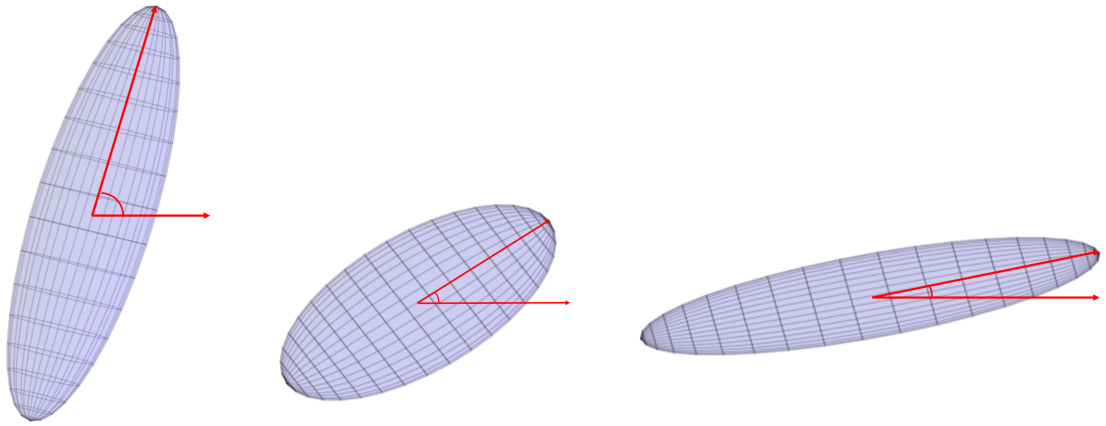
2^{nd} case - 75 degrees initial angle, R1=4R2=4R3

The void is now rotated an angle of 75 degrees, with the same semi-axes ratios as in the preceding subsection. This case shows possibly the largest differences between the implementation and Abaqus.

Initially, the void's largest semi-axis is in transverse direction of the largest strain direction (the x-direction). The void will be stretched in the x-direction, and it will rotate towards the x-axis. At a given deformation, the void will have several points with almost the exact same distance to the center of the void (see

Fig 6.6 and Fig 6.7 for plots from Abaqus of a very similar configuration). Abaqus assumes that the greatest distance to the origin is the principal axis. Therefore, Abaqus does not follow the same material point on the void's surface (as the CA model does), but rather the greatest distance to the origin. This is the reason for the sometimes irregular changes in the measured angle in the Abaqus results. The mesh used in the Abaqus model may also affect the obtained results, but the mesh used to obtain these results should be well within reasonable limits.

For several of the next cases, the changes in the angle between the largest principal axis and the x-axis are much more irregular. The above mentioned explanation is very likely the source of the irregular measures.



(a) Initial configuration (b) Early rotated configuration (c) Configuration at large deformation

Figure 6.3: Rotation of the void with 75 degree initial angle, R1=4R2=4R3

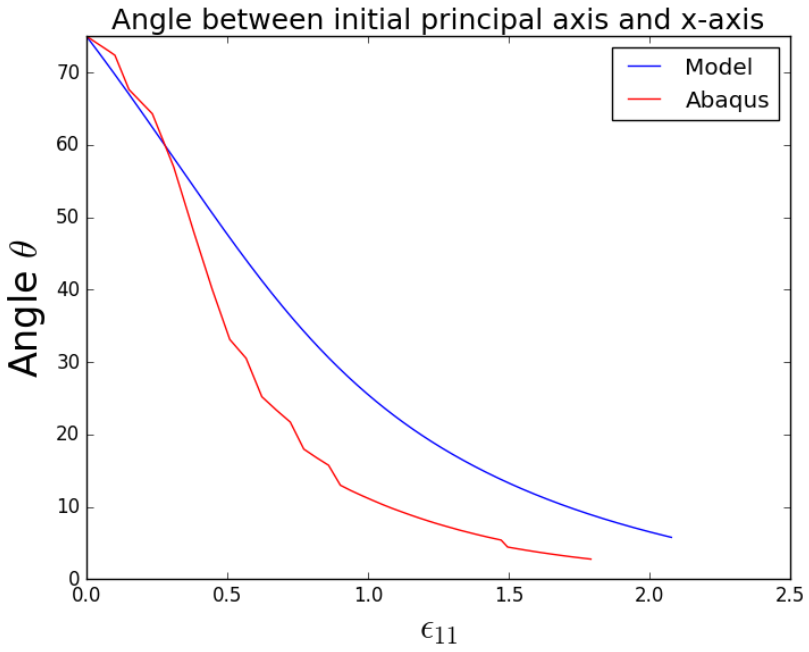


Figure 6.4: 2^{nd} case - 75 degree initial angle, R1=4R2=4R3

3^{rd} case - 15 degrees initial angle, R1=4R2=4R3

Once again (see Fig 6.5), very similar results are obtained from the model and Abaqus. At $\varepsilon_{eq} \approx 1.5$, the angle calculated from the CA model has reached an alignment with the global coordinate system. Further deformation does then not change the angle at all, which is exactly the physical behavior that can be expected!

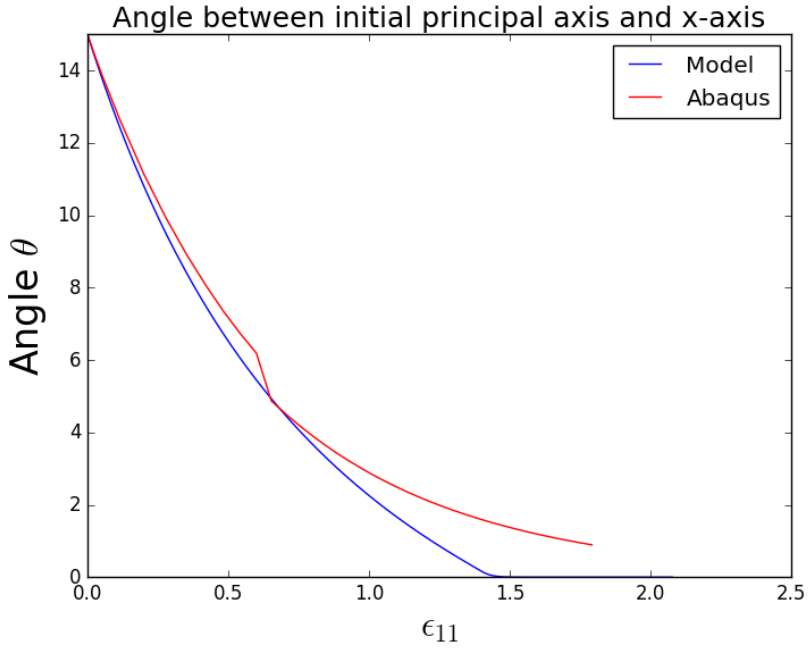


Figure 6.5: 3^{rd} case - 15 degree initial angle, R1=4R2=4R3

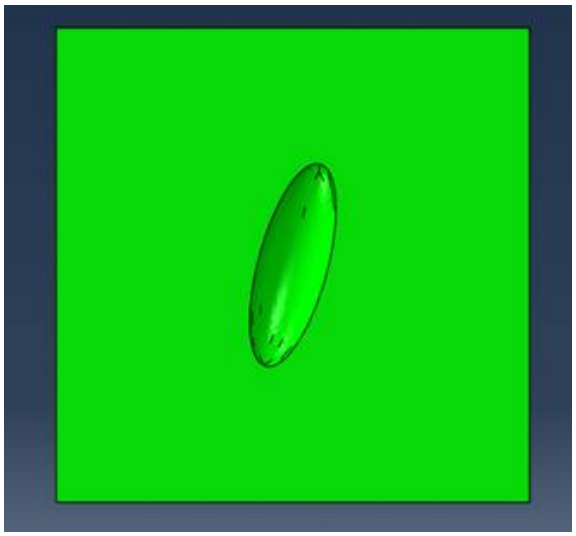
4^{th} case - 75 degrees initial angle, R1=3R2=3R3

In this case, the initial angle is still 75 degrees, but the weight ratios are slightly lower: R1=3R2=3R3. The graphical results from Abaqus are this time included (see Fig 6.6 and Fig 6.7), to show the graphical similarities during the deformation.

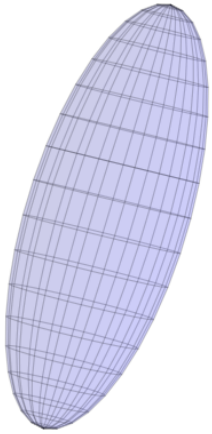
As Fig 6.8 shows, the angle computed in Abaqus is less smooth than the one obtained from the CA model. The sometimes irregular jumps in the Abaqus results are because of the change of the largest semi-axis, as mentioned in the preceding subsection. The Abaqus analysis was terminated early for this case, as the void had grown too much (nearly outside the box). Nonetheless, by comparing both the graphical and analytical results, they are once again quite similar.

5^{th} case - 85 degrees initial angle, R1=2R2=2R3

In Fig 6.10, it is shown how the change of the largest semi-axis happens. This change results in a discontinuous measurement of the angle and a change in which way the void rotates, shown in Fig 6.9. For a stretch ratio above 1 in the x-direction ($\lambda_1 > 1$), the void will rotate clockwise until the change of largest principal axis happens, and then the principal axis will change by 90 degrees.

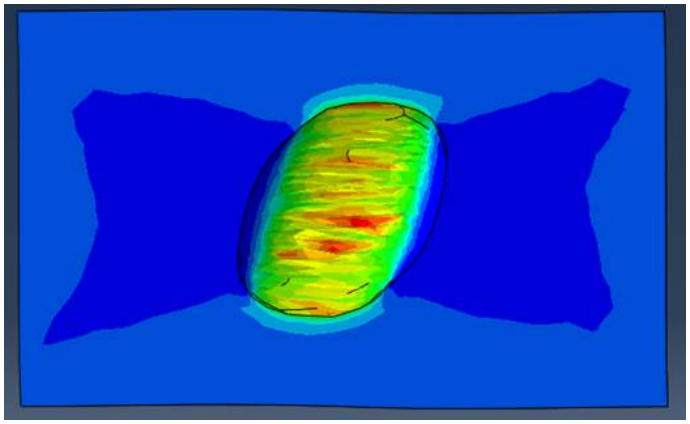


(a) Initial configuration in Abaqus

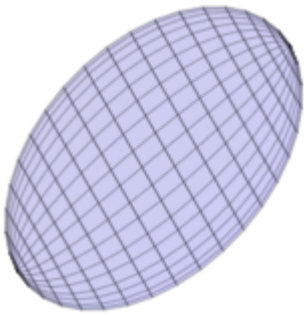


(b) Initial configuration in model

Figure 6.6: 4^{th} case, graphical comparison



(a) Final configuration in Abaqus



(b) $\approx$ corresponding configuration in model

Figure 6.7: 4^{th} case, graphical comparison

After this change, the largest principal-axis will now be oriented such that the void will rotate counterclockwise from here on out. This process is shown in Fig 6.11. The exact opposite will happen for $\lambda_x < 1$.

6^{th} case - 89 degree initial angle, R1=2R2=2R3

In Fig 6.1.1, the angle calculated by both the model and Abaqus are shown. The trend in both this case and the case of an initial angle of 85 degrees, is that the change in largest semi-axis happens a bit earlier in the Abaqus simulation.

To summarize this whole section; the obtained results are very similar. For the cases when the initial angle is quite large (75 degrees and larger), it seems like Abaqus rotates the void slightly faster than the model. For 45 degrees, the results are more or less identical, while for the case when the voids are rotated only 15 degrees, it actually rotates slightly faster in the CA model. The voids in the CA model will actually reach an angle of zero degrees, while that never

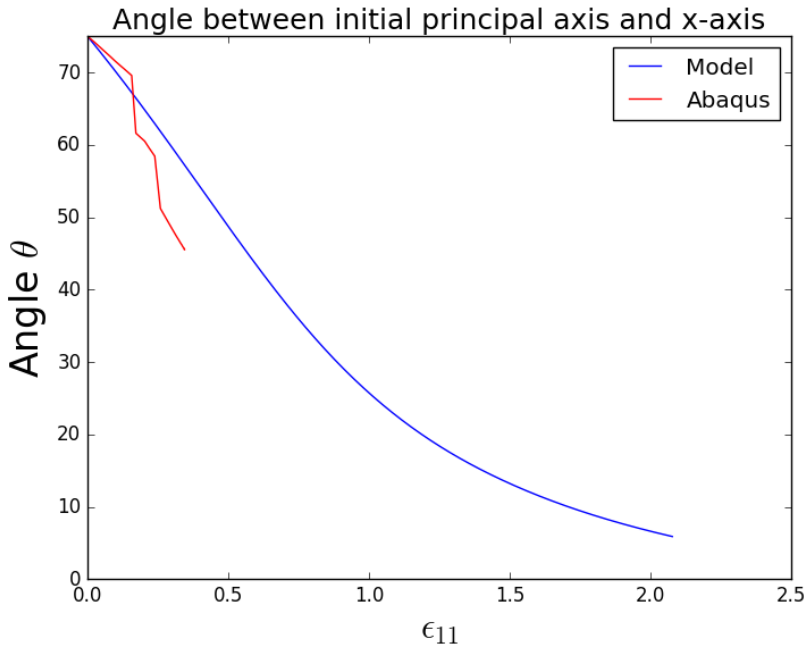


Figure 6.8: 4^{th} case - 75 degrees initial angle, R1=3R2=3R3

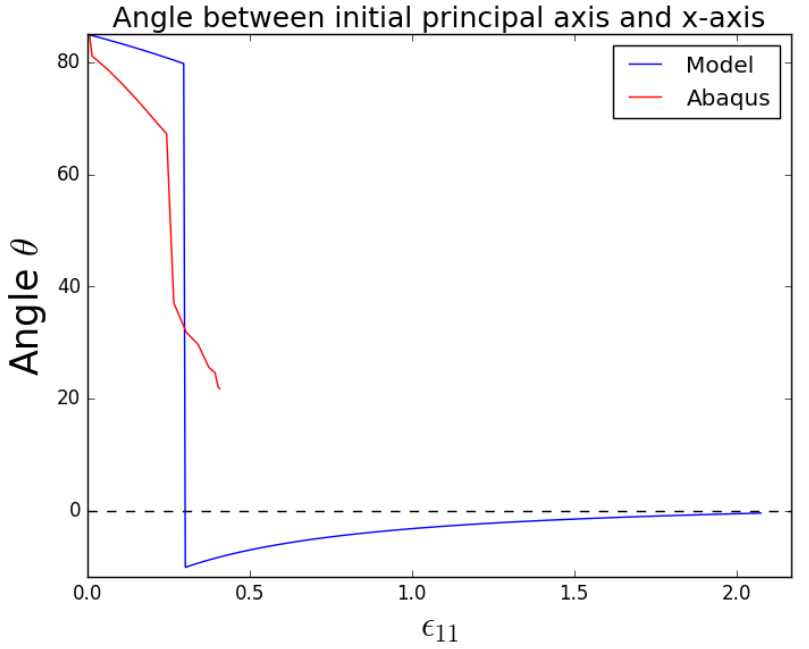
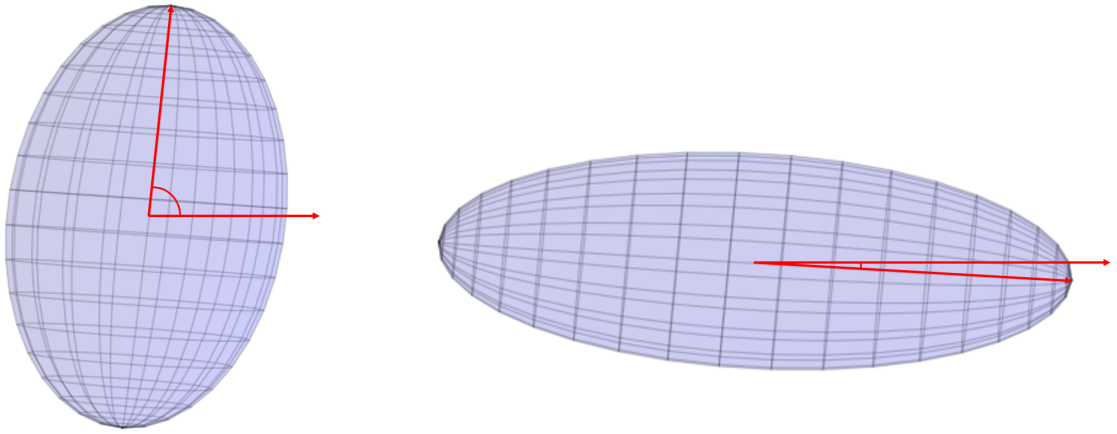


Figure 6.9: 5^{th} case - 85 degree initial angle, R1=2R2=2R3

happened in Abaqus. When the voids got pretty close to being aligned with the x-axis in Abaqus, they rotated very slowly, much slower than the CA model. Even though there were some differences, the results seems to be very promising. It is concluded that the rotations are described with sufficient accuracy.



(a) Initial configuration (b) Configuration at large deformation

Figure 6.10: 5^{th} case - 85 degree initial angle, R1=2R2=2R3

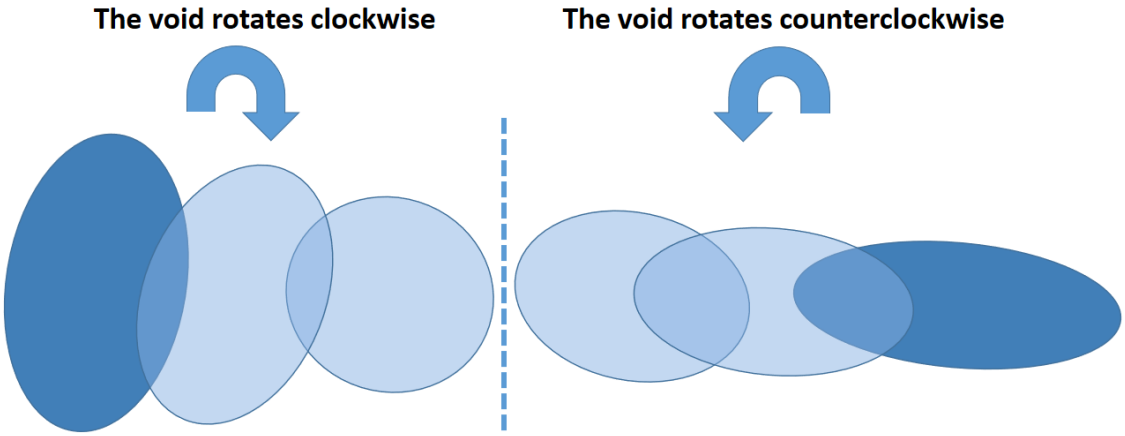


Figure 6.11: The voids need not rotate the same way throughout the whole deformation process. Here the deformation process between Fig 6.10a and Fig 6.10b are shown.

6.1.2 Growth Compared to Abaqus

In this section, the growth of the voids according to Rice&Tracey will be compared to the corresponding results from Abaqus. In Figure 6.13, the void volume evolution of an initial spherical void in Abaqus and the CA model, for three different Lode values are shown. In this case, the results are not as similar as in the void rotation cases in Section 6.1.1.

It is evident that the voids modeled in Abaqus will grow more slowly in the beginning, for so to start to grow very fast when the equivalent strain has reached a given value. A trend in the results may seem to be that the Rice&Tracey growth equation gives conservative results, except for rather large strain values.

To summarize the topic on void growth, the growth of the voids does clearly not experience the same growth evolution. The Rice&Tracey model follows an exponential function, while the results from Abaqus seems to have a very increased growth rate towards the end. The Abaqus analysis terminated at this point, because the void became to large, but if the analysis had continued, it may seem

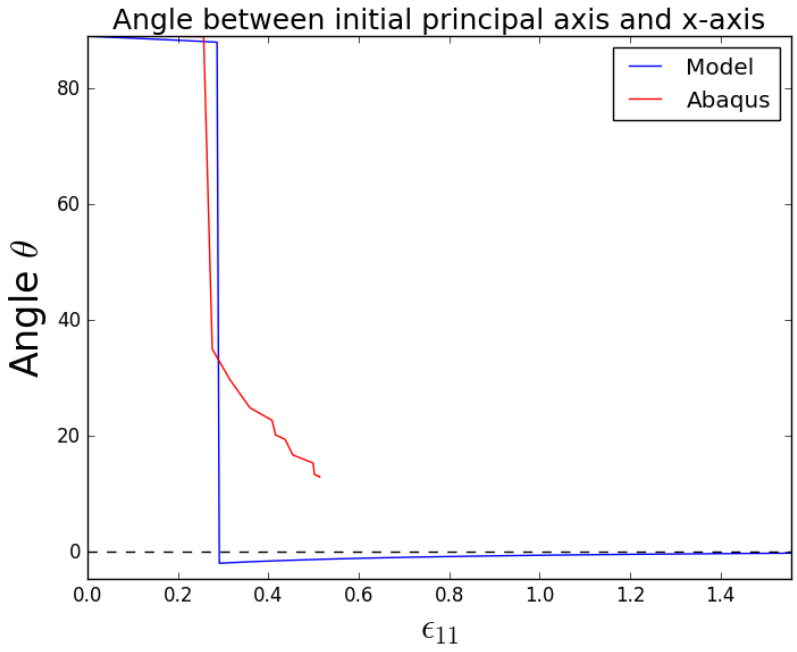


Figure 6.12: 6^{th} case - 89 degree initial angle, R1=2R2=2R3

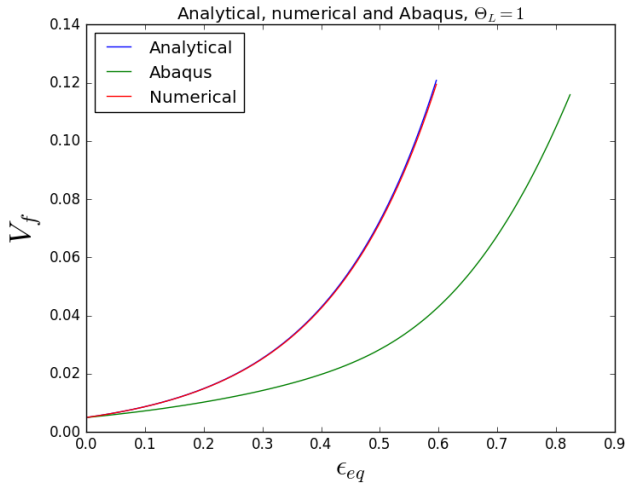
like the results for the Rice&Tracey model may not be conservative for very large values of the strain. As will be shown in the next chapter, the model will predict ductile fracture for much lower values of the equivalent strain than the values needed for the Rice&Tracey growth model to show less conservative results than Abaqus.

Therefore, it is concluded that the void growth equation given by Rice&Tracey is more conservative than the Abaqus simulations for all values of the equivalent strain that are relevant for the model. Coalescence and ductile fracture usually happens for much lower values of the equivalent strain ε_{eq} than the values needed for Abaqus to give more conservative results than the CA model. These values for the equivalent strain are approximately 0.8 ($\varepsilon_{eq} \approx 0.8$). The exception may be when $\Theta_L = -1$, where Abaqus will give more conservative results already from $\varepsilon_{eq} = 0.55$. But, coalescence will usually have happened before values like this are reached anyway.

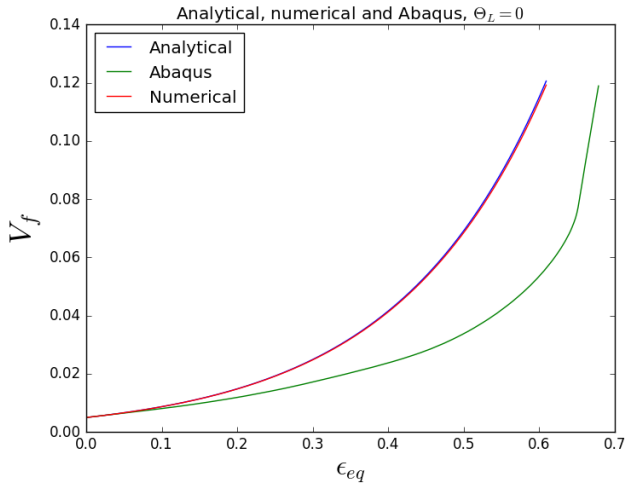
6.2 Spherical Void Growth - Rotated Semi-Axes

In this section, it will be studied what happens when the voids are initially spherical, but their principal axes are not aligned with the global coordinate system (the principal deformation directions). By comparing the numerical results from voids aligned, and not aligned, with the principal deformation directions, it can be measured how much error is introduced because of the heuristic approach of defining an equivalent void as described in Section 2.4 and shown in Fig 4.1. The results called 'Spherical error', is from an initial configuration where the void is rotated 45 degrees around the y-axis from an aligned position.

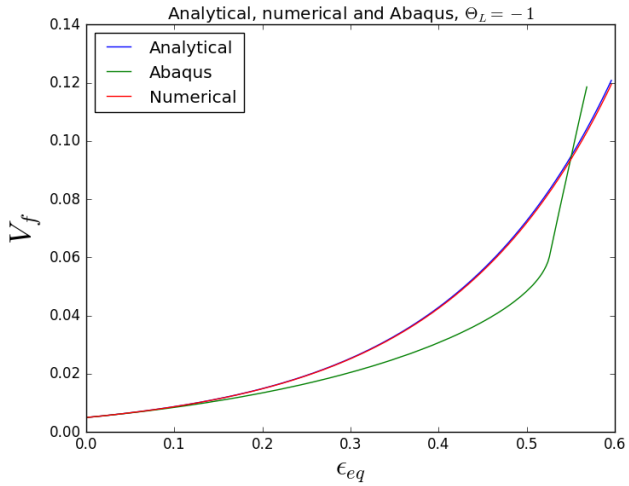
In Fig 6.14 and Fig 6.15, it is shown how the two different scenarios grows differently. The initially rotated void begin to rotate as it grows, something that



(a) Growth in Abaqus, $\Theta_L = 1$

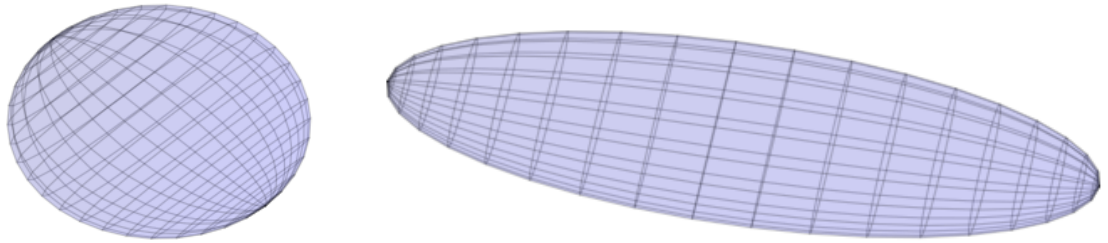


(b) Growth in Abaqus, $\Theta_L = 0$



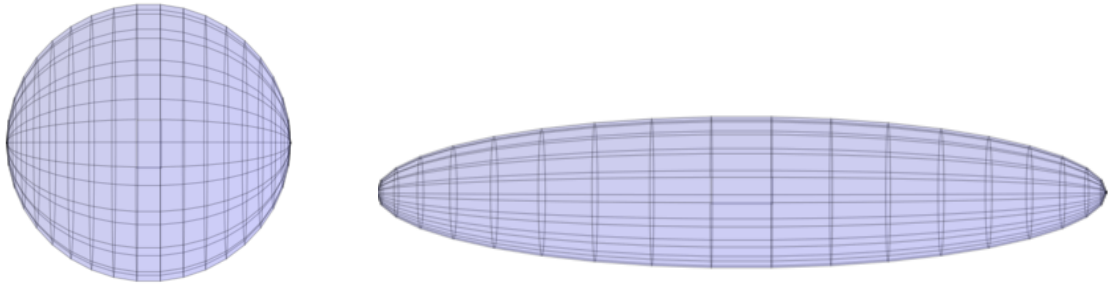
(c) Growth in Abaqus, $\Theta_L = -1$

Figure 6.13: Void growth from Abaqus compared to the CA model's numerical and analytical solution (where the latter two are shown to overlap)



(a) Initial configuration (b) Configuration at large deformation

Figure 6.14: "Spherical error"



(a) Initial configuration (b) Configuration at large deformation

Figure 6.15: "Spherical"

is obviously wrong. So, the rotation of the voids will apparently be affected by the heuristic approach of defining an equivalent void, but not to any great extent.

In Fig 6.16, it is shown the void volume fraction evolution when 100 time steps are used, $\Theta_L = 1$ and $\sigma^* = 1$. The differences are extremely small, but in this particular case the initially rotated void is actually closer to the analytical solution. The Rice&Tracey equation states how the semi-axes grows, and for the rotated void, all the three semi-axes are further away from the analytical solution than when the void were aligned, even though the volume is closer to the analytical evolution. R1 is a bit smaller, but R2 and R3 is a bit larger. A void's volume is as known by now $4\pi R1R2R3/3$, so the errors will actually help in this case. This of course means that it is pure luck that the void's volume gets a more accurate value for the non-aligned principal axes.

Nonetheless, the differences in the size of the void (semi-axes magnitudes) are absolutely negligible, even for as few time steps as 100. For the cases when $\Theta_L = 0$ and $\Theta_L = -1$, the exact same trend is showing. It seems like the heuristic approach of solving the growth equation may be quite an accurate approach.

6.3 Parameter Studies

Up until this point, it has been shown that the CA model quite accurately describes the voids behavior. In this section, the program is used to calculate the fracture strain of a given configuration of voids, which is the ultimate test. It shall now be revealed whether the results produced from the CA model gives results that are similar to what is known from experiments and other means of numerical analysis. As done in Hannard et al.[27], when coalescence happened, the new

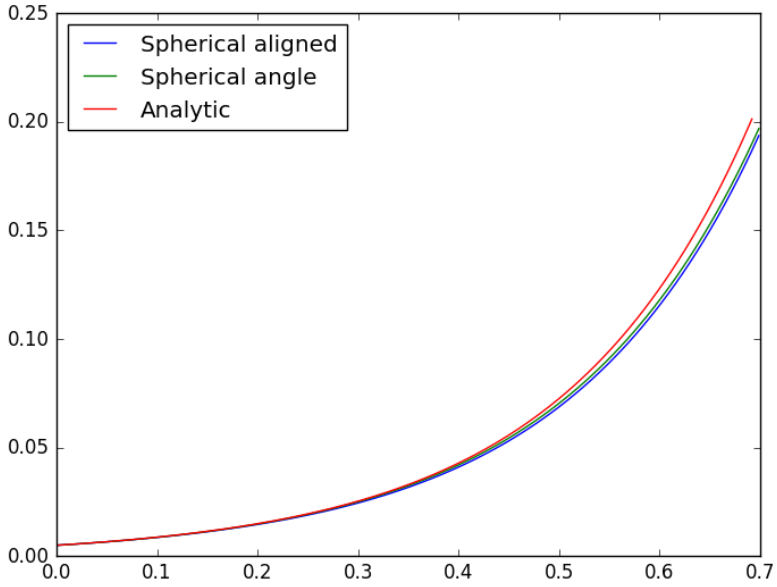


Figure 6.16: Void Volume fraction evolution, 100 timesteps

void was chosen to be modeled as a minimum volume enclosing ellipsoid. The same strategy was used in this model. Everything about the procedure was explained in Section 4.3, but the results will be discussed further here. The new MVE-ellipsoid will have larger volume than the sum of the two separate ellipsoids that are replaced. This means that every coalescence will result in a "jump" in the void volume fraction. This jump should actually be as small as possible, since the matrix can't disappear.

6.3.1 Parameter study 1 - Increasing Number of Voids

The first study is a parameter-study of how the model responds to an increasing amount of voids in the model, while holding the void volume fraction constant. The stress triaxiality is set to $\sigma^* = 1$, and the lode parameter as $\Theta_L = 1$ in all the simulations shown here. This is done by taking a much larger portion of the matrix under consideration. The purpose of this study is to see whether the fracture strain will converge towards a solution, which of course is the ideal scenario. How many voids (or equivalently; how large part of the matrix material) must be considered to yield results that seems to have converged?

The CA model is run 5 times for each value of n voids, where n takes the values 10, 50, 100, 200, 300, 500, 1000. Ductile fracture is assumed to happen when the void volume fraction reaches a critical value, in this case set as 0.2 of the box's volume ($V_f^{crit} = 0.2$). The equivalent strain ε_{eq} when this critical void volume is reached, is defined as the fracture strain ε_{eq}^f. The results are presented in the figures 6.17,6.18, 6.19, 6.20, 6.21, 6.22 and 6.23. The resulting expected (mean) fracture strain from these simulations are shown in a table (see Fig 6.24), where it may be easier to see how the ductility keeps decreasing as the number of voids used in the CA model are increased. A trend is quite clear, namely that

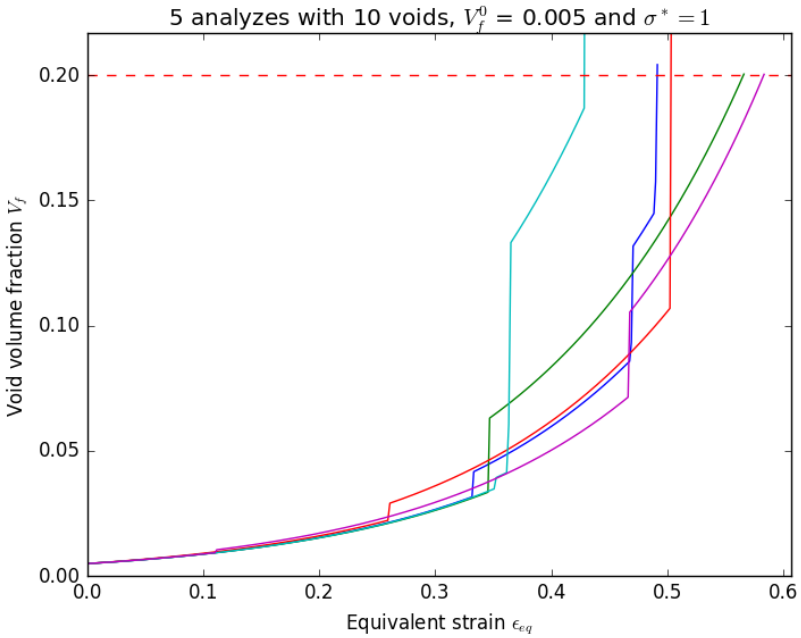
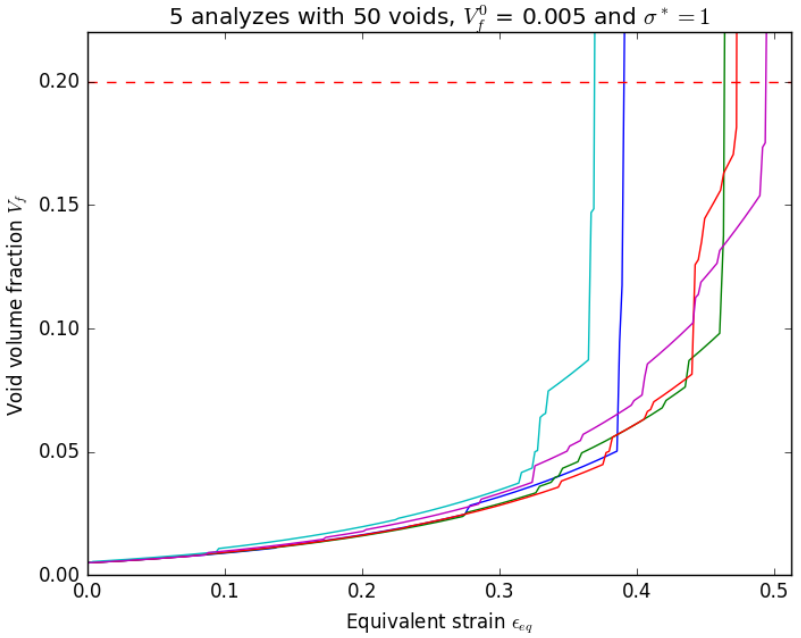


Figure 6.17: Void Volume evolution, 5 cases, 10 voids in the model

Figure 6.18: Void Volume evolution, 5 cases, 50 voids in the model



the ductility keeps decreasing as a larger part of the matrix is taken into consideration in the CA model. To measure the spread in the results, the largest and smallest equivalent strain ε_{eq} is identified in each case of a constant amount of voids, and the ratios presented in Fig 6.25 is found as the largest ε_{eq} divided by the smallest.

$$ratio = \frac{\varepsilon_{eq}^{max}}{\varepsilon_{eq}^{min}} \tag{6.1}$$

Figure 6.19: Void Volume evolution, 5 cases, 100 voids in the model

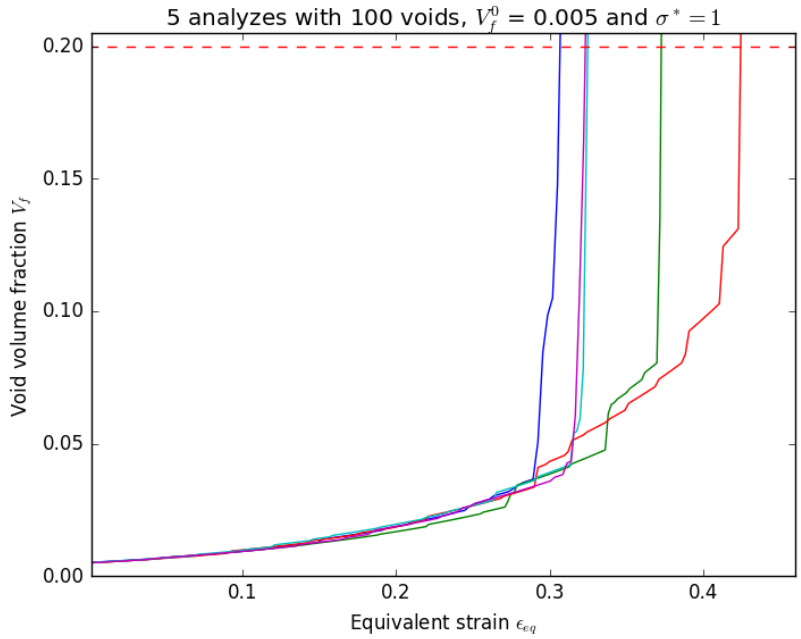
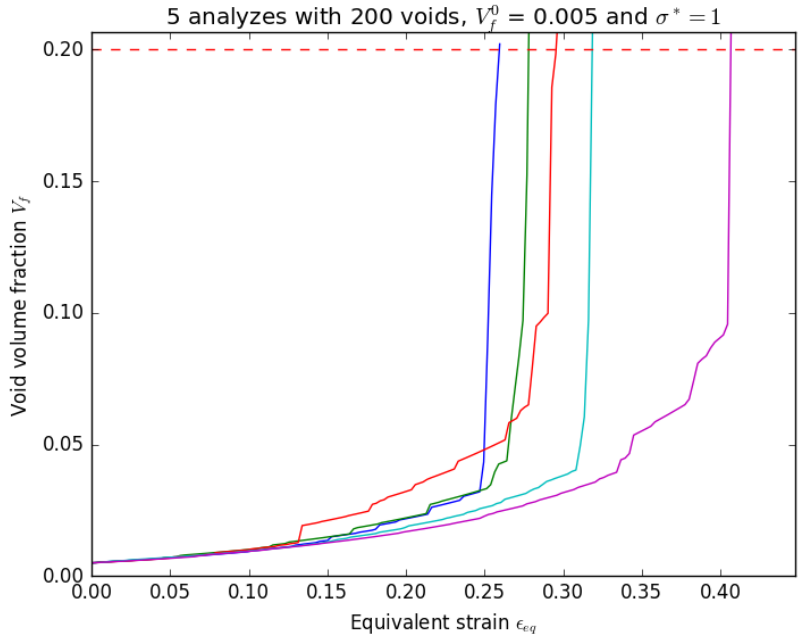


Figure 6.20: Void Volume evolution, 5 cases, 200 voids in the model



This ratio would ideally converge towards 1 as the number of voids under consideration increases. Unfortunately, the ratio does not seem to converge. The reason is assumed to be a combination of how the voids are given a position based on a uniform distribution, and that the MVE ellipsoid results in a void with at times *considerably* larger volume than the sum of the two colliding voids (The "Pac-Man" effect, see Section 4.3). This is especially true if the intersecting voids have quite different orientations. As the number of voids considered in the model are increased, the higher the probability of a "cluster", which means that

Figure 6.21: Void Volume evolution, 5 cases, 300 voids in the model

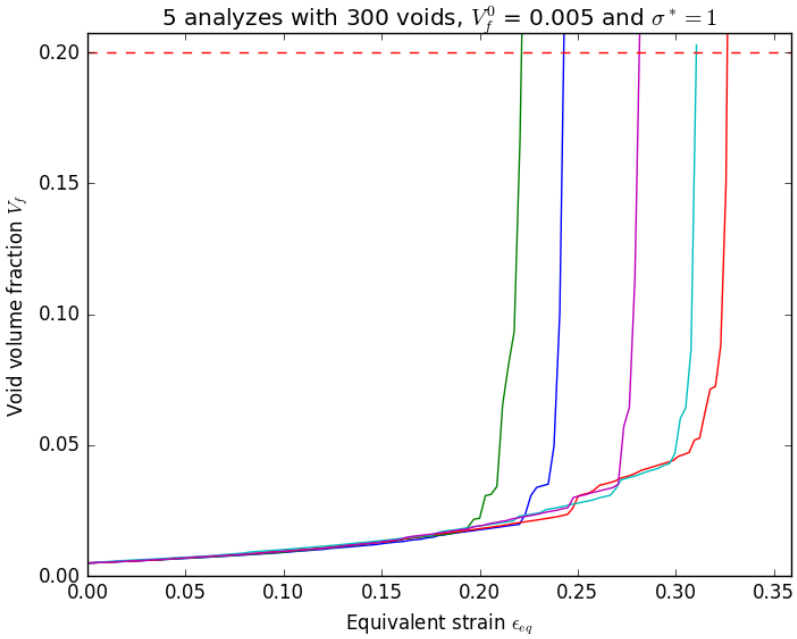
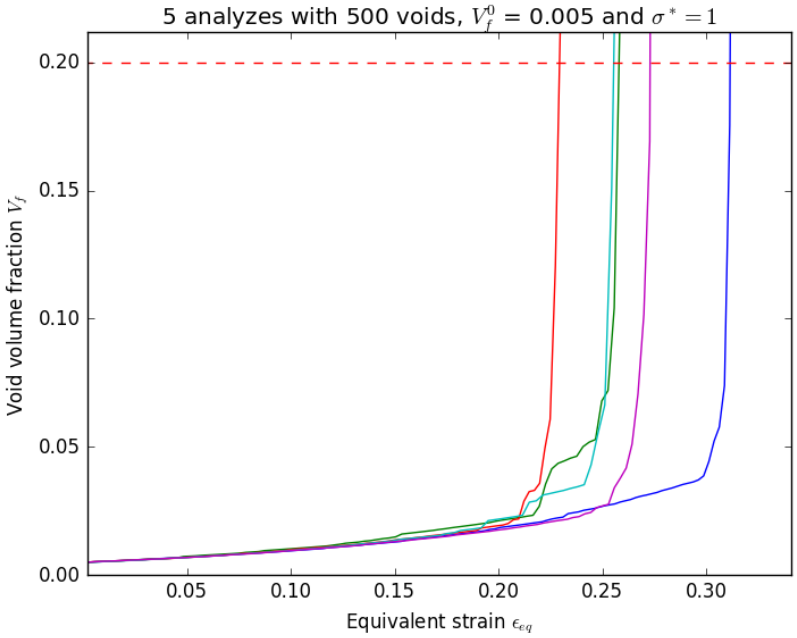


Figure 6.22: Void Volume evolution, 5 cases, 500 voids in the model



some of the voids are positioned closer to each other somewhere in the box. This means that the first coalescence comes earlier, and the more voids, the higher the probability that some of these voids have unfavorable positions considering the ductility. Since the coalescence process is allowed to flourish within each time step, one coalescence may turn into a series of coalescences, and thus the void volume typically reaches the critical value earlier than if first coalescences were postponed.

As the MVE-ellipsoid has larger volume than the two intersecting voids, it

Figure 6.23: Void Volume evolution, 5 cases, 1000 voids in the model

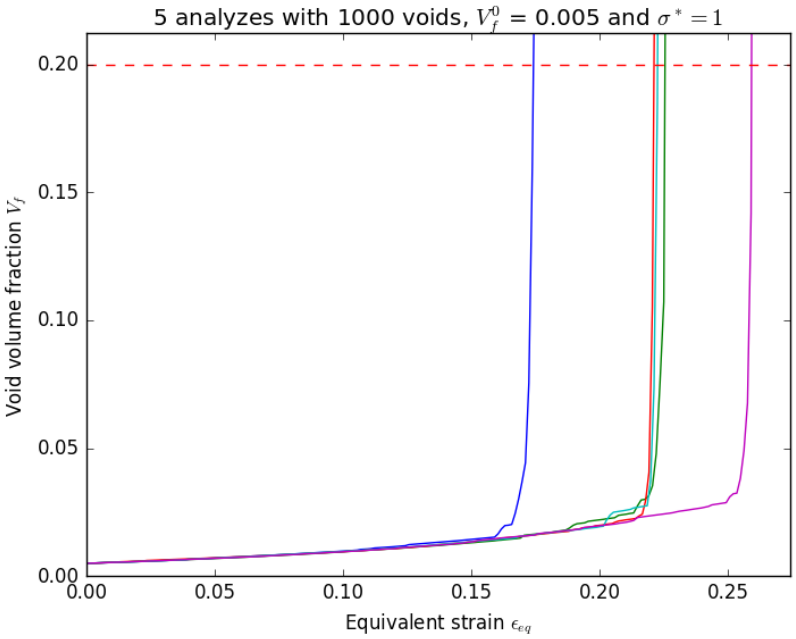


Figure 6.24: A table with fracture strains for the corresponding number of voids taken into consideration in the CA model

Number of voids n	10	50	100	200	300	500	1000
Fracture strain $\varepsilon_{eq}^f \approx$	0.52	0.44	0.37	0.32	0.27	0.26	0.22

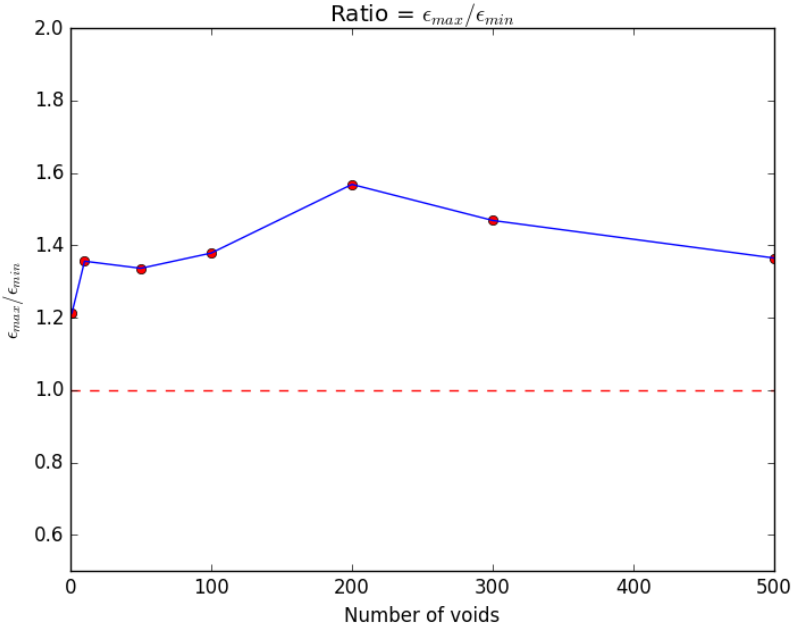


Figure 6.25: Ratio evolution from the parameterstudy

follows that each coalescence may remove a significant portion of the matrix. This means that the void volume increases in proportion to the number of co-

alescences during the deformation. Further, voids with a "safe" distance to the intersecting voids may still intersect the MVE void that replaces them. Each coalescence therefore increases the likelihood of a new coalescence. Because of both these effects, and *especially* the use of the uniform probability distribution for generating the voids centers, the coalescence process is more likely to flourish as more voids are taken into consideration. This is the reason for the increasingly steeper void volume evolution shown in the figures 6.17,6.18, 6.19, 6.20, 6.21, 6.22 and 6.23.

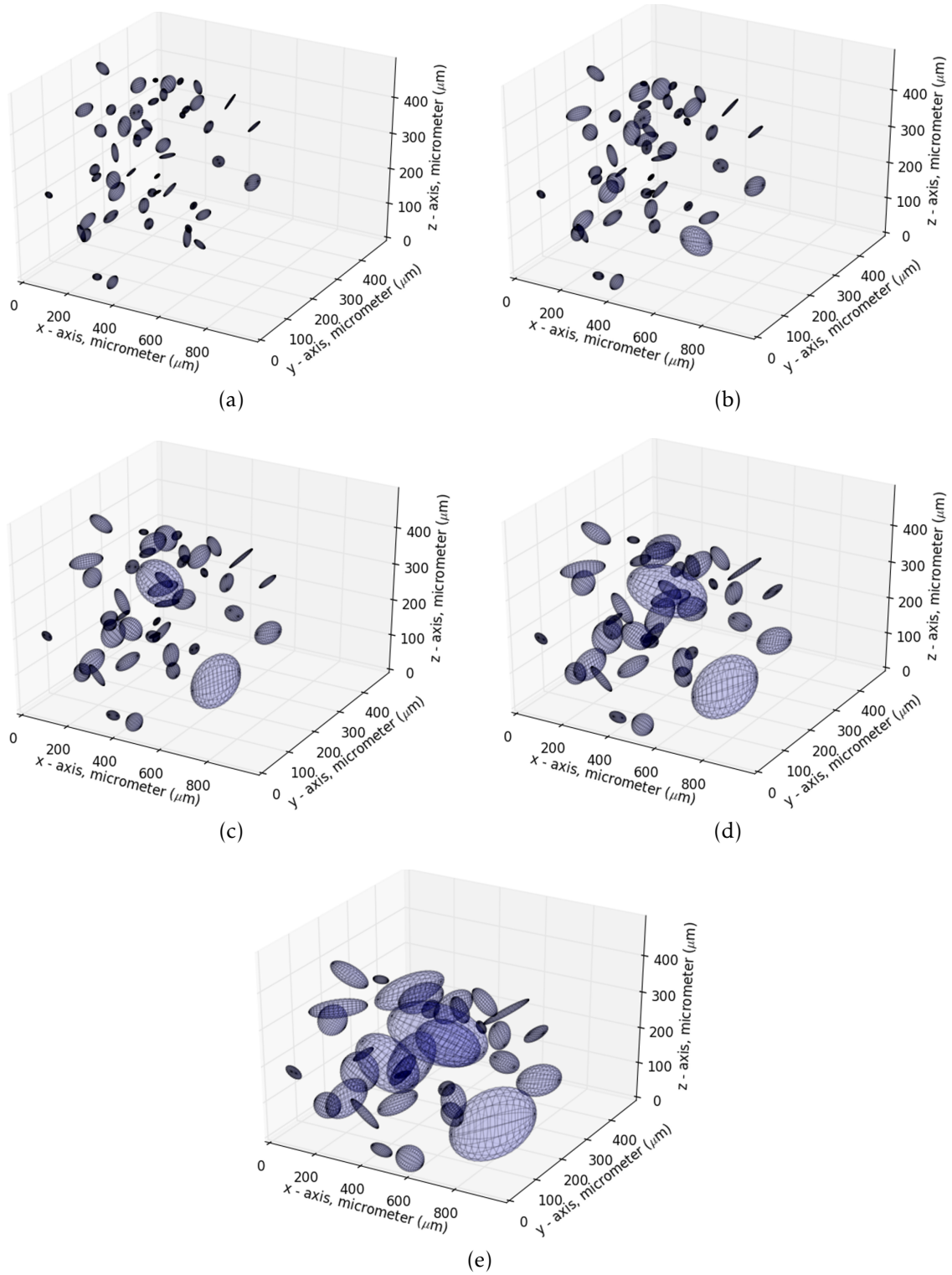
There are as mentioned two reasons for the increase in void volume: The growth of voids, and the "Pac-Man" effect. The growth of voids is the desired source, while the "Pac-Man" effect is most probably a source of error, since the matrix can't disappear in huge chunks at a time. It is assumed the matrix can be replaced by the voids as they grow, but during the deformation, there should probably not be any discontinuous jumps in the void volume. Therefore, another option was added to the program: The MVE ellipsoid algorithm is still used to produce the orientation and position of the new void, but the semi-axes are scaled so the new void does have the same volume as the sum of the two colliding.

In Hannard et al. [27], whom was the inspiration for this model, the position of the voids were determined by a CT scan of the tensile test specimen. In this thesis, the positions are generated from a probability distribution. The MVE-ellipsoid seemed to work perfectly in Hannard et al., but in this thesis it is seen as a possible improvement to reduce the volume of the MVE-ellipsoid to try to get a convergence as the number of voids are increased.

In Fig 6.26 the evolution of 50 random voids with $\Theta_L = 1$ and $\sigma^* = 1$ are shown. The red dots in 6.27 shows where in the simulation the plots are taken from. The red line in the plot is how much of the void volume that is the result of the "Pac-Man" effect. Pay attention to the fact that the "Pac-Man" effect is stepwise constant, since it is a result of disappearing matrix during coalescence only. The volume from the void growth are steadily increasing between these discontinuous gaps. The blue line is the void volume fraction calculated by the model. The green line shows the void growth for one spherical void given analytically from the Rice&Tracey growth equation, with the same initial volume fraction as is now divided among the 50 voids in the simulation. This illustrates the lower bound of the void volume fraction evolution. The more voids in the simulation, the closer this lower bound and the void volume fraction from the simulations in the CA model will be before the coalescence chain-reaction begins.

Pay attention to the fact that the axes limits in Fig 6.26 are held constant in all the plots, where the y- and z-axes are equal to the initial box's dimensions in these directions, while the x-axis is set equal to two times the box's initial dimension in the x-direction. It was concluded that the axes had to bee fixed to better understand the plots, and these values were chosen. Therefore, remember that the voids shown are actually twice as large in the x-direction as the plots may suggest.

The last plot in Figure 6.26 is just before the coalescence chain-reaction starts. This situation occurs earlier as the number of voids in the simulation are increased. In an attempt to avoid this situation, the MVE-ellipsoid may be replaced by a scaled ellipsoid with the same orientation as the MVEE, but with the same volume as the sum of the two intersecting voids.



(a) (b) (c) (d) (e)

Figure 6.26: 50 voids, $\Theta_L = 1$ and $\sigma^* = 1$. See Fig 6.27 for where in the deformation process the plots are taken from (the red dots)

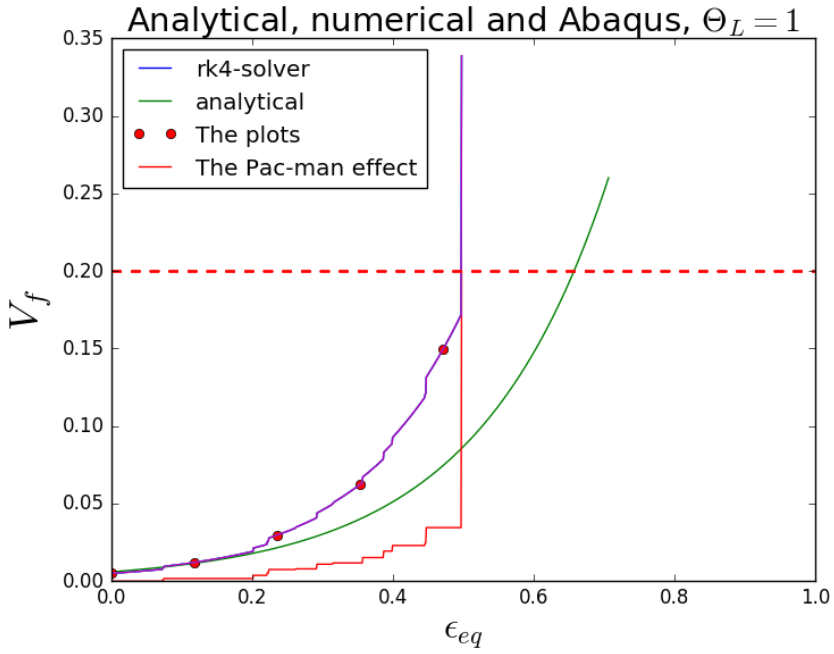


Figure 6.27: Void volume fraction evolution from the model

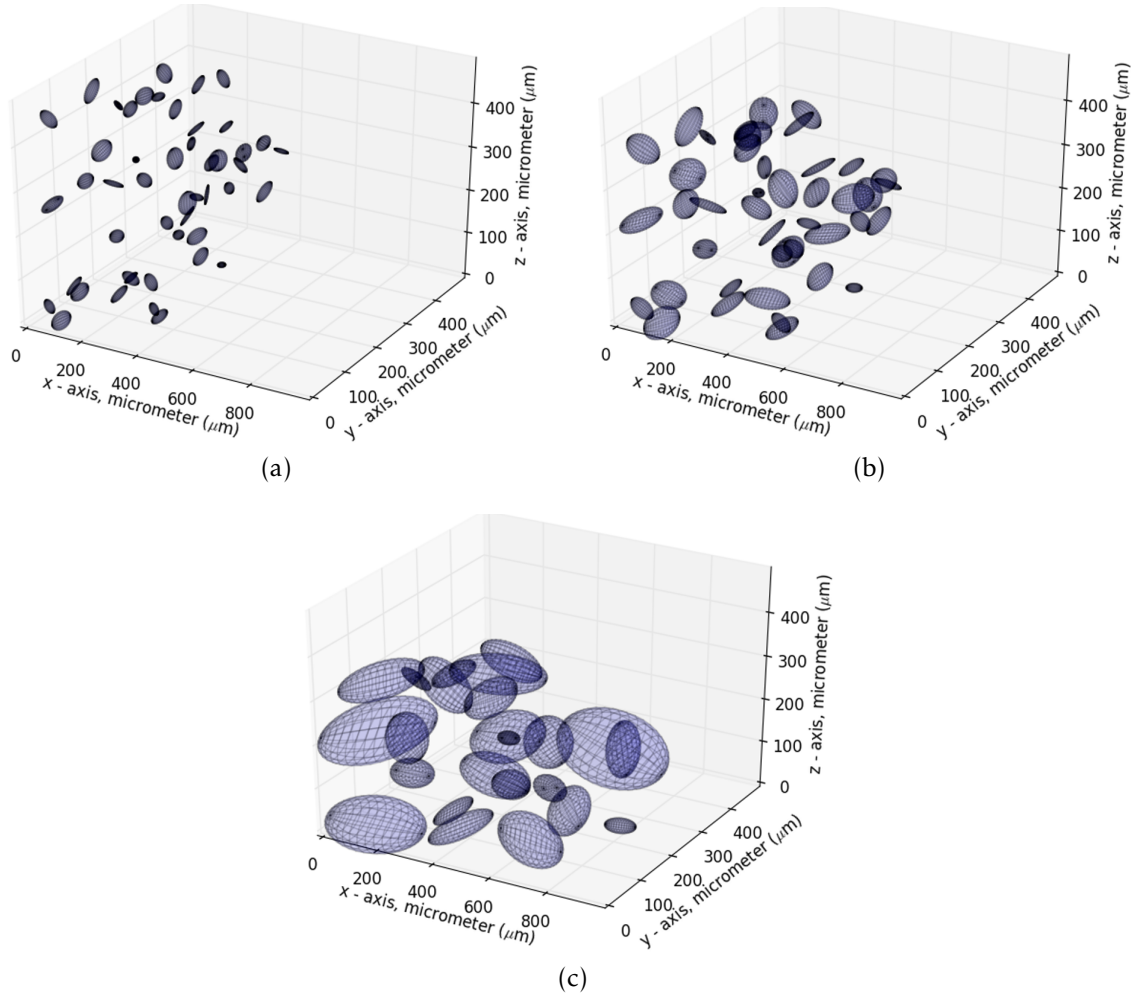
As shown in Fig 6.28, the "Pac-Man" effect is now removed. The fifty random voids with coalescence is now pretty close to the analytical solution of Rice&Tracey's growth equation, as shown in Fig 6.29. In this case with only 50 voids, the two solutions are pretty close, but as the void number increases, they becomes practically identical.

That the results converges towards the analytical solution is no coincidence. When the coalescence is adjusted so there will be no increase in the void volume, the model is back to a situation where no interactions between the voids are assumed. As the number of random voids are increased, and interactions between voids are "neglected", the results will converge towards the solution of the Rice&Tracey growth equation.

The "Pac-Man" effect may of course be adjusted between the two outer scenarios shown in this section. Therefore, as a means of calibrating the model for a set amount of voids, the Pac-Man effect could be adjusted to a given value between 0-100%) so the *expected* value of the fracture strain coincides with the results obtained from experiments or very detailed finite element simulations. Pay attention to the fact that the CA model can only be calibrated so the *expected* value of the fracture strain is as obtained from experiments. As long as the initial configuration of the voids are generated from a probability distribution, there will be a variance in the results. The only way to get rid of this variance, is to determine the initial configuration of the voids by some other means than a probability distribution, f.ex by a CT scan.

6.3.2 Parameter Study 2 - Periodically Distributed Voids

A periodic distribution of identical spherical voids shall now be simulated. A given configuration may for instance contain 500 voids, with an initial void vol-



(a) (b) (c)

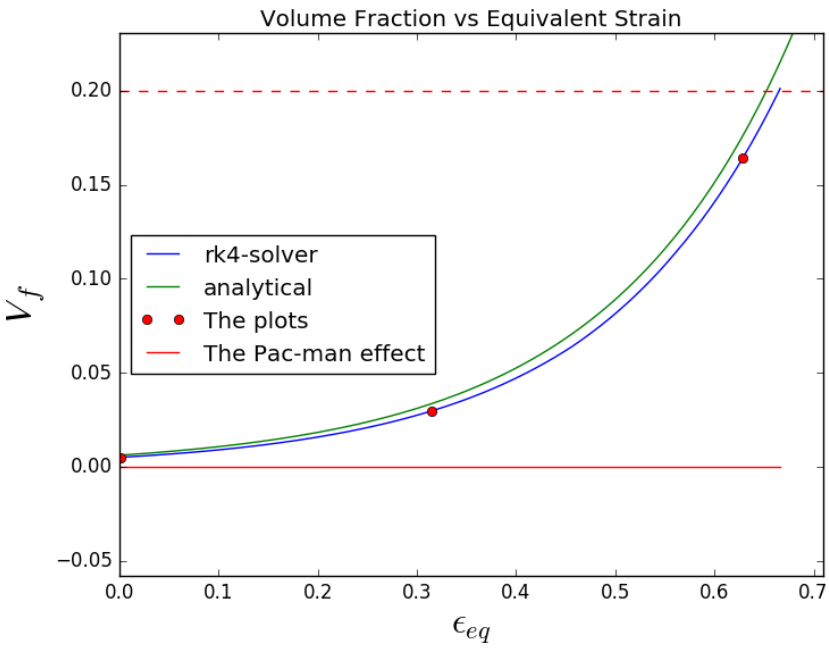
Figure 6.28: 50 voids, $Theta_L = 1$, with scaled "Pac-Man" effect to zero for each coalescence

ume fraction equal to $V_f^0 = 0.005$. Since the voids are all identical and spherical, this is the same as looking at only one void in a box, still with $V_f^0 = 0.005$. As it is shown in Fig 6.30, the large distribution of voids is equivalent to the modeling of only one void. Since all the voids undergoes the same deformations, coalescence will then occur when the void intersects its "own" box, shown in red. This scenario is *much* more ductile than the configurations generated from any probability distribution.

In Fig 6.31, 500 initially spherical voids following a uniform probability distribution are shown, with $\Theta_L = 1$ and $\sigma^* = 1$. The "Pac-Man" effect is allowed, which means that the calculated MVE ellipsoid aren't scaled. The results from this configuration shall now be compared to the case of periodically distributed initially spherical voids.

For the 500 randomly positioned initially spherical voids shown in Fig 6.31 and Fig 6.32, the critical void volume fraction is set to $V_f^{crit} = 0.2$, but this exact value for V_f^{crit} is of less importance: See the steep evolution of the void volume in Fig 6.32. If the critical void volume fraction was increased to f.ex $V_f^{crit} = 0.5$, there would only be a minimal difference in the fracture strain, if any.

Figure 6.29: Void volume evolution with the "Pac-Man" effect scaled to 0%



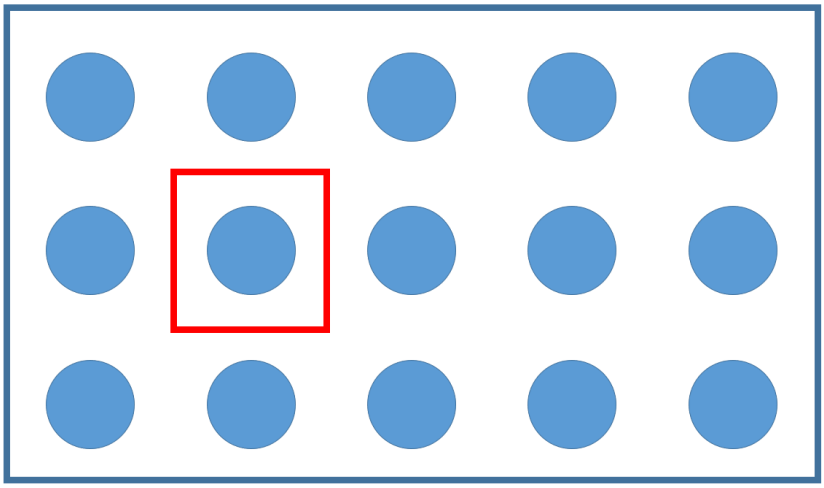
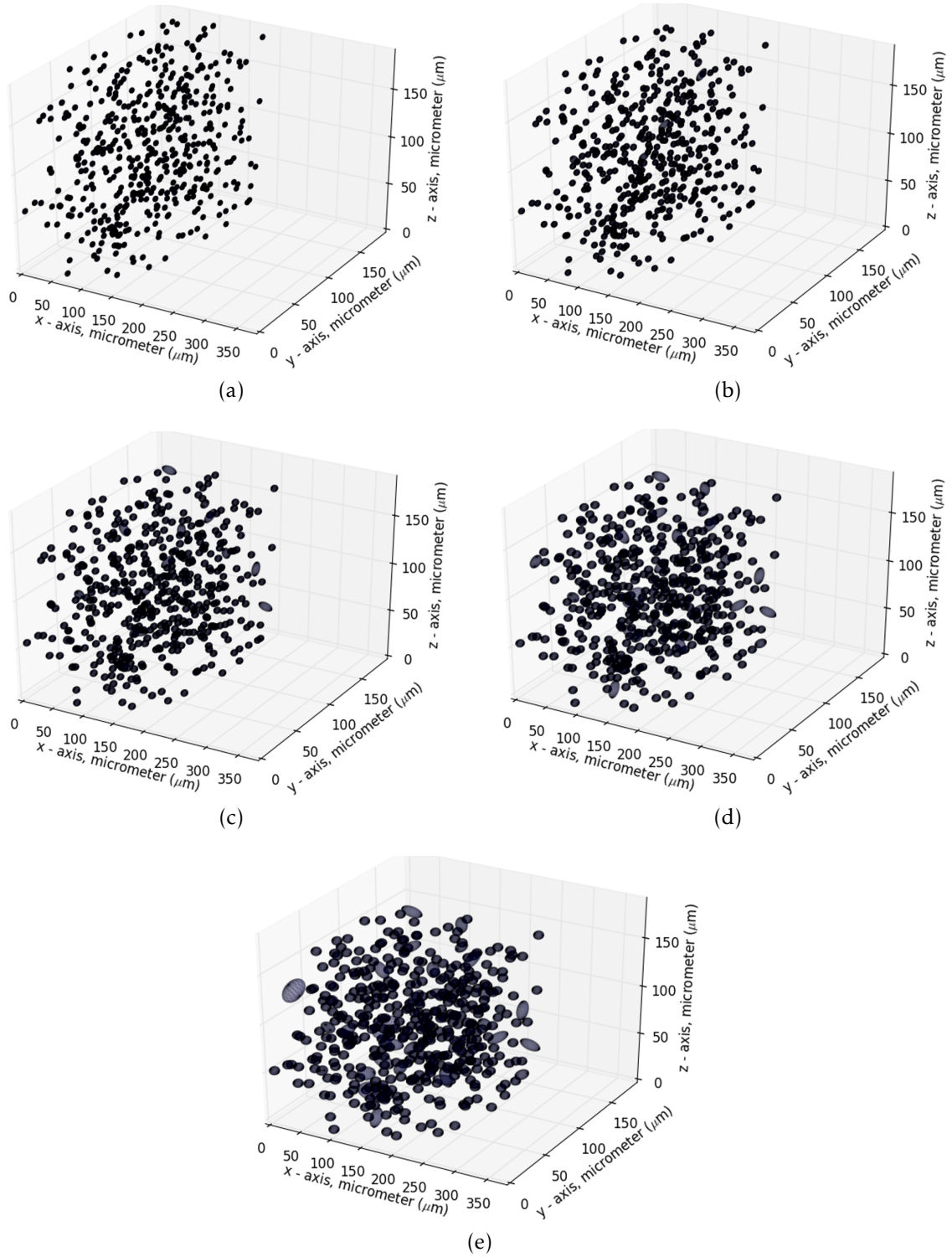


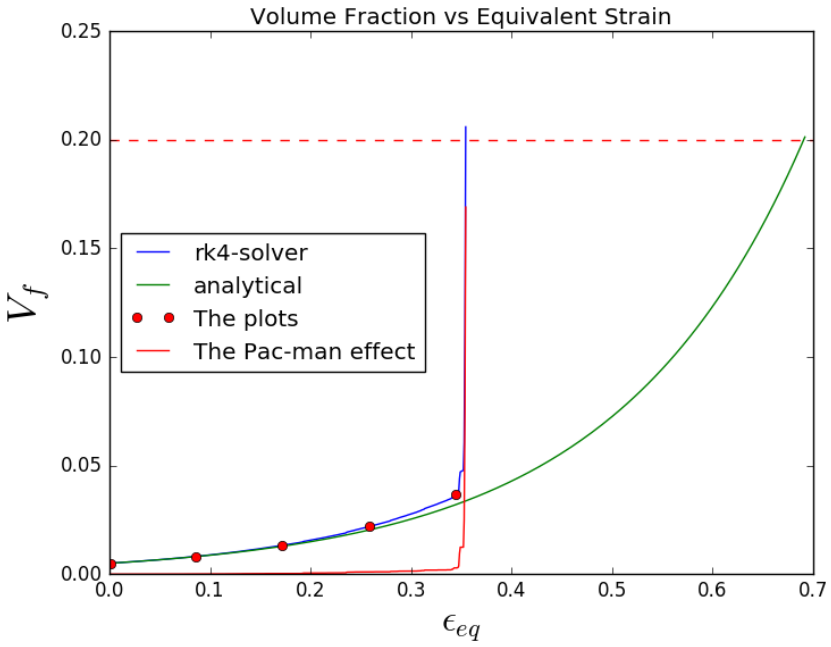
Figure 6.30: How to model a periodic distribution by only considering one void

For the periodically distributed voids, coalescence doesn't happen until the equivalent strain has reached the value $\varepsilon_{eq} = 0.819$, with a corresponding void volume fraction of $V_f = 0.394$ (see Fig 6.33. Imminent fracture will of course happen instantly after this point. If the critical void volume fraction had been defined as a higher value, f.ex $V_f^{crit} = 0.4$, there would have been even larger differences in the calculated ductility between the periodic and the random configuration. The fracture strain in the random positioned voids would probably not increase at all, while in the periodic case it would increase to $\varepsilon_{eq} \approx 0.82$. If V_f^{crit} was given a value so coalescence wasn't reached (i.e $V_f^{crit} \leq 0.394$), the calculated fracture strain will be a lower bound on the void volume because then only void



(a) (b) (c) (d) (e)

Figure 6.31: 500 voids, $\Theta_L = 1$. See Fig 6.32 for where in the deformation process the plots are taken from (the red dots)



(a)

Figure 6.32: 500 voids, $\Theta_L = 1$ and $\sigma^* = 1$

growth contributes to the void volume. Thus, the CA model is guaranteed to not yield a higher fracture strain (less conservative) with a random distribution than for a periodic distribution. These results are presented in Fig 6.34.

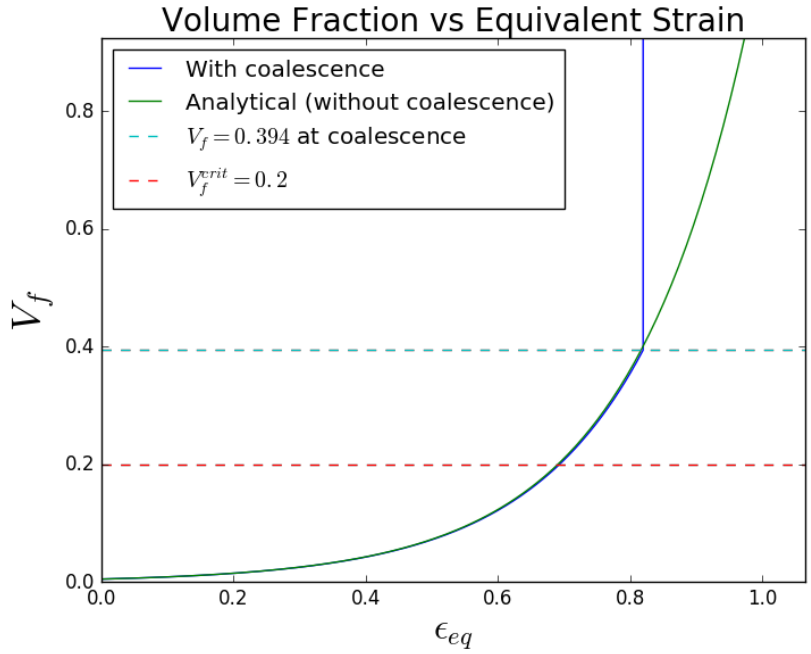


Figure 6.33: Void volume evolution with periodically distributed spherical voids

Figure 6.34: Fracture strain calculated from a random and a peridoc distribution of voids

Fracture defined at $V_f = 0.2$	
Random distribution	Periodic distribution
$\varepsilon_{eq}^{f} =$ 0.35	$\varepsilon_{eq}^{f} =$ 0.69

6.3.3 Summary of the Parameter Studies

- The main source of the increase in the void volume is seen to be the coalescence process, especially for larger number of voids, or higher stress triaxialities. From this, it is concluded that the coalescence of voids has the largest impact on the calculation of the fracture strain. It is essentially the largest voids that is involved in collisions with other voids, so the positioning of the largest voids is quite critical. If relatively large voids are given positions close to each other, the calculated fracture strain will be affected by decreasing correspondingly.

  The model is therefore very sensitive to the position and size of the voids. If the given position of the voids results in "clusters" (closely positioned voids), this is in fact very critical for the fracture strain.

- As the number of voids in consideration increases, the fracture strain decreases. It does seem like the CA model doesn't converge as long as an uniform random distribution is used to generate the initial configuration of voids.

- A periodic distribution of the positioned voids will always result in a much higher fracture strain. This is explained by the fact that a periodic distribution results in the largest possible initial distances between the voids, which postpones the coalescence significantly.

- The initial porosity has of course also a large effect on the predicted fracture strain ε_{eq}^{f}. Large voids grows faster than small voids, and if ductile fracture is assumed at the same critical void volume fraction, the result is of course lower ductility.

  The decrease in the fracture strain as the initial void volume fraction increases is obvious. See Fig 6.35 for the results obtained when 200 random voids are considered. In Fig 6.36, the analytical solutions of the void growth equation are shown. This is the same as not considering interactions between the voids, i.e for a "Pac-Man" effect set equal to 0%.

  It is concluded that the initial void volume fraction of course does have a *huge* impact on the ductility.

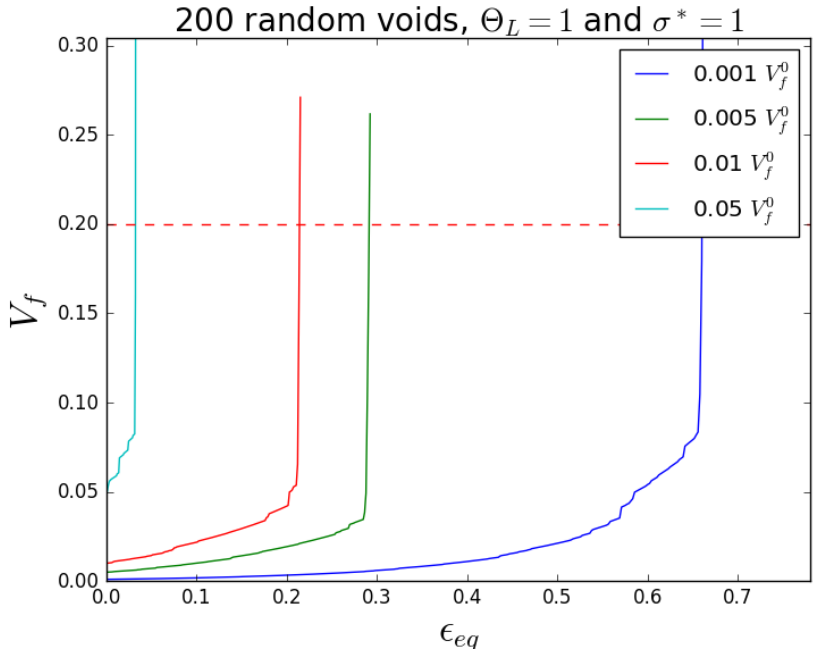


Figure 6.35: Sensitivity analysis, initial void volume fraction as parameter

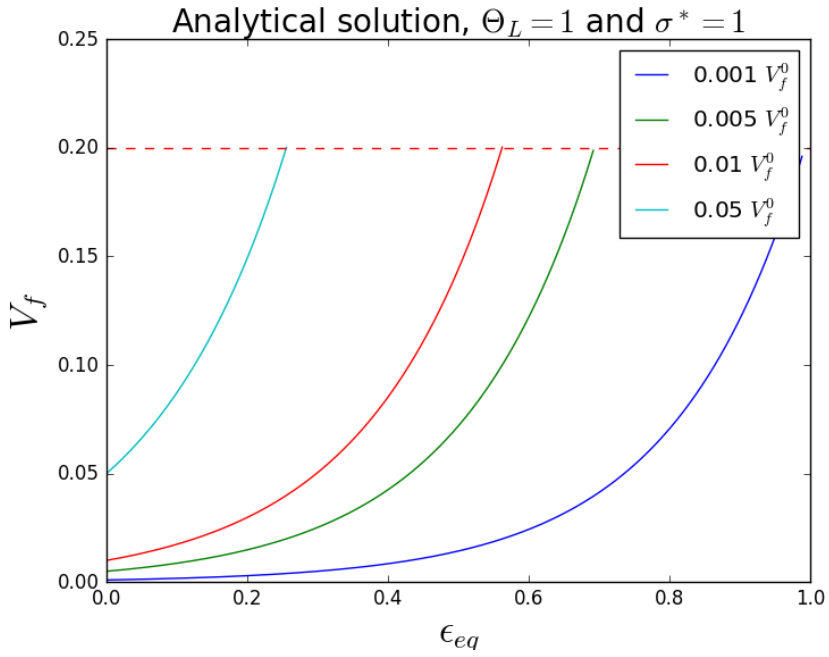


Figure 6.36: Sensitivity analysis, initial void volume fraction as parameter, only void growth (no coalescence)

6.3.4 Calculation of Fracture Locus

Before rounding up this thesis, the CA model will be used to calculate a fracture locus, also known as a "fracture surface". This is just a sample of fracture strains, calculated as the result of several analyses with different lode parameter Θ_L and

stress triaxiality σ^*. The fracture locus is expressed more mathematically as:

$$\varepsilon_{eq}^{f} = g(\Theta_L, \sigma^*) \tag{6.2}$$

The fracture strain may for instance be shown as a surface in 3D, or as points in a 2D plot. Both are shown in this section.

First, an expected fracture surface will be calculated based on 50 voids, where the MVE-ellipsoid was used with no scaling. The lode parameter took the following values

$$\Theta_L \in [-1,\ 0,\ 1] \tag{6.3}$$

and the stress triaxialities the values

$$\sigma^* \in [\frac{1}{3},\ \frac{2}{3},\ 1,\ 1.5,\ 2,\ 2.5,\ 3] \tag{6.4}$$

Each combination above was calculated 15 times, and the results shown in the figures are the expected value of the fracture strain ε_{eq}^{f} from these 15 calculations.

It was seen that for $\sigma^* = 1/3$, the voids seldom grew enough to coalesce. Even for $\sigma^* = 2/3$, coalescence was sometimes never reached. For such low values of σ^*, the voids will as shown before, increase a little, for so start decreasing in volume. The fracture strain will in these cases be false, since the the simulation didn't terminate because a critical void volume fraction was reached, but because the void volume fraction reached a lower limit. For these values of the triaxiality, shear fracture or maybe a mixed mode fracture between ductile fracture and shear fracture will probably occur. The value of the triaxiality for mixed mode fracture seems to be somewhere in the interval $\sigma^* \in [0.5, 1]$, according to [52]. Unfortunately, the CA model doesn't manage to capture this mixed mode fracture. It seems like the CA model is restricted to ductile fracture only, which is assumed to occur for approximately $\sigma^* \geq 1$. As the growth equation from Rice&Tracey was developed for higher triaxialities, this was as expected.

In Fig 6.37, the expected fracture strain for 50 random voids are shown. Pay attention to the values obtained for $\sigma^* = 1/3$ and $\sigma^* = 2/3$, which is marked within a red box. These results are as mentioned above sometimes obtained as the result of a terminating simulation because of the void volume approximately reaching zero. They are not valid results!

The same values plotted as a fracture locus is shown in Fig 6.38. Here, the results from $\sigma^* = 1/3$ and $\sigma^* = 2/3$ are included. It is quite difficult to see the dependence on the lode parameter Θ_L, which nonetheless are *very* small.

The fracture locus is also shown for the case when the Pac-Man effect is set to 0%. This case will just converge towards the analytical solution of the growth equation. To compute this, only one spherical void needs to be considered. For one void, the resulting fracture strain will never differ for the same values of Θ_L and σ^*, so the expected value is calculated based on only one simulation. The values for Θ_L and σ^* used this time is:

$$\Theta_L \in [-1, -\frac{2}{3}, -0.5, -\frac{1}{3}, 0, \frac{1}{3}, 0.5, \frac{2}{3}, 1] \tag{6.5}$$

and

$$\sigma^* \in [\frac{1}{3}, 0.5, \frac{2}{3}, 1.0, 1.25, 1.5, 1.75, 2.0, 2.25, 2.5, 3.0] \tag{6.6}$$

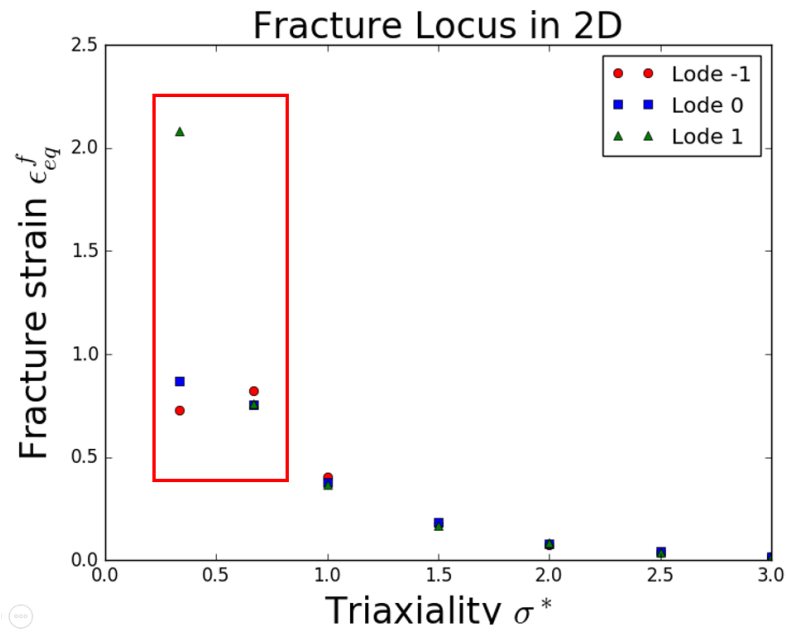


Figure 6.37: Expected fracture strain by the automaton model, 100% Pac-Man effect, $\sigma^* = 1/3$ and $\sigma^* = 2/3$does not give valid results

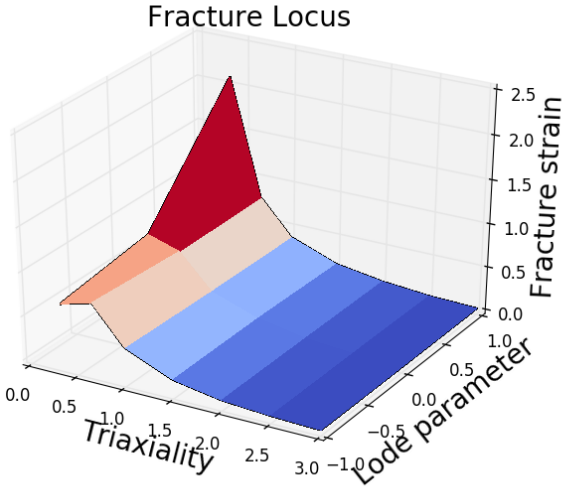


Figure 6.38: Expected fracture locus by the automaton model, 100% Pac-Man effect

For the plot in 2D (Fig 6.39), only values for $\Theta_L = 1$, $\Theta_L = 0$ and $\Theta_L = -1$ are shown. The red box are again used to mark the invalid results that were the result of termination. For the plot of the fracture surface (Fig 6.41), all the calculated values of Θ_L and σ^* are shown.

Fig 6.3.4 might be difficult to envision, but it is just the fracture locus seen from an angle perpendicular to what is shown in Fig 6.39. The goal with this plot was to show the fracture strains dependence on the lode parameter. There is a little dependence when $\sigma^* = 1.0$, but for higher triaxialities the differences

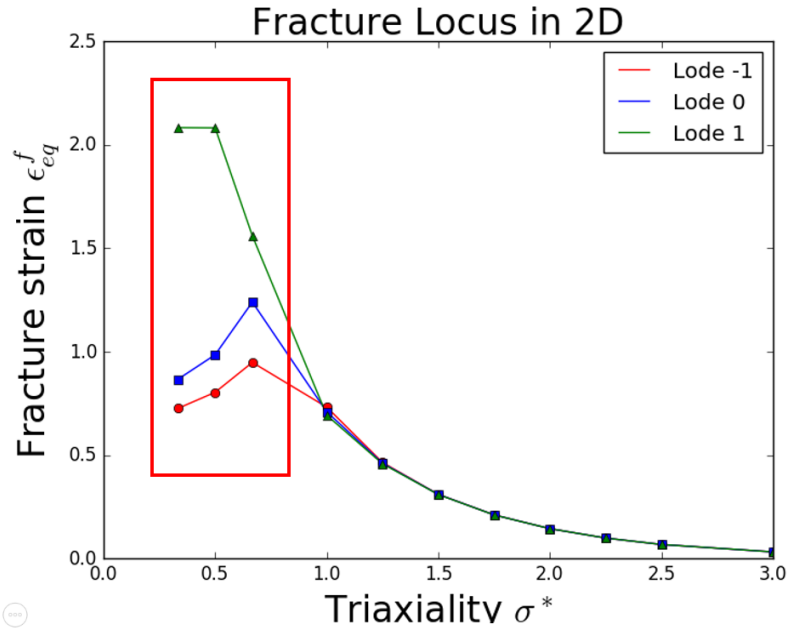


Figure 6.39: Expected fracture strain by the Rice&Tracey growth equation

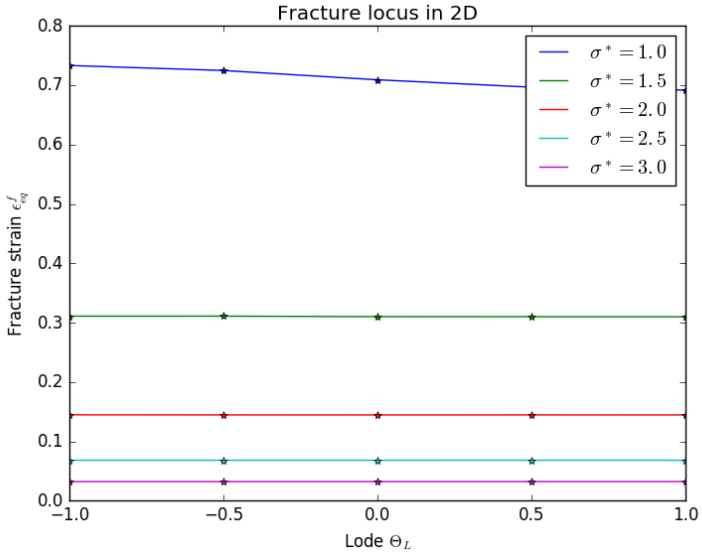


Figure 6.40: Expected fracture strain by the Rice&Tracey growth equation

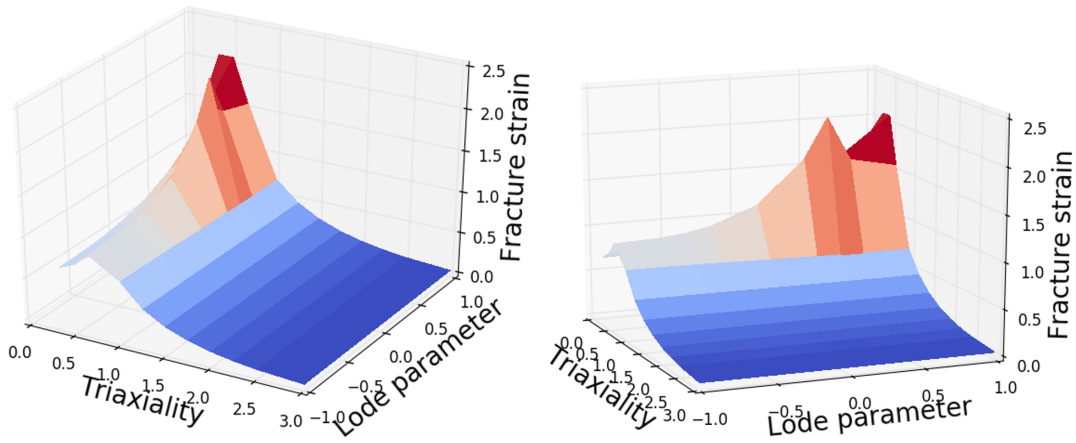


Figure 6.41: Expected fracture surface by the Rice&Tracey growth equation

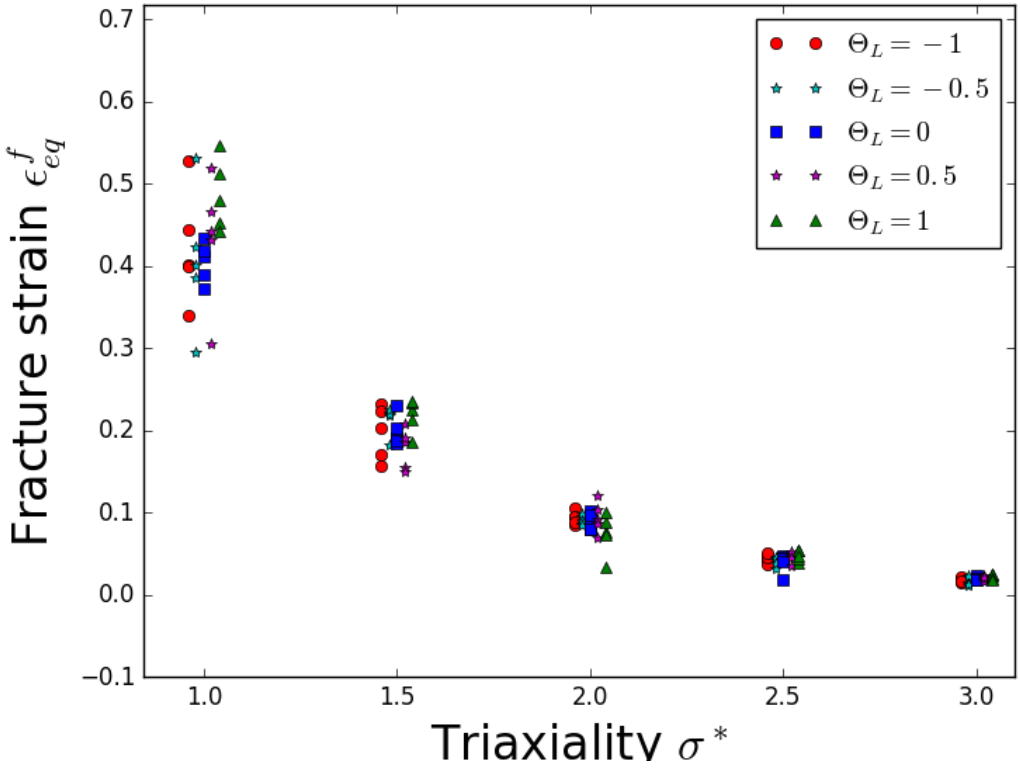


Figure 6.42: Fracture strain from 5 analyses for 30 voids

aren't visible for the naked eye at all. The corresponding fracture surface in 3D is shown in Fig 6.41.

In Fig 6.42, the variance in the results obtained with 30 random voids, $\sigma^* = 1$ and $\Theta_L = 1$ are shown for the valid results only ($\sigma^* \geq 1$). The expected fracture strain are calculated based on n results like this. It is evident from the figure how there is much larger variance for the lower values of the triaxiality than the higher values of the triaxiality. This is because the voids grows much faster for higher values of the triaxiality σ^*, so fracture will nevertheless occur quite early on in the deformation process. The reader must be aware that the points shown in each "group" in Fig 6.42, are all calculated for the same triaxiality. The plot is supposed to better show the variance in the result, but it may be misleading. The triaxialities used are 1.0, 1.5, 2.0, 2.5 and 3.0, as described in Eq (6.6).

7 Extension of Modeling Framework

As the development of the model herein was limited by time, some ideas remain unexplored. This chapter presents some concrete suggestions to future activities related to the cellular automaton model.

Probability Distribution for the Position of the Voids

How to give the voids an initial position is seen as the main problem with the CA model as it is presented in this thesis. The problem is avoided by using CT scanning of the material to obtain all the voids' position and size, but this is of restricted practical use. The ultimate goal is seen as to calibrate a probability distribution, so the results converges towards a value for the fracture strain with an increasing amount of voids, and at the same time, the variance goes towards zero. This section concludes that this desired probability distribution is *not* a uniform distribution. With uniformly distributed void position and a "Pac-Man" effect above 0%, the fracture strain is seen to steadily decrease as the number of voids in the model are increased. The distribution need to take into account the already positioned voids, which means a more sophisticated probability distribution such as for instance Markov-chains seems like a much better choice.

Nucleation of Voids

As mentioned in Chapter 2, void nucleation is actually a discontinuous process of successions of discrete nucleation events, where for instance the particle size can be a dominant feature for when the voids nucleate during the deformation process. Therefore, to model nucleation, each particle may be assigned an associated critical principal stress value. When this value is reached for the particle, it nucleates into a microvoid. Instead of giving position to a bunch of microvoids, the CA model would in this case instead generate positions for the particles. This way, a more accurate description of the ductile fracture process would be achieved.

Differently Defined Coalescence

Another possibility for the purpose of trying to copy the behavior of ductile fracture shown in experiments, would be to implement the coalescence differently. When the distance between the voids in question is below some critical value, possibly calculated based on the void's size (or based on some other argument), a new void could be created between them; which would represent the crack that links the two voids together, see Fig 7.1.

(a) Current coalescence (b) Alternative coalescence

Figure 7.1: How coalescence are implemented, and an alternative procedure

This way, cracks will be able to link between voids instead of just turning the colliding voids into one large "ball". The links that are created between the voids will not intersect any other voids, because if there had been any voids between them, there would have been created a link between these first. This linkage between the voids can spread throughout the model, to better represent the coalescence process. It is important to subtract the volume of the overlapping voids, i.e no volume should be counted twice. A possible strategy would be to calculate a convex hull of the intersecting area. The option to remove the part of the void's volume that is located outside the box's dimensions has already been implemented in this thesis, and this overlapping area between voids can be found by the same strategy, although it is of course a bit more complex mathematical problem.

Another possible improvement by implementing this strategy is that the coalescence chain-reaction that happens quite fast after the first voids intersects in the current model, would probably be somewhat restrained, as this new smaller void are quite unlikely to coalesce with other voids in the model, since it is created *inbetween* two neighbors. These links between voids is much closer to physical results than a MVE ellipsoid.

It may even be expedient to define the fracture criterion differently: Instead of defining a critical void volume fraction, fracture may be defined to occur when n number of voids are linked together in the model, or maybe define local fracture to occur when a given void is linked directly to m other voids. The point is that these linkages between voids gives the user of the program *a lot* of information about what happens in the model. These linkages are kind of creating a graph structure during the deformation, by connecting the nodes (voids) with edges (links) between them. But, no matter how the reader chooses to imagine this, it must be said that it seems like a very interesting approach.

This differently defined coalescence, combined with a better probability distribution and implementation of particles that may nucleate instead of already

nucleated voids, seems like the three best options for getting the model to converge towards the physical results obtained in experiments, without the need for calibration!

Other Possible Improvements

Other parts of the program with improvement potential are more in a computer programming sense of improvement. Each void grow independent to all the other voids (there are no memory dependences between them as they grow), until collisions are controlled at the end of the step. This is an obvious case where parallel programming would give a speed boost. Also, to find each void's neighbors, a divide&conquer algorithmic implementation is known to result in a faster asymptotic running time, but not enough time was find during the work with this model to implement it. This is the only algorithm in this thesis that performs below the desired performance of the author.

8 Concluding Remarks

This thesis presents the foundation, implementation, verification and validation of a cellular automaton model for ductile fracture in alloys. It is quite similar to the work presented by Hannard et al. [27], at least the concept of the CA model. By taking into account void growth and coalescence, a method for calculating the ductility of alloys was established.

Chapter 1 starts by presenting ductile fracture observations, specifically that it results from nucleation, growth and coalescence of microvoids, and mentions several research results obtained in this field primarily during the last 50 years. The goal of the thesis was presented; being to understand and model the effect of micro structure heterogeneities on damage accumulation in metallic alloys.

In Chapter 2, ductile fracture observations were presented. The main reasons for ductile fracture were also discussed in depth.

The theory considered necessary in order to follow the procedures implemented in the model, was presented in Chapter 3. The mentioned topics are from mathematics, material mechanics and programming, since these three areas creates the foundation for this thesis.

In Chapter 4, the most important procedures in the model was discussed in depth, where the mathematical basis was presented first, followed by flowcharts of the whole program, and then pseudo-code. The source code was implemented around the algorithms and ideas presented here. A thorough understanding of the model was hopefully established in this chapter.

Chapter 5 is about the verification of the model. The recurring theme here were: "Are the equations are solved correctly?". The numerical solutions are compared to the analytical, and the convergence is discussed. There were pointed out differences in the results from the explicit solver schemes. The asymptotic running time of the program was presented through a pseudo-code formulation, were the running time of each commando in the main part of the program was stated. Choices taken to implement a fast program was pointed out.

As the model was working properly according to the theoretical foundation, a validation of the model was done in Chapter 6. The void growth and rotation was compared to corresponding results from Abaqus, for a wide range of situations. Shortcomings of assumptions made in the program, and the behavior of especially the MVE-ellipsoid algorithm was discussed.

The finished model was tested at a wide range of cases, from parameter studies to establishing a fracture locus. A sensitivity analysis was also summarized. Unfortunately, the use of a uniform distribution of the void positions was shown to not be refined enough. As the number of voids used in the model increased, the lower the expected calculated ductility. This was concluded to be the result of a higher probability of unfavorably positioned voids as the void number increased, combined with the fact that the MVE-ellipsoid may remove too much matrix at each coalescence.

The main focus in the chapter "Extension of Modeling Framework" was therefore to develop a probability distribution that takes into account the current state of void positions and sizes, and give the next void a position accordingly. Also, the coalescence could be defined differently, by creating linkages between voids. Last, but not least, nucleation should also be implemented to better model the ductile fracture process, without the very conservative assumption made that every void is nucleated before the deformation starts.

References

[1] Lynn S. Beedle. *Why plastic design*. 1956. URL: http://digital.lib.lehigh.edu/fritz/pdf/205_52.pdf (visited on 06/01/2016).

[2] Per Kr. Larsen. *Dimensjonering av staalkonstruksjoner - 2nd Edition*. Tapir Akademisk Forlag, 2010.

[3] *Fracture mechanics - Wikipedia*. 2016. URL: https://en.wikipedia.org/wiki/Fracture_mechanics (visited on 05/01/2016).

[4] A.A. Benzerga and J.-B. Leblond. "Ductile fracture by void growth to coalescence". In: *Adv App Mech*, *Adv App Mech*, *Vol. 44* (2010), 169–305.

[5] T.L. Anderson. *Fracture Mechanics - Third Edition*. Taylor & Francis, 2005.

[6] K. Tanaka, T. Mori, and T. Nakamura. "Cavity formation at the interface of a spherical inclusion in a plastically deformed matrix". In: *Philos. Mag*, *21* (1970), 267–279.

[7] A. Argon, J. Im, and R. Safoglu. "Cavity formation from inclusions in ductile fracture". In: *Metall. Mater Trans. A* (1975), 825–837.

[8] C. F. Tipper. "The fracture of metals". In: *Metallurgia* (1949), pp. 33–133.

[9] A.Needleman, V.Tvergaard, and J.W.Hutchinson. *Void Growth in Plastic Solids*. Springer, 1992, pp. 145–178.

[10] F.A. McClintock. "A criterion for ductile fracture by the growth of holes". In: *J. Appl. Mech* (1968), 363–371.

[11] J.R. Rice and D.M. Tracey. "On the ductile enlargement of voids in triaxial stress fields". In: *J. Mech. Phys. Solids* (1969), 201–217.

[12] P. Thomason. *Ductile Fracture of Metals*. Pergamon Press, 1990.

[13] A. Pineau and T. Pardoen. "Failure of Metals". In: *Comprehensive Structural Integrity, Pergamon, Oxford* (2007), 684–797.

[14] T. Pardoen et al. "Multiscale modeling of ductile failure in metallic alloys". In: *C R. Phys*, *11* (2010), 326–345.

[15] O.S Hopperstad. "A brief review of Thomason's model for ductile fracture of metals". In: *Technical report, NTNU* (2000).

[16] F. Beremin. "Cavity formation from inclusions in ductile fracture of A508 steel". In: *Metall. Mater Trans. A* (1981), 723–731.

[17] D. Lassance et al. "Micromechanics of room and high temperature fracture in 6xxx Al alloys". In: *Prog. Mater Sci* (2007), 62–129.

[18] A. Needleman and A. Kushner. "An analysis of void distribution effects on plastic flow in porous solids". In: *Eur. J. Mech. A-Solids* (1990), 193–206.

[19] Y. Huang. "The role of nonuniform particle distribution in plastic flow localization". In: *Mech. Mater* (1993), 265–279.

[20] C. Thomson et al. "Void coalescence within periodic clusters of particles". In: *J. Mech. Phys. Solids* (2003), 127–146.

[21] T. Pardoen and J.W. Hutchinson. "An extended model for void growth and coalescence". In: *J. Mech. Phys. Solids* (2000), 2467–2512.

[22] n.d. *Necking*. 2016. URL: http://www.engineeringarchives.com/les_mom_necking.html (visited on 05/27/2016).

[23] *Steel*. 2016. URL: https://en.wikipedia.org/wiki/Steel (visited on 05/05/2016).

[24] *Steel*. 2016. URL: https://en.wikipedia.org/wiki/Alloy (visited on 05/05/2016).

[25] n.d. photograph. *SEM fractorgaph*. URL: http://products.asminternational.org/fach/data\-/fullDisplay.do?database=faco (visited on 05/16/26).

[26] n.d. photograph. *Cup and Cone Photography*. URL: http://web.vtc.edu/mt/114SUM01/TTsum01/Spec19/CupandCone.jpg (visited on 05/23/2016).

[27] F.Hannard et al. "Characterization and micromechanical modelling of microstructural heterogeneity effects on ductile fracture of 6xxx aluminium alloys". In: *Acta Materialia* (2015).

[28] A.L. Gurson. "Continuum theory of ductile rupture by void nucleation and growth,1. Yield criteria and flow rules for porous ductile media". In: *J. Eng. Mater. Technol.99* (1977), pp. 2–15.

[29] K. Nahshon and J.W. Hutchinson. "Modification of the Gurson Model for shear failure". In: *European Journal of Mechanics A/Solids 27* (2008), pp. 1–17.

[30] n.d. photograph. *Alloy Photography*. URL: https://en.wikipedia.org/wiki/Alloy (visited on 05/27/2016).

[31] *Steel Uncer Microscope*. URL: http://inside.mines.edu/~sliu/courses/mtgn475-477/inclusions-alignment.jpg (visited on 05/28/2016).

[32] Erwin Kreyszig. *Advanced engineering mathematics - 10^{th} Edition*. Wiley, 2011.

[33] n.d. *Trapezoidal Rule*. URL: http://www.bragitoff.com/wp-content/uploads/2015/08/TrapezoidRule1.png (visited on 05/27/2016).

[34] n.d. *Projection*. URL: https://en.wikibooks.org/wiki/Linear_Algebra/Projection_Onto_a_Subspace (visited on 05/28/2016).

[35] Dimitri P. Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods*. 1996. URL: http://www.mit.edu/~dimitrib/Constrained-Opt.pdf (visited on 03/11/2016).

[36] Fridtjov Irgens. *Continuum Mechanics*. Springer, 2008.

[37] O.S Hopperstad and T. Børvik. *TKT4135 Materials mechanics - Lecture Notes, Part 1*. Department of Structural Engineering - NTNU, 2015.

[38] O.S Hopperstad and T. Børvik. *KT8306 Materials mechanics - Lecture Notes, Part 2*. Department of Structural Engineering - NTNU, 2015.

[39] *LS-DYNA Online Documentation*. 2016. URL: http://www.dynasupport.com/tutorial/computational-plasticity/the-equations-for-isotropic-von-mises-plasticity (visited on 03/14/2016).

[40] *Continuum Mechanics Picture 1, n.d.* URL: https://en.wikipedia.org/wiki/Continuum_mechanics (visited on 05/27/2016).

[41] *Continuum Mechanics Picture 2, n.d.* URL: https://en.wikipedia.org/wiki/Continuum_mechanics (visited on 05/27/2016).

[42] Magnus L. Hetland. *Beginning Python*. Springer, 2005.

[43] Thomas H. Cormen, Charles E. Leiserson, and Charles E. Leiserson. *Introduction to Algorithms*. The MIT Press; 3^{rd} Edition, 2009.

[44] Magnus L. Hetland. *Python Algorithms*. Springer, 2010.

[45] n.d. *Class and Object*. URL: http://www.4guysfromrolla.com/webtech/chapters/BuildASPNETWebSite/ch02.2.shtml (visited on 05/23/2016).

[46] *Asymptotic Notations, Arv Kunday*. URL: http://www.kunday.com/post/290464306/asymptotic-notations (visited on 05/27/2016).

[47] Stephen B. Pope. *Algorithms for Ellipsoids*. 2008. URL: https://tcg.mae.cornell.edu/pubs/Pope_FDA_08.pdf (visited on 04/12/2016).

[48] Leonid G. Khachiyan. "Rounding of Polytopes in the Real Number". In: *Model of Computation. Mathematics of Operations Research Vol. 21* (1996), pp. 307–320.

[49] *Port MATLAB bounding ellipsoid code to Python*. 2016. URL: http://stackoverflow.com/questions/14016898/port-matlab-bounding-ellipsoid-code-to-python (visited on 03/19/2016).

[50] *MINIMUM VOLUME ENCLOSING ELLIPSOIDS*. 2005. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.116.7691\&rep=rep1\&type=pdf (visited on 02/16/2016).

[51] *Minimum volume enclosing ellipsoid*. 2006. URL: https://www.researchgate.net/profile/Nima_Moshtagh/publication/254980367_MINIMUM_VOLUME_ENCLOSING_ELLIPSOIDS/links/54aab5260cf25c4c472f487a.pdf (visited on 02/16/2016).

[52] Yanshan Loua, Jeong Whan Yoonb, and Hoon Huhc. "Modeling of shear ductile fracture considering a changeable cut-off value for stress triaxiality". In: *International Journal of Plasticity Volume 54* (2014), 56–80.

A

Source Code for the Cellular Automaton Model

```
from __future__ import division
import numpy as np
import matplotlib.pyplot as plt
from sympy import symbols,diff,solve,exp
import random
from mpl_toolkits.mplot3d import Axes3D
import numpy.linalg as la
from scipy.spatial import ConvexHull
import math as m

################################################
#           CELLULAR AUTOMATON MODEL           #
################################################

#-------------------------------------------------------------
#                User Input
#-------------------------------------------------------------
# User decides number of voids in the simulation:
num_voids=50

# User specifies orientation of the voids; the choices are
# 'Rand', 'Spherical', 'Skra', 'Langs', 'Spherical_error',
# 'One_with_angle', and 'Two_with_coalescence':
orientation='Rand'

# The following are only used if the orientation is 'One_with_angle':
angle_for_1_void = 45.
ratio_for_1_void = 1.

# The following are only used if orientation is 'Two_with_coalescence':
Two_with_coalescence_orientation = 'Spherical'
# Same choices as above, except 'Rand' is in the xz-plane only.

# User specifies initial volume percentage microvoids:
Volume_fraction=0.005
# 0.005 is a representative value for most industrial alloys...

# The stress-triaxiality and Lode may be defined as constant or
# linearly varying parameters, depending on the number of
# points in the assigned list of values:
Stress_triaxiality = [1.]
```

```
phi=[1.]

# The infamous 'Pac-Man' effect may be adjusted here by a value
# between 0 and 1, 0 gives MVEE, 1 gives no Pac-Man effect at all:
Reduced_Pac_Man_Effect = 0.

# How large the term 'dt*L_star*g' can be compared to 'g_original',
# to avoid that the voids shrinks to negative semi-axes -> instability!
g_vectors_tol = 0.1
# When using many timesteps, a smaller value here is OK

# Number of timesteps, higher stress triaxiality needs more timesteps:
N_timesteps = 700

# Chose a numerical solver among 'euler', 'heun' and 'rk4':
Which_explicit_integration_scheme = 'rk4'

# Number of plots the user wants:
number_of_plots = 3

# Set the stretch ratio for the x-direction larger or smaller than 1;
# the program terminates when the critical
# void volume fraction is reached nonetheless:
lamba_x = 2

# Critical void volume fraction:
Upper_Void_Volume_Limit = 0.2

# Should the volume outside the box be removed? User decides:
Remove_Volume_Outside_Box = False



#-----------------------------------------------------------------
#               Preliminary setup
#-----------------------------------------------------------------

# Calculate the constant used in the velocity gradient L
constant_a = np.log(lamba_x)

# Smallest possible semi-axis, micrometer
d_min = 5
# Largest possible semi-axis, micrometer
d_max = 50

t_start = 0.

n_ellipsoid_samplepoints=6
# Uses the square of the value points from the ellipsoids
# in the algorithm MVEE (6**2 = 36 points...)

N_ellipsoid_plotting_points=20
#For the plotting, 20 results in 400 points from the ellipsoid

#The transparency for the ellipsoids, 1 is max, 0 invisible:
alpha_plot=0.1
```

```
# Constants used in Rice&Tracey growth equation
E=2.0/3
alpha=0.427

if len(Stress_triaxiality) == 1 and len(phi) == 1:
    # If the length is equal to 1, the values are constant during the
    # deformation! The values calculated here does then not need
    # to be updated during the analysis:
    # The other constants in the velocity gradient tensor L is
    # calculated:
    constant_b = -phi[0]*constant_a*2/(3+phi[0])
    constant_c = constant_a*(phi[0] - 3)/(phi[0] + 3)

    # Gather them in the same array:
    constant=np.array([constant_a,constant_b,constant_c])
    L_diag=np.array([constant[0],constant[1],constant[2]])

    # The velocity gradient tensor L is calculated:
    L=np.diag(L_diag)

    # L is the same for all the voids, it is therefore defined
    # as a global variable

    # Simplifies the calculation of the Rice&Tracey growth
    # equation by defining:
    epsilon_j_squared = L_diag[0]**2 + L_diag[1]**2 + L_diag[2]**2

    # The strain rate is equal to L in the case of uniform extension:
    epsilon_rate=L_diag



#-----------------------------------------------------------------
#               Class 'Void' definition
#-----------------------------------------------------------------

class Void():
    def __init__(self,T,D,T_inv,C,name):
        """
        The constructor of the class, these parameters
        must be given upon creation of an instance.
        """
        self.T=T        # Eigenvectors, i.e principal directions of A
        self.D=D          # Eigenvalues, i.e principal values of A
        self.T_inv=T_inv        # The inverse of T
        self.C=C                # The position of the void's center
        self.C_origin=C          # The original position,
                                 # to calculate the translation.
        self.neighboors=set()   # A set of the void's neighbors

        # Remember which neighbors that have been checked:
        self.checked_neighboors=[]
        # Unique name, to avoid overwriting existing voids:
        self.name=name

        # Only need to rearrange the eigenvectors once:
        self.Checked_Eigenvectors_Counter = 0
```

```
        # Upon creation, some methods are called to get the rest of the
        # values immediately, through UpdateVoid():
        self.Update_Void()

    def Volume(self):
        """
        Returns the volume of the void
        """
        volume = 4.0/3*np.pi*self.a*self.b*self.c
        return volume

    def calcA(self):
        """
        Calculates the A matrix of the ellipsoid (void)
        """
        self.A=self.T*self.D*self.T_inv
        self.A=np.matrix(self.A,dtype=float)
        return

    def COEFF(self):
        """
        Calculates the semi-axes's lengths
        """
        self.coefficients=np.array([np.sqrt(1./self.D[i,i]) for i in
            xrange(3)])
        self.a=self.coefficients[0]
        self.b=self.coefficients[1]
        self.c=self.coefficients[2]
        return self.coefficients

    def Global_Semi_Axes(self):
        """
        To find the 'equivalent' ellipsoid, where the intersections in
        the the global directions are calculated.
        """
        self.a_glob=np.sqrt(1./self.A[0,0])
        self.b_glob=np.sqrt(1./self.A[1,1])
        self.c_glob=np.sqrt(1./self.A[2,2])
        return np.array([self.a_glob,self.b_glob,self.c_glob],dtype=
            float)

    def RiceTracey(self,global_coeff):
        """
        Returns the growth rates found from the Rice&Tracey
        growth equation:
        """
        a_glob,b_glob,c_glob = global_coeff
        R_mean = (a_glob + b_glob + c_glob)/3.0
        rad_rate=RT(R_mean)
        return rad_rate

    def rk4(self,global_coeff):
        """Runge-Kutta 4. order explicit scheme:"""
        dt2=dt/2.0
        # predictor step 1:
        k1=np.asarray(self.RiceTracey(global_coeff))
        # predictor step 2:
        k2=np.asarray(self.RiceTracey(global_coeff+k1*dt2))
```

```
        # predictor step 3:
        k3=np.asarray(self.RiceTracey(global_coeff+dt2*k2))
        # predictor step 4:
        k4=np.asarray(self.RiceTracey(global_coeff+dt*k3))
        # corrector step:
        global_coeff=global_coeff+dt/6.0*(k1 + 2.0*k2 + 2.0*k3 + k4)
        return global_coeff

    def euler(self,global_coeff):
        """Euler 1. order explicit scheme"""
        # predictor step:
        k1=np.asarray(self.RiceTracey(global_coeff))
        # corrector step:
        global_coeff=global_coeff+dt*k1
        return global_coeff

    def heun(self,global_coeff):
        """Heun 2. order explicit scheme"""
        # predictor step 1:
        k1=np.asarray(self.RiceTracey(global_coeff))
        # predictor step 2:
        k2=np.asarray(self.RiceTracey(global_coeff+k1*dt))
        # corrector step:
        global_coeff=global_coeff+dt/2.*(k1 + k2)
        return global_coeff

    def Rotate_and_grow(self,time):
        global_coeff = self.Global_Semi_Axes()
        if Which_explicit_integration_scheme == 'rk4':
            rad_glob=self.rk4(global_coeff)
        elif Which_explicit_integration_scheme == 'heun':
            rad_glob=self.heun(global_coeff)
        elif Which_explicit_integration_scheme == 'euler':
            rad_glob=self.euler(global_coeff)
        rad_rate=self.RiceTracey(global_coeff)
        # The local velocity gradient tensor is unique for each void,
        # and is defined as:
        L_star_diag=np.array([rad_rate[0]/rad_glob[0],rad_rate[1]/
            rad_glob[1],
                              rad_rate[2]/rad_glob[2]])
        L_star=np.diag(L_star_diag)

        # To calculate the void growth, the following procedyre is
            followed:
        coefficients=self.COEFF()
        tempo1 = self.a
        tempo2 = self.b
        tempo3 = self.c

        temp1 = np.array(self.T[:,0])
        temp2 = np.array(self.T[:,1])
        temp3 = np.array(self.T[:,2])

        g1=temp1*tempo1
        g2=temp2*tempo2
        g3=temp3*tempo3
        g1=np.array(g1[:,0],dtype=float)
        g2=np.array(g2[:,0],dtype=float)
```

```
g3=np.array(g3[:,0],dtype=float)

# Rearrange the eigenvectors so the voids rotates in the
#  correct direction, only needed once:
if self.Checked_Eigenvectors_Counter == 0:
    Checked_Eigenvectors_Counter = 1
# This ensures it is not checked again!
if tempo1 >= tempo2 and tempo1 >= tempo3:
        pass    # The order of the eigenvectors was correct!
elif tempo2 >= tempo1 and tempo2 >= tempo3:
 #Must swap g1 and g2, and self.a and self.b:
    temp_value = self.a
    self.a=self.b
    self.b=temp_value
    temp_vector = g1
    g1 = g2
    g2 = temp_vector
elif tempo3 >= tempo1 and tempo3 >= tempo2:
    #Must swap g1 and g3, and self.a and self.c:
    temp_value = self.a
    self.a=self.c
    self.c=temp_value
    temp_vector = g1
    g1 = g3
    g3 = temp_vector

# The voids are stopped from oscillating under
# low stress triaxialities:
if np.linalg.norm(dt*np.dot(L_star,g1)) >
                g_vectors_tol*np.linalg.norm(g1):
    magn1 = np.linalg.norm(g1)
    g1 = g1 + dt*np.dot(L_star,g1) * g_vectors_tol
else:
    g1=g1+dt*np.dot(L_star,g1)
    magn1 = np.linalg.norm(g1)
if np.linalg.norm(dt*np.dot(L_star,g2)) >
                g_vectors_tol*np.linalg.norm(g2):
    magn2 = np.linalg.norm(g2)
    g2 = g2 + dt*np.dot(L_star,g2) * g_vectors_tol
else:
    g2=g2+dt*np.dot(L_star,g2)
    magn2 = np.linalg.norm(g2)
if np.linalg.norm(dt*np.dot(L_star,g3)) >
                g_vectors_tol*np.linalg.norm(g3):
    magn3 = np.linalg.norm(g3)
    g3 = g3 + dt*np.dot(L_star,g3) * g_vectors_tol
else:
    g3=g3+dt*np.dot(L_star,g3)
    magn3 = np.linalg.norm(g3)

# The principal axes of the void has now changed directions
# and magnitudes, and must be adjusted into an orthonormal
# basis once again by a Gauss-Seidel Decomposition:
tempor = np.matrix([[g1[0],g2[0],g3[0]],[g1[1],g2[1],
                g3[1]],[g1[2],g2[2],g3[2]]])
#Gauss Seidel decomposition:
q, r = np.linalg.qr(tempor)
# This process may result in det(tempor) = +1 or -1, so should
```

```
        # control that the directions hasn't changed 180 degrees:
        if g1[0] > 0:
            if q[0,0] < 0:
                q = q * (-1)
        if g1[0] < 0:
            if q[0,0] > 0:
                q = q * (-1)

        # The void has now rotated and grown, so its attributes
        # should be updated:
        self.D=np.diag([1./magn1**2,1./magn2**2,1./magn3**2])
        self.D=np.matrix(self.D)
        self.T=q
        self.T_inv=np.transpose(self.T)
        self.Update_Void()

        # The void also need to translate:
        New_C=np.array([self.C_origin[0]*lamba(time)[0],
                        self.C_origin[1]*lamba(time)[1],
                        self.C_origin[2]*lamba(time)[2]])
        self.C=New_C
        return

    def Update_Void(self):
        """Just to gather the updating methods in one method:"""
        self.calcA()
        self.COEFF()
        self.Global_Semi_Axes()
        return

    def Ellipsoid_samplepoints(self,n_ellipsoid_samplepoints=
                n_ellipsoid_samplepoints):
        rx, ry, rz = self.a,self.b,self.c
        # Calculate the spherical angles:
        u = np.linspace(0, 2 * np.pi, n_ellipsoid_samplepoints)
        v = np.linspace(0, np.pi, n_ellipsoid_samplepoints)
        # Cartesian coordinates that correspond to
        # the spherical angles:
        x = rx * np.outer(np.cos(u), np.sin(v))
        y = ry * np.outer(np.sin(u), np.sin(v))
        z = rz * np.outer(np.ones_like(u), np.cos(v))
        # Must return the points on the form [[x_1,y_1,z_1],
        # [x_2,y_2,z_2],[x_3,...]], where the points are adjusted for
        # the fact that the void are not centered in origo:
        liste=[]
        for i in xrange(x.shape[0]):
            for j in xrange(x.shape[1]):
                liste.append([x[i,j]+self.C[0],y[i,j]+self.C[1],z[i,j]+
                    self.C[2]])
        return liste
```

```
#-------------------------------------------------------------------
#                    Function definitions
#-------------------------------------------------------------------

def lamba(t):
    """
    lambda is a keyword in Python, therefore called lamba
    Returns the stretch ratios at the moment, given the time t
    """
    return np.array([np.exp(constant[0]*t),
                np.exp(constant[1]*t),np.exp(constant[2]*t)])

def epsilon_equivalent_func(t):
    """Returns the equivalent strain at the moment, given the time t"""
    lambda_value_1,lambda_value_2,lambda_value_3 = lamba(t)
    epsilon_1= np.log(lambda_value_1)
    epsilon_2= np.log(lambda_value_2)
    epsilon_3= np.log(lambda_value_3)
    epsilon_eq = (2./3*(epsilon_1**2 + epsilon_2**2 +
                        epsilon_3**2))**0.5
    return epsilon_eq

def epsilon_plastic_rate():
    """Returns the equivalent strain rate, given the time t"""
    rate = (2./3*(constant_a**2+constant_b**2+constant_c**2))**0.5
    return rate

def RT(R_mean):
    return ((1+E)*epsilon_rate
          + (2/3.0*epsilon_j_squared)**0.5*D_constant)*R_mean

def Analytic_solved_RT(void):
    """
    Analytical solution of an initial spherical void with
    constant lode and triaxiality:
    """
    T_constant = Stress_triaxiality[0]
    if T_constant>1.0:
        D_constant=alpha*np.exp(3.0/2*T_constant)
    else:
        D_constant=alpha*T_constant**0.25*np.exp(3.0/2*T_constant)
    #The radius' are measured in principal space for the voids...
    phi_local = phi[0]
    R_1,R_2,R_3 = void.COEFF()
    R_1 = [R_1]
    R_2 = [R_2]
    R_3 = [R_3]
    R_mean = (R_1[0] + R_2[0] + R_3[0])/3.

    for dtime in analytic_time[1:]:
    # analytic_time is defined later, as a global variable...
        epsilon_1,epsilon_2,epsilon_3 = np.log(lamba(dtime))
        A_constant = np.exp((2*(3+phi_local**2)**0.5)/
                            (3+phi_local)*D_constant*epsilon_1)
        B_constant = ((1+E)/D_constant*(A_constant - 1))
        R_1.append(R_mean * (A_constant + (3+phi_local)/
                             (2*(3+phi_local**2)**0.5)*B_constant))
        R_2.append(R_mean * (A_constant - (phi_local)/
```

```
                                ((3+phi_local**2)**0.5)*B_constant))
        R_3.append(R_mean * (A_constant + (phi_local-3)/
                                (2*(3+phi_local**2)**0.5)*B_constant))
    # The evolution of the semi-axes are calculated as:
    R_1=np.array(R_1)
    R_2=np.array(R_2)
    R_3=np.array(R_3)

    #Initial void volume fraction:
    W0 = box_dim[0]/2.
    V_f0 = np.pi/6*(R_mean/W0)**3
    #Current void volume fraction:
    V_f = V_f0 * R_1*R_2*R_3/R_mean**3

    epsilon_1,epsilon_2,epsilon_3 = np.log(lamba(analytic_time))
    # Returns void volume fraction, and the length of the
    # principal semi-axes of the void:
    return V_f,R_1,R_2,R_3

def Analytically_find_time_limits(void):
    T_constant = Stress_triaxiality[0]
    if T_constant>1.0:
        D_constant=alpha*np.exp(3.0/2*T_constant)
    else:
        D_constant=alpha*T_constant**0.25*np.exp(3.0/2*T_constant)
    phi_local = phi[0]
    R_1,R_2,R_3 = void.COEFF()
    R_mean = (R_1 + R_2 + R_3)/3.

    dtime=0.001
    total_time = 0.
    W0 = box_dim[0]/2
    V_f = np.pi/6*(R_mean/W0)**3
    V_f0 = np.pi/6*(R_mean/W0)**3
    while V_f < Upper_Void_Volume_Limit/float(num_voids)
                        and V_f > 0.0:
        # In case of low triaxiality, the voids will shrink, and this
          function
        # will therefore also terminate if the void volume fraction
          reaches 0.
        epsilon_1,epsilon_2,epsilon_3 = np.log(lamba(total_time))
        A_constant = np.exp((2*(3+phi_local**2)**0.5)/
                        (3+phi_local)*D_constant*epsilon_1)
        B_constant = ((1+E)/D_constant*(A_constant - 1))
        R_1 = (R_mean * (A_constant + (3+phi_local)/
                        (2*(3+phi_local**2)**0.5)*B_constant))
        R_2 = (R_mean * (A_constant - (phi_local)/
                        ((3+phi_local**2)**0.5)*B_constant))
        R_3 = (R_mean * (A_constant + (phi_local-3)/
                        (2*(3+phi_local**2)**0.5)*B_constant))
        total_time += dtime
        V_f = V_f0 * R_1*R_2*R_3/R_mean**3
    return total_time
    # Assumed time until the termination criteria is reached,
    # based on constant triaxiality and lode, spherical voids,
    # and no coalescence...
```

```
def Make_linear_function(vector):
    """
    If varying lode or triaxiality is desired, this makes a linear
    varying function between the points given for lode/triaxiality.
    """
    n = len(vector)
    if n == 1:
        Result = np.linspace(vector[0],vector[0],N_timesteps)
        Result = np.append(Result,float(vector[-1]))
    elif n == 2:
        Result = np.linspace(vector[0],vector[1],N_timesteps)
        Result = np.append(Result,float(vector[-1]))
    elif n > 2:
        Result = []
        n_parts = n - 1
        dt_parts = (t_end-t_start)/float(n_parts)
        for i in xrange(n-1):
            Result = Result + list(np.linspace(vector[i],
                     vector[i+1],int(N_timesteps/float(n_parts))))
        len_result = len(Result)
        if len_result != N_timesteps:
            difference = N_timesteps - len_result
            if difference < 0:
                for i in xrange(difference):
                    Result.pop()
            elif difference > 0:
                for i in xrange(difference):
                    Result.append(float(vector[-1]))
        Result = np.append(Result,float(vector[-1]))
    return Result

def define_box(void_volume):
    """Calculates the necessary size of the box:"""
    Volume_box=void_volume/Volume_fraction
    box_dim=[Volume_box**(1/3.0) for i in xrange(3)]
    return box_dim

def UpdateBox(t):
    """The current box dimensions given the time t"""
    current_box=lamba(t)*box_dim
    return current_box

def Make_rand_coord_sys():
    """Create a random oriented orthonormal coordinate basis"""
    a = np.array([[random.random(),random.random(),
                   random.random()] for i in xrange(3)])
    q = np.linalg.qr(a)[0]
    q=np.matrix(q)
    return q,q.T

def Generate_Void(statistical_distribution='Uniform'):
    """Creates unique A and C for the voids"""
    if statistical_distribution=='Uniform':
        T,T_inv=Make_rand_coord_sys()
        semi_axes=[random.uniform(d_min/2.0,d_max/2.0) for
                           i in xrange(3)]
        D=np.diag([1./i**2 for i in semi_axes])
        D=np.matrix(D)
```

```
        return T,D,T_inv
    elif statistical_distribution=='Other':
        # THIS IS AN IMPORTANT FOCUS IN FURTHER WORK! A MUCH MORE
        # SOPHISTICATED PROBABILITY DISTRIBUTION THAN
        # UNIFORM DISTRIBUTION IS NECESSARY.
        """Insert very elegant code here..."""
        return T,D,T_inv,C

def One_Ellipsoid_plot(A,C):
    """
    To plot each ellipsoid
    """
    U, D, V = np.linalg.svd(np.array(A,dtype=float))
    centroid=np.array(C)
    rx, ry, rz = [1/np.sqrt(d) for d in D]
    u, v = np.mgrid[0:2*np.pi:20j,-np.pi/2:np.pi/2:10j]
    exec('u,v=np.mgrid[0:2*np.pi:'+str(N_ellipsoid_plotting_points)
         +'j,-np.pi/2:np.pi/2:'+str(N_ellipsoid_plotting_points)+'j]')

    x=rx*np.cos(u)*np.cos(v)
    y=ry*np.sin(u)*np.cos(v)
    z=rz*np.sin(v)

    for idx in xrange(x.shape[0]):
        for idy in xrange(y.shape[1]):
            x[idx,idy],y[idx,idy],z[idx,idy] = np.dot(np.transpose(V),
                    np.array([x[idx,idy],y[idx,idy], z[idx,idy]])) +
                        centroid

    ax.plot_surface(x, y, z, cstride = 1, rstride = 1, alpha=alpha_plot
        )
    plt.show()
    return

def mvee(points, tol = 0.001):
    """
    Finds the ellipse equation in "center form"
    (x-c).T * A * (x-c) = 1
    See the Theory chapter for an in-depth explanation
    """
    N, d = points.shape
    Q = np.column_stack((points, np.ones(N))).T
    err = tol+1.0
    u = np.ones(N)/N
    while err > tol:
        X = np.dot(np.dot(Q, np.diag(u)), Q.T)
        M = np.diag(np.dot(np.dot(Q.T, np.linalg.inv(X)), Q))
        jdx = np.argmax(M)
        step_size = (M[jdx]-d-1.0)/((d+1)*(M[jdx]-1.0))
        new_u = (1-step_size)*u
        new_u[jdx] += step_size
        err = la.norm(new_u-u)
        u = new_u
    c = np.dot(u,points)
    A = la.inv(np.dot(np.dot(points.T, np.diag(u)), points)
               - np.multiply.outer(c,c))/d
    return A, c
```

```
def Plot_Ellipsoid_2D(void,color):
    """
    To plot the ellipsoids in 2D
    """
    A = void.A
    a_11 = A[0,0]
    a_12 = A[0,1]
    a_22 = A[1,1]
    a_23 = A[1,2]
    a_33 = A[2,2]
    a_13 = A[0,2]
    x_2 = 0.0 - void.C[1]
    # The plot goes through the value x_2=0 (y=0), but may very
    # easily be extended to go through a user defined value
    # The solution is based on the existence of a real discriminant:
    z_1 = (-x_2*(a_11*a_23 - a_12*a_13) + np.sqrt(a_11*(-a_11
            *a_22*a_33*x_2**2 + a_11*a_23**2*x_2**2
            + a_11*a_33 + a_12**2*a_33*x_2**2
            - 2*a_12*a_13*a_23*x_2**2 + a_13**2*a_22*x_2**2
            - a_13**2)))/(a_11*a_33 - a_13**2)
    z_2 = (-a_11*a_23*x_2 + a_12*a_13*x_2 - np.sqrt(a_11*(-a_11
             *a_22*a_33*x_2**2 + a_11*a_23**2*x_2**2
             + a_11*a_33 + a_12**2*a_33*x_2**2
             - 2*a_12*a_13*a_23*x_2**2 + a_13**2*a_22*x_2**2
             - a_13**2)))/(a_11*a_33 - a_13**2)
    z_values = np.linspace(float(z_1),float(z_2),700)
    x_3 = z_values
    x_values1 = (-a_12*x_2 - a_13*x_3 + np.sqrt(-a_11
             *a_22*x_2**2 - 2*a_11*a_23*x_2*x_3
             - a_11*a_33*x_3**2 + a_11 + a_12**2*x_2**2
             + 2*a_12*a_13*x_2*x_3 + a_13**2*x_3**2))/a_11
    x_values2 =  -(a_12*x_2 + a_13*x_3 + np.sqrt(-a_11
              *a_22*x_2**2 - 2*a_11*a_23*x_2*x_3
              - a_11*a_33*x_3**2 + a_11 + a_12**2*x_2**2
              + 2*a_12*a_13*x_2*x_3 + a_13**2*x_3**2))/a_11
    x_values1 = np.delete(x_values1,0)
    x_values1 = np.delete(x_values1,-1)
    x_values2 = np.delete(x_values2,0)
    x_values2 = np.delete(x_values2,-1)
    z_values = np.delete(z_values,0)
    z_values = np.delete(z_values,-1)
    z_values = z_values + void.C[2]
    x_values1 = x_values1 + void.C[0]
    x_values2 = x_values2 + void.C[0]

    plt.plot(x_values1,z_values,str(color))
    plt.plot(x_values2,z_values,str(color))
    plt.plot([x_values1[0],x_values2[0]],[z_values[0],
                  z_values[0]],str(color))
    plt.plot([x_values1[-1],x_values2[-1]],[z_values[-1],
                  z_values[-1]],str(color))
    return
```

```
def Collision_Box(A,C,box_dims):
    """
    This code will check collision with the current box dimensions
    """
    a=[C[0]-0,box_dims[0]-C[0]]
    b=[C[1]-0,box_dims[1]-C[1]]
    c=[C[2]-0,box_dims[2]-C[2]]
    a00,a01,a02,a11,a12,a22=A[0,0],A[0,1],A[0,2],
                                A[1,1],A[1,2],A[2,2]
    Collision_status = []
    # For the case when it is inserted for x2:
    if 2*(-a00*a11 + a01**2) < 0:
        for value in c:
            x2 = value
            # x1 gets its ritical value:
            x1 = x2*(-a00*a12 + a01*a02)/(a00*a11 - a01**2)
            #Check the function value, with variables x1 and x2:
            if (-a00*a22*x2**2 + a00 + a02**2*x2**2
                      + x1**2*(-a00*a11 + a01**2) + x1*(-2*a00*a12*x2
                      + 2*a01*a02*x2)) > 0:
                Collision_status.append(True)
            else:
                Collision_status.append(False)
    else:
        Collision_status.append(True)

    # For the case when it is inserted for x0:
    if 2*(-a11*a22 + a12**2) < 0:
        for value in a:
            x0 = value
            # x2 gets its critical value:
            x2 = x0*(a01*a12 - a02*a11)/(a11*a22 - a12**2)
            #Check the function value, with variables x0 and x2:
            if (-a00*a11*x0**2 + a01**2*x0**2 + a11
                        + x2**2*(-a11*a22 + a12**2) + x2*(2*a01*a12*x0
                         - 2*a02*a11*x0)) > 0:
                Collision_status.append(True)
            else:
                Collision_status.append(False)
    else:
        Collision_status.append(True)

    # For the case when it is inserted for x1:
    if 2*(-a00*a22 + a02**2) < 0:
        for value in b:
            x1 = value
            # x0 gets its critical value:
            x0 = x1*(-a01*a22 + a02*a12)/(a00*a22 - a02**2)
            #Check the function value, with variables x0 and x1:
            if (-a11*a22*x1**2 + a12**2*x1**2 + a22
                    + x0**2*(-a00*a22 + a02**2) + x0*(-2*a01*a22*x1
                    + 2*a02*a12*x1)) > 0:
                Collision_status.append(True)
            else:
                Collision_status.append(False)
    else:
        Collision_status.append(True)
```

```
    if any(Collision_status):
        return True
    else:
        return False


def Collision_Ellipsoids_new(void1,void2,args=[]):
    """
    Determines if two ellipsoids are intercepting or not
    """
    if len(args) == 0:
        A_1 = void1.A
        T_1 = void1.T
        T_1_inv = void1.T_inv
        a,b,c = void1.COEFF()
        #The transformation of A to a unit sphere is done through
        # the following coordinate transformation:
        # y = R * x
        R = np.matrix(np.zeros((3,3)))
        R[0,0] = T_1[0,0] * a
        R[1,0] = T_1[1,0] * a
        R[2,0] = T_1[2,0] * a
        R[0,1] = T_1[0,1] * b
        R[1,1] = T_1[1,1] * b
        R[2,1] = T_1[2,1] * b
        R[0,2] = T_1[0,2] * c
        R[1,2] = T_1[1,2] * c
        R[2,2] = T_1[2,2] * c

        #The transformation of A is then done by R_T * A * R
        A_1_unit = R.T * A_1 * R

        # The other ellipsoid shall then undergo the same
           transformation:
        A_2 = void2.A
        A_2_transf = R.T * A_2 * R

        # Distance between the ellipsoids centers:
        dist1 = np.matrix(void1.C)
        dist2 = np.matrix(void2.C)
        dist = np.linalg.inv(R) * dist2.T - np.linalg.inv(R) * dist1.T

        # The transformed void shall be described in its
        # own coordinate system:
        D_diag, T_new = np.linalg.eig(A_2_transf)
        a, b, c = (1./D_diag[0])**0.5,(1./D_diag[1])**0.5,
                        (1./D_diag[2])**0.5

        # Since A is a symmetric positive definite matrix,
        # the following is also correct:
        T_new_inv = T_new.T
        A_2_eq = T_new_inv * A_2_transf * T_new
        #We now have A_2_eq described in its own coordinate system,
        # i.e with its own eigenvectors as the coordinate system.

        # The point should be transformed in the opposite direction;
        # i.e be stationary while the coord. system rotates:
        point_temp = T_new_inv*dist
```

```
point=np.array([0.,0.,0.])
point[0]=point_temp[0,0]
point[1]=point_temp[1,0]
point[2]=point_temp[2,0]

#The problem is now reduced to find the minimum distance
# between the point 'point' and the ellipsoid
#described by A_2_eq, and see if this is less than unity (1.)!
D_diag, T = np.linalg.eig(A_2_eq)
x0, y0, z0 = point[0],point[1],point[2]

# In the following code, a,b and c is used to denote the
# squares of the semi-axes, which until now has been
# denoted a,b and c.
# I.e, from now on, a == a**2, b==b**2 and c==c**2:
a = 1./D_diag[0]
b = 1./D_diag[1]
c = 1./D_diag[2]
# The 6.order polynomial has the following coefficients:
# c_1*x**n + c_2 * x**n-1 + ... + c_n = 0, where
# the values has been calculated (with SymPy)to:
c1 = 1.
c2 = -2.*(a+b+c)
c3 = a**2+b**2+c**2+4.*(b*c+c*a+a*b)
            -(a*x0**2+b*y0**2+c*z0**2)
c4 = 2.*(-a**2*(b+c)-b**2*(c+a)-c**2*(a+b)-4*a*b*c
           +a*(b+c)*x0**2+b*(c+a)*y0**2 +c*(a+b)*z0**2)
c5 = b**2*c**2+c**2*a**2+a**2*b**2+4*a*b*c*(a+b+c)
              -a*(b**2+c**2)*x0**2-b*(c**2+a**2)*y0**2
              -c*(a**2+b**2)*z0**2-4*a*b*c*(x0**2+y0**2+z0**2)
c6 = 2*a*b*c*((b+c)*x0**2+(c+a)*y0**2+(a+b)*z0**2
              -b*c-c*a-a*b)
c7 = a*b*c*(a*b*c-b*c*x0**2-c*a*y0**2-a*b*z0**2)

# Find all six roots:
t = np.roots([c1,c2,c3,c4,c5,c6,c7])

# Determine which are the real roots:
p = np.imag(t) == 0

# Narrow t vector down to only real roots:
t = t[p]
# Just to remove +0j at the end of the term:
t = np.real(t)

# Find corresponding x,y,z coordinates
x = a/(a-t)*x0
y = b/(b-t)*y0
z = c/(c-t)*z0

# Choose the minimum distance:
d = (min((x-x0)**2+(y-y0)**2+(z-z0)**2))**0.5
# This distance is not the real physical distance between the
# two voids, but the length 1 is the limit value for
   intersection
# or not, in the different coordinate system!
if d <=1.:
    return True
```

```
        else:
            return False
    else:
        """
        This part is very similar, but is needed to compute
           intersection
        between potential voids, i.e they have yet to be instantiated
        as a Void instance! The code underwent some minor changes.
        """
        A_1 = void1.A
        T_1 = void1.T
        T_1_inv = void1.T_inv
        a,b,c = void1.COEFF()
        dist1 = np.matrix(void1.C)
        dist2 = np.matrix(args[1])
        A_2 = args[0]

        R = np.matrix(np.zeros((3,3)))
        R[0,0] = T_1[0,0] * a
        R[1,0] = T_1[1,0] * a
        R[2,0] = T_1[2,0] * a
        R[0,1] = T_1[0,1] * b
        R[1,1] = T_1[1,1] * b
        R[2,1] = T_1[2,1] * b
        R[0,2] = T_1[0,2] * c
        R[1,2] = T_1[1,2] * c
        R[2,2] = T_1[2,2] * c

        #The transformation of A is then done by R_T * A * R
        A_1_unit = R.T * A_1 * R

        #The other ellipsoid shall then undergo the same transformation
           :
        A_2_transf = R.T * A_2 * R

        #Distance between the ellipsoids centers:
        dist = np.linalg.inv(R) * dist2.T - np.linalg.inv(R) * dist1.T

        #The transformed void shall be described in its own
        # coordinate system:
        D_diag, T_new = np.linalg.eig(A_2_transf)
        a, b, c = (1./D_diag[0])**0.5,(1./D_diag[1])**0.5,
                          (1./D_diag[2])**0.5

        #Since A is a symmetric positive definite matrix:
        T_new_inv = T_new.T

        A_2_eq = T_new_inv * A_2_transf * T_new

        #The point should be transformed in the opposite direction;
        # i.e be stationary while the coord. system rotates:
        point_temp = T_new_inv*dist
        point=np.array([0.,0.,0.])
        point[0]=point_temp[0,0]
        point[1]=point_temp[1,0]
        point[2]=point_temp[2,0]

        #The problem is now reduced to find the minimum distance
```

```
        # between the point 'point' and the ellipsoid described by
        # A_2_eq, and see if this is less than unity (1.).
        D_diag, T = np.linalg.eig(A_2_eq)
        x0, y0, z0 = point[0],point[1],point[2]
        a = 1./D_diag[0]
        b = 1./D_diag[1]
        c = 1./D_diag[2]
        # The 6.order polynomial has the following coefficients:
        # c_1*x**n + c_2 * x**n-1 + ... + c_n = 0, where
        # the values has been calculated (with SymPy)to:
        c1 = 1.
        c2 = -2.*(a+b+c)
        c3 = a**2+b**2+c**2+4.*(b*c+c*a+a*b)
                    -(a*x0**2+b*y0**2+c*z0**2)
        c4 = 2.*(-a**2*(b+c)-b**2*(c+a)-c**2*(a+b)-4*a*b*c
                    +a*(b+c)*x0**2+b*(c+a)*y0**2 +c*(a+b)*z0**2)
        c5 = b**2*c**2+c**2*a**2+a**2*b**2+4*a*b*c*(a+b+c)
                      -a*(b**2+c**2)*x0**2-b*(c**2+a**2)*y0**2
                      -c*(a**2+b**2)*z0**2-4*a*b*c*(x0**2+y0**2+z0**2)
        c6 = 2*a*b*c*((b+c)*x0**2+(c+a)*y0**2+(a+b)*z0**2
                      -b*c-c*a-a*b)
        c7 = a*b*c*(a*b*c-b*c*x0**2-c*a*y0**2-a*b*z0**2)

        # Find all six roots:
        t = np.roots([c1,c2,c3,c4,c5,c6,c7])
        # Determine which are the real roots:
        p = np.imag(t) == 0

        # Narrow t vector down to only real roots:
        t = t[p]
        # Just to remove +0j at the end of the term:
        t = np.real(t)

        # Find corresponding x,y,z coordinates
        x = a/(a-t)*x0
        y = b/(b-t)*y0
        z = c/(c-t)*z0

        # Choose the minimum distance:
        d = (min((x-x0)**2+(y-y0)**2+(z-z0)**2))**0.5
        if d <=1.:
            return True
        else:
            return False



def Generate_C():
    """
    Generate a random position for a void
    """
    C=np.array([random.uniform(0.0,box_dim[0]),
                random.uniform(0.0,box_dim[1]),
                random.uniform(0.0,box_dim[2])],dtype=float)
    return C
```

```
def Find_position(A):
    """
    Keep generating positions for the voids, until the position doesn't
    result in any intersections with other voids
    """
    status1 = 'Collision'
    while status1=='Collision':
        C = Generate_C()
        status1 = 'Not Collision'
        for void in Voids:
            if Collision_Ellipsoids_new(void,void,args=[A,C]):
                status1 = 'Collision'
    return C


def checkEqual(lst):
    """
    Checks wether a list consists of only one value, i.e all the
    terms in the list are equal
    """
    return lst[1:] == lst[:-1]


def find_neighboors():
    """
    Checks wich of the voids that can be classified as neighbors.
    A circle with radius 1.1 times the largest semi-axis of the void
    is thought to encompass each of the 2 voids in question,
    and if these circles intersects, the voids are classified as
       neighbors.
    """
    tempor_index = 0
    for void_search_1 in Voids:
        tempor_index +=1
        maximum_axis_1 = 1.1*max(void_search_1.a,
                                 void_search_1.b,void_search_1.c)
        for void_search_2 in Voids[tempor_index:]:
            maximum_axis_2 = 1.1*max(void_search_2.a,
                                 void_search_2.b,void_search_2.c)
            if np.linalg.norm(void_search_1.C - void_search_2.C)
                                  < (maximum_axis_1+maximum_axis_2):
                void_search_1.neighboors.add(void_search_2)
                void_search_2.neighboors.add(void_search_1)
    return
```

```
#-----------------------------------------------------------------
#       Generate voids before void creation
#-----------------------------------------------------------------

"""
The voids can not be created with a given position before the
box's dimension is determined, which cannot be determined
before the voids total volume is known. In this part of the program,
all the voids are generated, but the void creation is postponed
to the next section.
"""
A_from_voids=[]
Volume_Voids = 0
for i in xrange(num_voids):
    # Depending on the user specified parameter 'orientation':

    if orientation == 'Rand':
        T,D,T_inv=Generate_Void()

    # or:
    elif orientation == 'Spherical':
        T=np.matrix([[1,0,0],[0,1,0],[0,0,1]])
        T_inv=T.T
        D=np.matrix(np.diag([1/(d_min/2.)**2 for kl in xrange(3)]))

    elif orientation == 'Langs':
        T,D,T_inv=Generate_Void()
        T=np.matrix([[1,0,0],[0,1,0],[0,0,1]])
        T_inv=T.T

    elif orientation == 'Skra':
        T,D,T_inv=Generate_Void()
        T = np.matrix([[1,0,1],[0,1,0],[-1,0,1]])
        T_inv = T.T

    elif orientation == 'Spherical_error':
        T,D,T_inv=Generate_Void()
        D=np.matrix(np.diag([1/(d_min/2.)**2 for kl in xrange(3)]))

    elif orientation == 'One_with_angle':
        #Get the angle in radians:
        angle_temp = angle_for_1_void*np.pi/180.
        T = np.matrix([[1.,0.,0.],[0.,1.,0.],[0.,0.,1.]])
        #Rotation about the y-axis:
        R = np.matrix([[np.cos(angle_temp), 0., -np.sin(angle_temp)],
                [0., 1., 0.],[np.sin(angle_temp), 0., np.cos(
                    angle_temp)]])
        T = R*T
        T_inv = T.T
        semi_axes_b,semi_axes_c = d_min,d_min
        semi_axes_a = ratio_for_1_void * semi_axes_b
        D=np.matrix(np.diag([1/(semi_axes_a/2.)**2, 1/(semi_axes_b/
                            2.)**2, 1/(semi_axes_c/2.)**2]))
        list_of_angles = [angle_temp*180./np.pi]

    elif orientation == 'Two_with_coalescence':
        if Two_with_coalescence_orientation == 'Spherical':
            T=np.matrix([[1,0,0],[0,1,0],[0,0,1]])
```

```
            T_inv=T.T
            D=np.matrix(np.diag([1/(d_min/2.)**2 for kl in xrange(3)]))
        elif Two_with_coalescence_orientation == 'Rand':
            T,D,T_inv=Generate_Void()
            T = np.matrix([[1.,0.,0.],[0.,1.,0.],[0.,0.,1.]])
            #Rotation about the y-axis:
            angle_for_2_void = random.uniform(0,np.pi/2.)
            R = np.matrix(
                  [[np.cos(angle_for_2_void), 0.,-np.sin(
                      angle_for_2_void)],
                  [0., 1., 0.],
                  [np.sin(angle_for_2_void), 0., np.cos(angle_for_2_void
                      )]])
            T = R*T
            T_inv = T.T
        elif Two_with_coalescence_orientation == 'Skra':
            T,D,T_inv=Generate_Void()
            T = np.matrix([[1,0,1],[0,1,0],[-1,0,1]])
            T_inv = T.T
        elif Two_with_coalescence_orientation == 'Langs':
            T,D,T_inv=Generate_Void()
            T=np.matrix([[1,0,0],[0,1,0],[0,0,1]])
            T_inv=T.T
    A=T*D*T_inv
    A_from_voids.append([A,T,D,T_inv])
    coefficients = np.array([np.sqrt(1./D[i,i]) for i in xrange(3)])
    a,b,c = coefficients[0],coefficients[1],coefficients[2]
    volume=4.0/3*np.pi*a*b*c

    # Get the total volume of the voids, needed to define the box:
    Volume_Voids += volume

#Define the box:
box_dim = define_box(Volume_Voids)
Volume_box = box_dim[0] * box_dim[1] * box_dim[2]
```

```
#-----------------------------------------------------------------------
#    Void creation and setup for the main part of the program
#-----------------------------------------------------------------------

Voids=[]
if num_voids != 1:
    if orientation != 'Two_with_coalescence':
        for i in xrange(num_voids):
            k= i+1
            A,T,D,T_inv = A_from_voids[i]
            if k == 1:
                C=Generate_C()
             #The voids are created with their own unique name:
                exec('void_'+str(k)+'=Void(T,D,T_inv,C,\'void_'+str(k)+
                    '\')')
                exec('Voids.append(void_'+str(k)+')')
            else:
                D_diag=np.diag(D)
                COEFF=np.array([np.sqrt(1./D_diag[l]) for l in xrange
                    (3)])
                C = Find_position(A)
                exec('void_'+str(k)+'=Void(T,D,T_inv,C,\'void_'+str(k)+
                    '\')')
                exec('Voids.append(void_'+str(k)+')')
    else:
        # Which means that 'orientation'=='Two_with_coalescence'...
        # Creation of void_1:
        A,T,D,T_inv = A_from_voids[0]
        largest_semi_axes = max([(1./D[i,i])**0.5 for i in xrange(3)])
        C = np.array([box_dim[0]/2.-2*largest_semi_axes,
                                  0.0,box_dim[0]/2.])
        void_1=Void(T,D,T_inv,C,'void_1')
        Voids.append(void_1)

        # Creation of void_2:
        A,T,D,T_inv = A_from_voids[1]
        largest_semi_axes = max([(1./D[i,i])**0.5 for i in xrange(3)])
        C = np.array([box_dim[0]/2.+2*largest_semi_axes,
                                    0.0,box_dim[0]/2.])
        void_2=Void(T,D,T_inv,C,'void_2')
        Voids.append(void_2)
        void_1.neighboors.add(void_2)
elif num_voids==1:
    A,T,D,T_inv = A_from_voids[0]
    # The void is positioned in the center of the box:
    C=np.array(box_dim)/2.
    void_1=Void(T,D,T_inv,C,'void_1')
    Voids.append(void_1)

# Find the time where the analytical solution has reached the
# given limit volume value:
if num_voids == 1:
    t_end = Analytically_find_time_limits(void_1)
    dt = (t_end-t_start)/float(N_timesteps)
    analytic_time = np.linspace(t_start,t_end,200)
    if len(Stress_triaxiality) == 1 and len(phi) == 1:
        # An analytical solution is calculated:
        Analytical_solution = Analytic_solved_RT(void_1)[0]
```

```
        Analytical_eps_eq = epsilon_equivalent_func(analytic_time)
else:
    t_end=[]
    len_Voids = len(Voids)
    """
    Finding the analytical time for every tenth void, and choosing
    the largest as t_end. Coalescence is not taken into account,
    so t_end is not reached if coalescence happens, which it usually
    does.
    """
    for void in Voids[:int(m.ceil(len_Voids/10.))]:
        t_end.append(Analytically_find_time_limits(void))
    t_end = max(t_end)
    dt = (t_end-t_start)/float(N_timesteps)
    analytic_time = np.linspace(t_start,t_end,200)
    Analytical_solution = np.zeros(len(analytic_time))
    Analytical_eps_eq = epsilon_equivalent_func(analytic_time)
    if len(Stress_triaxiality) == 1 and len(phi) == 1:
        #Calculate an analytical solution, to have something
        # to compare the results to. As the number of voids increases,
        # they will converge towards this if it weren't for the
        # Pac-Man effect.
        for void in Voids:
            Analytic_solution = Analytic_solved_RT(void)
            Analytical_solution += np.array(Analytic_solution[0],
                                            dtype=float)
```

```
#-------------------------------------------------------------
#             Main part of the program
#-------------------------------------------------------------

# To keep track of how the void volume fraction evolves,
# a list is made:
Volume_percentage=[Volume_Voids/Volume_box]

# To keep track of the Pac-Man effect:
Volume_percentage_fake = [0.,0.]

# To remember at which times the plot was taken from:
Time_at_plot = []

fig_number=1
time_counter=0
t = t_start
epsilon_equivalent=[0.]

Stress_triaxiality = Make_linear_function(Stress_triaxiality)
phi_vector = Make_linear_function(phi)
epsilon_plastic_rate_vector=[]
epsilon_11 = []

while t < t_end and Volume_percentage[-1] < Upper_Void_Volume_Limit:
    if time_counter % 7 == 0:
    # Every 7th increment, starting with the first, the
    # neighboors are controlled:
        find_neighboors()

    # Stress triaxiality at this time increment:
    T_constant=Stress_triaxiality[time_counter]

    # Lode parameter at this time increment:
    phi = phi_vector[time_counter]

    if T_constant>1.0:
        D_constant=alpha*np.exp(3.0/2*T_constant)
    else:
        D_constant=alpha*T_constant**0.25*np.exp(3.0/2*T_constant)

    # Determine the constants in the velocity gradient tensor L:
    constant_b = -phi*constant_a*2/(3+phi)
    constant_c = constant_a*(phi - 3)/(phi + 3)
    constant = np.array([constant_a,constant_b,constant_c])
    L_diag=np.array([constant[0],constant[1],constant[2]])
    L=np.diag(L_diag)

    epsilon_plastic_rate_vector.append(epsilon_plastic_rate())
    epsilon_j_squared = L_diag[0]**2 + L_diag[1]**2 + L_diag[2]**2
    epsilon_rate=L_diag
    epsilon_11.append(constant_a*t)

    # Find the current dimensions of the box:
    Current_box = box_dim*lamba(t)

    # Uses the current box's volume, just to control that the box's
    # volume actually are kept constant (which it should):
```

```
Current_box_volume=Current_box[0]*Current_box[1]*Current_box[2]
Current_Volume = 0

for void in Voids:
    void.Rotate_and_grow(t)
    Current_Volume += void.Volume()
    if Remove_Volume_Outside_Box == True:
        """
        The following code may be used to remove the parts of the
        voids that are outside the current box, i.e reduce the void
        volume fraction in the box if the voids are partly outside
        the box. Not used in this thesis, because the voids outside
            the
        box are just as likely to grow partly into the box.
        The assumption made is that these two effects cancel each
        other out. But, the code may be used to remove overlapping
        ellipsoids as described in further work,
        with only minor changes, so the procedure is shown here.
        """
        x_list_low=[]
        x_list_high=[]
        y_list_low=[]
        y_list_high=[]
        z_list_low=[]
        z_list_high=[]
        if New_Collision_Box(void.A,void.C,Current_box):
            points=void.Ellipsoid_samplepoints(
                n_ellipsoid_samplepoints=10)
            for x,y,z in points:
                if x < 0:
                    x_list_low.append([x,y,z])
                if x > Current_box[0]:
                    x_list_high.append([x,y,z])
                if y < 0:
                    y_list_low.append([x,y,z])
                if y > Current_box[1]:
                    y_list_high.append([x,y,z])
                if z < 0:
                    z_list_low.append([x,y,z])
                if z > Current_box[2]:
                    z_list_high.append([x,y,z])
                """
                Need at least 4 points to make a ConvexHull
                in 3D, and if these points are extremely close
                to each other, there may occur some
                instabilities upon creation of the
                ConvexHull, so approximately 8 points outside the
                box should occur before it is deemed necessary
                to remove the excess void volume.
                """
                if len(x_list_low) > 8:
                    try:
                        x_low=ConvexHull(x_list_low)
                        negative_volume=x_low.volume
                        Current_Volume -= negative_volume
                    except: pass
                if len(x_list_high) > 8:
                    try:
```

```
                        x_high=ConvexHull(x_list_high)
                        negative_volume=x_high.volume
                        Current_Volume -= negative_volume
                    except: pass
                if len(y_list_low) > 8:
                    try:
                        y_low=ConvexHull(y_list_low)
                        negative_volume=y_low.volume
                        Current_Volume -= negative_volume
                    except: pass
                if len(y_list_high) > 8:
                    try:
                        y_high=ConvexHull(y_list_high)
                        negative_volume=y_high.volume
                        Current_Volume -= negative_volume
                    except: pass
                if len(z_list_low) > 8:
                    try:
                        z_low=ConvexHull(z_list_low)
                        negative_volume=z_low.volume
                        Current_Volume -= negative_volume
                    except: pass
                if len(z_list_high) > 8:
                    try:
                        z_high=ConvexHull(z_list_high)
                        negative_volume=z_high.volume
                        Current_Volume -= negative_volume
                    except: pass
"""
Finally, the rather unnecessary part of the code is over.
"""
Volume_percentage.append(Current_Volume/Volume_box)
checked = set()
for void in Voids:
    for neighbor in void.neighboors:
        if (neighbor,void) in checked:
            # If controlled before, just the other way around, skip
                it:
            continue
        if Collision_Ellipsoids_new(void,neighbor):
            points=np.array(list(void.Ellipsoid_samplepoints())
                            + list(neighbor.Ellipsoid_samplepoints()))
            points=np.array(points,dtype=float)
            A,C=mvee(points)
            D_diag,T=np.linalg.eig(A)
            T=np.matrix(T)
            T_inv=T.T
            C = np.array(C)
            C_origin = C / lamba(t)

            # The scaling of the voids, to control the Pac-Man
                effect:
            V_s = void.Volume() + neighbor.Volume()
            scaler = (4*np.pi/(3*V_s*(D_diag[0]*D_diag[1]
                                *D_diag[2])**0.5))**(2./3)
            scaler = (1./scaler + (1 - 1./scaler)
                                *Reduced_Pac_Man_Effect) * scaler
            D_diag = D_diag*scaler
```

```
                D=np.matrix(np.diag(D_diag))
                name= void.name[4:]+neighbor.name[4:]
                exec('void'+name+'=Void(T,D,T_inv,C,\'void'+ name +'\')
                    ')
                exec('void'+name+'.C_origin=C_origin')
                exec('void'+name+'.Update_Void()')
                exec('Voids.append(void'+name+')')
                exec('void'+name+
              '.neighboors=void.neighboors.union(neighbor.neighboors)')

                for void_temp in Voids:
                    void_temp.neighboors.discard(void)
                    void_temp.neighboors.discard(neighbor)
                ind1=Voids.index(void)
                temp_name=Voids[ind1].name
                # Delete void_1 from the list
                del Voids[ind1]
                # Delete void_1 from the namespace
                exec('del ' + temp_name)
                ind2=Voids.index(neighbor)
                temp_name=Voids[ind2].name
                # Delete void_2 from the list
                del Voids[ind2]
                # Delete void_2 from the namespace
                exec('del ' + temp_name)
                # The new void need to know its neighbors:
                find_neighboors()
                break
            # If not collision, the void pair is marked as checked:
            checked.add((void,neighbor))

    # To find the amount of void volume from the Pac-Man effect:
Current_Volume_fake = 0
for void in Voids:
    Current_Volume_fake += void.Volume()
Volume_percentage_fake_now = Current_Volume_fake/Volume_box
Volume_percentage_beforefake = Current_Volume/Volume_box
Volume_percentage_fake.append(Volume_percentage_fake_now
                  -Volume_percentage_beforefake)

t += dt
eps_eq=epsilon_equivalent_func(t)
epsilon_equivalent.append(eps_eq)

# Plotting of the voids, if the user have specified it:
if fig_number % fig_divider == 0 or fig_number == 1:
    # Remember when the plot was made:
    Time_at_plot.append(int(round(t/dt)))

    #Making the figure:
    exec('fig'+str(fig_number)+'=plt.figure()')
    exec('ax=fig'+str(fig_number)+
                      '.add_subplot(111,projection=\'3d\')')
    orig_limits = np.array([[0,box_dim[0]*2],[0,box_dim[1]],
                            [0,box_dim[2]]],dtype=float)
    limits = np.array([[0,Current_box[0]],[0,Current_box[1]],
                            [0,Current_box[2]]],dtype=float)
```

```
        # The extra scaling above is to ensure that the voids
        # stays in the figure..
        ax.set_xlim3d(orig_limits[0])
        ax.set_ylim3d(orig_limits[1])
        ax.set_zlim3d(orig_limits[2])

        ax.set_xlabel('x-axis,micrometer($\mu$m)')
        ax.set_ylabel('y-axis,micrometer($\mu$m)')
        ax.set_zlabel('z-axis,micrometer($\mu$m)')
        for void in Voids:
            One_Ellipsoid_plot(void.A,void.C)
    fig_number += 1

    #Tell the user how much of the increments that are done:
    time_counter+=1
    if time_counter % 4 == 0:
        print '{} % of the increments are done...'.format(int(t*100./
            t_end))

# Using the trapez method to find the equivalent plastic strain
# if varying lode angle or stress triaxiality during the increments:
if checkEqual(list(Stress_triaxiality)) and
                    checkEqual(list(phi_vector)):
    # Constant values, so no need to use numerical integration:
    pass
else:
    delta_time = (t_end-t_start)/float(N_timesteps+1)
    integrating_vec = 0.5*(np.array(epsilon_plastic_rate_vector[:-1])
                     + np.array(epsilon_plastic_rate_vector[1:]))
    incremental_plastic_strains = integrating_vec * delta_time
    epsilon_plastic_vector = np.array([0.])
    running_plastic_strain_sum = np.cumsum(incremental_plastic_strains)
    epsilon_plastic_vector = np.append(epsilon_plastic_vector,
                                       running_plastic_strain_sum)
    epsilon_plastic_vector = np.append(epsilon_plastic_vector,
                epsilon_plastic_vector[-1] +
                    epsilon_plastic_rate_vector[-1]*delta_time)

# The Pac-Man effect is the cumulative sum of the increased
# volume at coalescence:
Volume_percentage_fake = np.cumsum(Volume_percentage_fake)

#-------------------------------------------------------------
#                       RESULT
#-------------------------------------------------------------

#The result of the program is the calculated fracture strain:
print epsilon_equivalent[-1]
```