Numerical Implementation of Brittle Failure Model for Glass

Nora Storebø Næss

Master of Science in Engineering and ICT
Submission date: January 2017
Supervisor: Odd Sture Hopperstad, KT
Co-supervisor: David Morin, KT
Karoline Osnes, 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:
Computational Mechanics	January 5, 2017	9 + 92 + 45

TITLE:
Numerical implementation of brittle failure model for glass

BY:
Nora Storebø Næss

SUMMARY:
This thesis is concerned with a strength model for float glass. Failure of glass is usually initiated from surface flaws and the glass is reckoned as failed with the first failing flaw. The model implements a flaw map over the glass surface of arbitrary orientations and flaw lengths of maximum 200 µm. Every flaw is checked for failure for an applied load until the critical load for the first failing flaw is found through a search method. The plate geometry is applied several flaw maps for a Monte Carlo analysis, resulting in a probability distribution for strength. This means that rather than assuming the probability distribution a priori which requires parameters obtained from experimental test series, as is done for the traditional probabilistic strength model Weibull distribution, it is generated from the analysis. Additionally, the model provides information about fracture origin in the glass plate.

An analytical solution of the model was implemented in MATLAB, and verified with data from Abaqus FEA. Then the model was implemented in an Abaqus post-processing Python script and further verified by comparing with the analytical solution. The verification processes yielded satisfying results and the model is assumed to be working according to the mathematical foundation. Additionally the flaw map property of the model was applied to an Abaqus VUSDLFD subroutine, exploring visualization of failing flaws.

The model was subjected to parameter studies examining the model response from different plate and flaw geometries, yielding reasonable results. The resulting distributions were compared with fitted Normal and Weibull distributions, and in most cases the Weibull distribution provided the best fit. The Weibull modulus from the fitted Weibull distributions varied for plates with different numbers of flaws, suggesting it is inaccurate to assume this parameter as a material constant. Further, case studies comparing the model behavior to experimental results from four point bending tests were performed. In addition the model behavior were examined under the influence of blast loads. The studies disclosed the notable potential of the model. Finally, suggestions for further work with the model and the software in this thesis were presented.

RESPONSIBLE TEACHER: Prof. Odd Sture Hopperstad

SUPERVISOR(S) Prof. Odd Sture Hopperstad, Dr. David Morin and Ph.D. Karoline Osnes

CARRIED OUT AT: NTNU

Institutt for konstruksjonsteknikk
Fakultet for ingeniørvitenskap og teknologi
NTNU- Norges teknisk- naturvitenskapelige universitet

TILGJENGELIGHET

MASTEROPPGAVE 2016

FAGOMRÅDE:	DATO:	ANTALL SIDER:
Beregningsmekanikk	05.01.2017	9 + 92 + 45

TITTEL:
Numerisk implementering av modell for sprøtt brudd i glass.

UTFØRT AV:
Nora Storebø Næss

SAMMENDRAG:
Denne avhandlingen tar for seg en styrkemodell for float glass. Brudd av glass starter vanligvis i en overflatefeil, og glasset kan regnes som knust når den første overflatefeilen sprekker opp. Modellen implementerer et kart av overflatefeil over glassoverflaten med tilfeldige orienteringer og feillengder på maksimum 200 µm. Hver feil er kontrollert for brudd for en påført last inntil den kritiske lasten for den feilen som forårsaker brudd er funnet gjennom en søkemetode. Plategeometrien er påført flere feilfordeling for en Monte Carlo analyse, hvilket resulterer i en sannsynlighetsfordeling for styrke. Dette betyr at heller enn å anta sannsynlighetsfordelingen a priori, noe som krever parametere fra en rekke eksperimentelle tester, slik som for den tradisjonelle probabilistiske styrkemodellen Weibullfordeling, så blir sannsynlighetsfordelingen generert fra analysen. I tillegg så gir modellen informasjon om lokasjonen til bruddet i glassplaten.

En analytisk løsning av modellen var implementert i MATLAB, og videre verifisert med data fra Abaqus. Deretter ble modellen implementert i et Abaqus postprosesseringsscript i Python og denne ble videre verifisert ved å sammenligne resultater med den analytiske løsningen. Verifiseringsprosessene ga gode resultater og modellen antas å fungere i forhold til det matematiske grunnlaget. I tillegg ble modellens egenskap feilfordelingen anvendt i en Abaqus VUSDLFD subrutine med formålet å utforske visualisering av brudd i glass.

Det ble utført et parameterstudie på modellen som undersøkte modellens respons på forskjellige plate og feilgeometrier, hvilket ga rimelige resultater. De resulterende sannsynlighetsfordelingene var sammenlignet med tilpassede Normal og Weibullfordelinger, og i de fleste tilfellene var det Weibullfordelingen som beskrev resultatene best. Det ble funnet at for plater med forskjellig antall feil ville Weibullmodulen fra de tilpassede Weibullfordelingene variere. Dette antyder at det blir feilaktig å anta at Weibullmodulen er en materialkonstant. Videre var det utført case studier på modellen som sammenligner modellens oppførsel med foreliggende eksperimentelle resultater fra firepunktbøyningstester. I tillegg var modellens oppførsel under eksplosjonslaster undersøkt. Disse studiene viser det nevneverdige potensialet til styrkemodellen. Til slutt ble forslag til videre arbeid med modellen og programvaren i denne avhandlingen presentert.

FAGLÆRER: Prof. Odd Sture Hopperstad

VEILEDER(E): Prof. Odd Sture Hopperstad, Dr. David Morin og Ph.D. Karoline Osnes

UTFØRT VED: NTNU

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

MASTER'S THESIS 2016

for

Nora Storebø Næss

Numerical implementation of brittle failure model for glass

Glass is a brittle material and behaves elastically until failure. The fracture strength is largely driven by existing microscopic cracks distributed over the surface. When tensile stresses are applied to the glass and normal to the small cracks, they will open and grow when the stresses are greater than a given threshold, leading to fast fracture. Owing to the presence of microscopic surface cracks, the strength will have a large scatter, and the size of the loaded surface and the state of stress have a significant influence on the fracture strength. This is due to the increased probability of finding a dominant micro-crack at a right angle to the maximum stress. Consequently, the strength of the glass is lower for a larger specimen than for a smaller, and under biaxial tension compared to uniaxial tension. With regards to the fracture modelling of glass, it is necessary to use a statistical approach due to the large scatter in strength. Recently, a new brittle failure model for assessing the strength of structural members made of annealed glass plates was presented by Yankelevsky [D. Z. Yankelevsky. Strength prediction of annealed glass plates – A new model. Engineering Structures 79 (2014) 244–255)].

The research project has three main objectives: (1) to implement the above-mentioned brittle failure model for glass materials in a stand-alone code (MATLAB and Python) and the nonlinear finite element code Abaqus; (2) to verify the implementations by use of existing experimental, analytical and numerical results; (3) to apply the material model in a parametric study on the behaviour of glass.

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

1. Behaviour of glass: Use existing literature to study the behaviour and modelling of glass materials.
2. Model formulation: Establish the mathematical formulation of the brittle failure model for glass materials including the statistical distribution of the cracks.
3. Numerical implementation: Establish the algorithm for numerical implementation of the brittle failure model in the stand-alone code and Abaqus.
4. Verification study: Use existing experimental, analytical and numerical results to verify the implementation of the brittle failure model.
5. Numerical study: Perform a parametric study on the effects of the plate geometry and shape of the flaws, and a case study on different loading situations.

Supervisors: Karoline Osnes, David Morin, Odd Sture Hopperstad (NTNU)

The candidate may agree with the supervisors to pay particular attention to specific parts of the investigation, or to include other aspects than those already mentioned. The thesis must be written as a research report, according to current requirements and submitted to Department of Structural Engineering, NTNU, no later than January 5, 2017.

NTNU, August 15, 2016

Odd Sture Hopperstad

Odd Sture Hopperstad
Professor

Acknowledgement

This thesis was carried out the fall semester of 2016 as part of the study program Engineering and ICT with the specialization Structural Engineering at the Norwegian University of Science and Technology. The thesis was assigned by the research group Structural Impact Laboratory (SIMLab) at the Department of Structural Engineering.

I would like to express my sincere gratitude to my supervisors:

Professor Odd Sture Hopperstad for his motivating positivity and interest in the project, and practical understanding for any problem I presented to him.

Dr. David Morin for his invaluable help and guidance with the technologies and software implementations in this thesis.

PhD Candidate Karoline Osnes for her continuous support and assistance whenever needed.

I also want to extend a special thanks to PhD Candidate and friend Henrik Granum for acting as a sparring partner in times of despair.

Nora Storebø Vagg

Trondheim, Norway

January 5*th*, 2017

Abstract

This thesis is concerned with a strength model for float glass. Failure of glass is usually initiated from surface flaws and the glass is reckoned as failed with the first failing flaw. The model implements a flaw map over the glass surface of arbitrary orientations and flaw lengths of maximum 200 μm. Every flaw is checked for failure for an applied load until the critical load for the first failing flaw is found through a search method. The plate geometry is applied several flaw maps for a Monte Carlo analysis, resulting in a probability distribution for strength. This means that rather than assuming the probability distribution a priori which requires parameters obtained from experimental test series, as is done for the traditional probabilistic strength model Weibull distribution, it is generated from the analysis. Additionally, the model provides information about fracture origin in the glass plate.

An analytical solution of the model was implemented in MATLAB, and verified with data from Abaqus FEA. Then the model was implemented in an Abaqus post-processing Python script and further verified by comparing with the analytical solution. The verification processes yielded satisfying results and the model is assumed to be working according to the mathematical foundation. Additionally the flaw map property of the model was applied to an Abaqus VUSDLFD subroutine, exploring visualization of failing flaws.

The model was subjected to parameter studies examining the model response from different plate and flaw geometries, yielding reasonable results. The resulting distributions were compared with fitted Normal and Weibull distributions, and in most cases the Weibull distribution provided the best fit. The Weibull modulus from the fitted Weibull distributions varied for plates with different numbers of flaws, suggesting it is inaccurate to assume this parameter as a material constant. Further, case studies comparing the model behavior to experimental results from four point bending tests were performed. In addition the model behavior were examined under the influence of blast loads. The studies disclosed the notable potential of the model. Finally, suggestions for further work with the model and the software in this thesis were presented.

Contents

Chapter 1

Introduction

1.1 Motivation

Glass is a brittle material, and will therefore behave elastically until failure. The strength of glass is largely affected by the size of surface flaws[1]. At the present date, the traditional strength models for brittle materials require data from experimental tests, and due to the large scatter of results, a significant number of tests. Even then, the results often yield dissatisfying accuracy when applied to other combinations of loads and geometries than in the test series[2].

For determining the strength of a glass plate there are two main approaches today, the deterministic and the probabilistic approach[3]. The large scatter in strength suggests a probabilistic behavior and thus a deterministic approach to be unsuitable. Probablistic methods has commonly been applied to the problem of predicting glass strength, where the Weibull distribution is frequently used[4]. Other distributions such as the Normal and Log-Normal has also been suggested, but it is uncertain which distribution is most robust with regards to failure of glass[5]. The statistical strength models for brittle materials might need to be reconsidered.

In 2014 D.Z. Yankelevsky presented a new model for strength in float glass suggesting the probability distribution should not be assumed a priori, but rather come as a result of the analysis[6]. Not only eliminating the need for a pre-determined probability distribution, the model is also independent of parameters from exhausting test series. This thesis will explore the potential of this model by implementing a version of it using the scripting interface of Abaqus FEA, which further will be subjected to parameters studies and case studies. This version of Yankelevskys model will take orientations of the flaws into account and use a geometry correction factor corresponding to the flaws depth to length ratio.

1.2 Objective

The objective of this thesis is to understand and model the effect of surface flaws on the strength of float glass by implementation of a strength model for brittle materials, based on failure from surface flaws and independent of test data calibration. This model should further be verified, and used to predict strength of glass based on some given initial configuration. The subobjectives are:

1. the establishment of the mathematical foundation,
2. the implementation of the model in MATLAB, acting as an analytical solution,
3. the implementation of the model in Python as an Abaqus FEA post-processing routine,
4. the implementation of the model in FORTRAN as an Abaqus FEA subroutine,
5. a verification process with the analytical solution,
6. a parameter study
7. and a case study.

1.3 Scope

The scope of the thesis is confined to the following items:

- Only soda-lime-silica float glass is considered.
- The flaw density if uniform. This is a simplification which excludes shielding and amplifying effects of stresses from close positioning of the flaws.

1.4 Overview of thesis

This thesis is divided into chapters presenting the development of the glass strength model in chronological order. Following is a short description of each of these chapters:

Chapter 2: Theory Chapter 2 presents the theoretical foundation needed to fully understand the concepts and results in this thesis.

Chapter 3: Implementation Chapter 3 describes the numerical model and the implementations of it in the three programming languages MATLAB, Python and FORTRAN. The MATLAB code was preliminary work and serves as an analytical solution, while the Python code is part of the post-processing scripting interface of Abaqus, and finally the FORTRAN code is part of the subroutine interface of Abaqus. The main implementations, the MATLAB and Python codes, are described by presenting the overall framework, algorithms, asymptotic running time and pseudo codes. All source codes are included in the Appendix.

Chapter 4: Verification Chapter 4 first presents the preliminary controls taken in developing the code. Thereafter, the MATLAB code is verified as an analytical solution and further used to verify the Python and FORTRAN codes by comparison of results.

Chapter 5: Parameter studies In Chapter 5 parameter studies regarding plate geometry and flaw geometry is performed with the Python program. The resulting distributions are fitted with Normal and Weibull distributions, examining which is the best fit.

Chapter 6: Case studies Chapter 6 compares four point bending tests carried out in a laboratory by [7] with results from the model implementation. Further, the models behavior exposed to a blast wave is examined. The model results derive from the Python implementation. At last a short study for visualization is performed with the FORTRAN code.

Chapter 7: Concluding remarks Chapter 7 presents a summary of the results from this thesis with concluding remarks.

Chapter 8: Future work Chapter 8 discusses suggestions for future development of the strength model and the software implementations developed in this thesis.

Chapter 2

Theory

2.1 Fracture Mechanics

In this section linear elastic fracture mechanics relevant to the thesis is presented. The following is adapted from [8], unless otherwise stated. For a more comprehensive description, please refer to this source.

2.1.1 Strength of brittle materials

The theoretical cohesive strength of a material is related to the energy E required to break the attractive bonds between the atoms in the material. The bonding energy between molecules is be described by:

$$E_b = \int_{x_0}^{\infty} P dx \tag{2.1}$$

where P is the force required to separate the atoms and x_0 is the equilibrium spacing, the distance between two atomic nuclei at which the potential energy is at minimum. Further, the force-displacement relationship for breaking interatomic bonds, assuming small displacements, is given by:

$$P = P_c\left(\frac{\pi x}{\lambda}\right) \tag{2.2}$$

where P_c is the cohesive strength of the atomic bonds and λ is the approximate distance required for the to break the atomic bonds. Combining Equations (2.1) and (2.2) and assuming λ is approximately equal to the atomic equilibrium spacing, the theoretical cohesive stress of a material can be estimated as

$$\sigma_c \approx \frac{E}{\pi} \tag{2.3}$$

It is however a documented fact that the actual strengths for brittle materials is up to four orders of magnitudes lower than the theoretical strength. Stress concentrates around flaws in the material, and thus the flaws are responsible for stresses exceeding the strengths of the atomic bonds. These stress concentrations correlates to the size of the flaw. The founder of fracture mechanics, A.A.Griffith, explained this by the laws of thermodynamics. A flaw will grow if and only if the release of potential energy is greater than or equal to the surface energy needed for the crack to grow. The energy balance in increasing the surface area by dA is described by the equilibrium equation

$$-\frac{d\prod}{dA} = \frac{dW_s}{dA} \tag{2.4}$$

$\prod$ represents the potential energy from external loads and internal strain and W_s is the applied work in creating new surfaces.

For the case of a flaw of length $2a$, where the length is much greater than the width, for an infinitely wide plate it can be shown that

$$\prod = \prod_0 - \frac{\pi\sigma^2 a^2 B}{E} \tag{2.5}$$

$\prod_0$ represents the potential energy of an uncracked plate, a the half-length of the flaw, σ the remotely applied stress and B is the thickness of the plate.

For the through-thickness flaw, the work required to create new surfaces is given by

$$W_s = 4aB\gamma_s \tag{2.6}$$

γ_s is the surface energy of the material. The results from combining Equations (2.4), (2.5) and (2.6) are

$$-\frac{d\Pi}{dA} = \frac{\pi\sigma^2 a}{E} = G \tag{2.7}$$

and

$$\frac{dW_s}{dA} = 2\gamma_s \tag{2.8}$$

G is the energy release rate, defined as the energy available for an increment of crack extension. Combining Equations (2.4), (2.7) and (2.8) the strength of a flaw of length $2a$ in an infinitely large plate stands:

$$\sigma_f = \sqrt{\frac{2E\gamma_s}{\pi a}} \tag{2.9}$$

where σ_f is the fracture stress, perpendicular to the major axis of a flaw.

2.1.2 Stress intensity factor

As covered in Section 2.1.1 failure for brittle materials almost always begins with stress concentrating around flaws in the material. The stress intensity factor K is a measure of the stress state near the tip of a flaw subjected to stresses. Different modes of loading leads to different values of the stress intensity factor.

There are three modes of loading: mode I which is caused by stress perpendicular to the flaw, mode II caused by in-plane shear stress and mode III caused by out-of-plane shear stress. Most relevant for float glass is mode I as it requires the least energy, and thus is the only mode focused on in this thesis. Figure 2.1 illustrates the three modes.

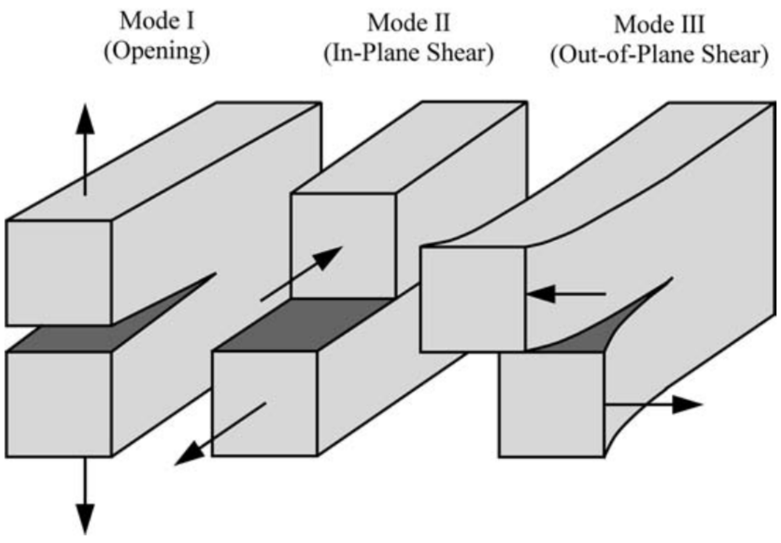


Figure 2.1: The modes of fracture, retrieved from [8].

The general expression for the mode I stress intensity factor K_I for a finite crack, where the plate geometry is much greater than the flaw geometries, is

$$K_I = \lambda_s \sigma \sqrt{\frac{\pi a}{Q}} f(\phi) \tag{2.10}$$

where

$$Q = 1 + 1.464\left(\frac{a}{c}\right)^{\frac{33}{20}}$$
$$\lambda_s = \left[1.13 - 0.09\left(\frac{a}{c}\right)\right][1 + 0.1(1 - sin\phi)^2] \tag{2.11}$$
$$f(\phi) = \left[sin^2(\phi) + \left(\frac{a}{c}\right)^2 cos^2(\phi)\right]^{\frac{1}{4}}$$

Q is a flaw shape parameter and λ_s is the surface correction factor. c and $2a$ represents the depth and length of the flaw, respectively, as illustrated in Figure 2.2(a), while ϕ is defined in Figure 2.2(b). The expression of K_I can be simplified into

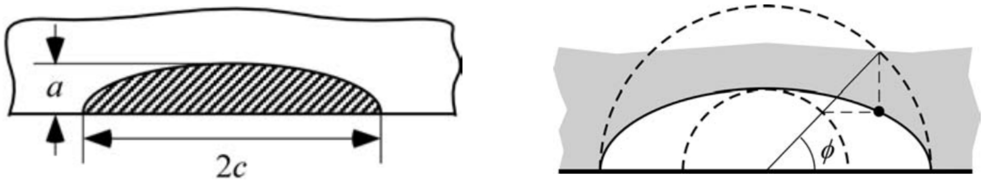
$$K_I = Y\sigma\sqrt{\pi a} \tag{2.12}$$

where Y is a geometrical correction factor, depending on the shape of the crack and its position on the plate. When a flaw lies in an infinitely wide plate, $Y \approx 1$ [9]. Combining this and Equations (2.9) and (2.12), the critical stress intensity factor can be written

$$K_c = \sqrt{2E\gamma_s} \tag{2.13}$$

This holds for all flaws exposed to a single mode loading, and failure will therefore occur when

$$K = K_c \tag{2.14}$$



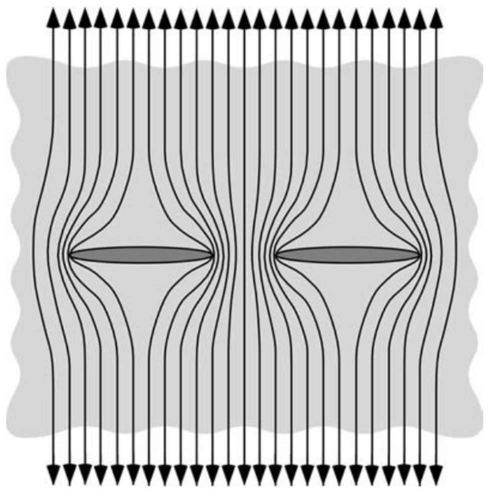
(a) Surface flaw. (b) Angle parameter of surface flaw.

Figure 2.2: Parameters for Equation (2.11). Illustrations retrieved from [8].

2.1.3 Strength-influencing properties of flaws

As stated above, presence of flaws affects the strength greatly for brittle materials. Equation (2.9) shows that an important strength-affecting property of flaws is the size. However, the flaws orientations, density of flaws and relative positions are also relevant properties.

As the stress critical for mode I failure is the resultant stress normal to the major axis of the flaw[10], the orientations of the flaws affect the strength. Consequently different types of loading have different impacts on strength. As an example, it is more likely that positive biaxial load will cause failure in a glass plate than a positive uniaxial load of the same magnitudes. This is because the resultant stresses normal to the flaw length are more likely to be larger for the biaxial loading. For a glass plate subjected to bending, the properties of the flaws on the compression side becomes irrelevant, as only flaws in tension will fail.


(a) Amplifying effect of coplanar flaws.

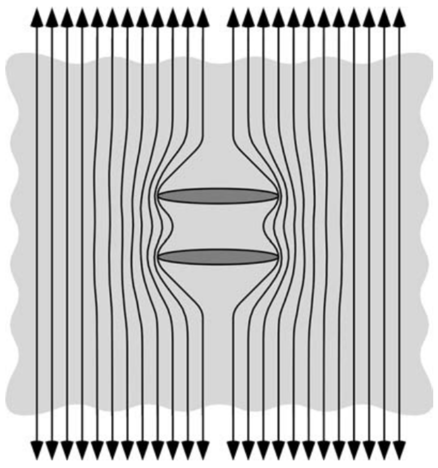

(b) Shielding effect of parallel flaws.

Figure 2.3: Stress concentrations around flaws. Illustrations retrieved from [8].

Further, the density of the flaws also impacts strength. The likelihood of failure increases with the number of flaws. This applies to the span of the plate as well. With a large plate the number of flaws will be correspondingly high, and the probability for a dominant flaw present is increased.

The relative positions of the flaws can either decrease or increase strength. Figure 2.3(a) illustrates the interaction of coplanar flaws. The stress concentrations will increase and the stress perpendicular to the flaws will limit to a value of a flaw with the length of both flaws.

Figure 2.3(b) illustrates the interaction of parallel flaws with the result of decreased stress concentrations around each flaw. The parallel flaws have a shielding effect on one another.

2.2 Statistical Theory

In this section statistical theory relevant to this thesis is presented. This includes a short presentation of the Monte Carlo method, a traditional treatment for strength and the weakest link theory.

2.2.1 Monte Carlo method

This section is adapted from [11]. The Monte Carlo method is an umbrella term for a wide range of stochastic techniques using random sampling to solve numerical problems. The problems are often complex where results are easier obtained by such a method of approximation. Results are often obtained by large scale sampling for a close approximation. The compositions of Monte Carlo algorithms varies, although they typically follow the pattern below, as given by [11]:

1. Define a domain of possible inputs.
2. Generate inputs randomly from a probability distribution over the domain.
3. Perform a deterministic computation on the inputs.
4. Aggregate the results.

2.2.2 Traditional statistical treatment of strength of brittle materials

The theory in this section is adapted from [4]. The Weibull distribution is widely used to represent the statistics of brittle failure. There are two common Weibull distributions; the two and three-parameter distributions. Only the cumulative two-parameter distribution is presented in this section. It is given by:

$$P_f(\sigma) = 1 - \exp\left[-\left(\frac{\sigma}{\sigma_0}\right)^m\right] \tag{2.15}$$

where σ is the stress and σ_0 is the Weibull scale parameter. σ_0 is the 63rd percentile, meaning that the probability of failure occurring at or below the stress σ_0 is 0.63. m is the Weibull modulus, and is an inverse of the distribution width. This means that when the Weibull distribution is used to describe the material strength, the scatter will increase with a decreasing m.

It is often useful to relate the probability of failure for specimens of different geometries. This is traditionally done by using a scale parameter of unit volume Σ_0. This is given by

$$\sigma_0 = \Sigma_0 V^{\frac{-1}{m}} \tag{2.16}$$

where V is the volume of the specimen. It should be noted that both Σ_0 and m are assumed materials constants.

The probability of failure can also be related to specimens subjected to different types of loading. This can be done by employing a loading factor k. With σ_{max} as the maximum stress anywhere in the specimen, the cumulative probability is then given by

$$P_f = 1 - \exp\left[-kV\left(\frac{\sigma_{max}}{\Sigma_0}\right)^m\right] \tag{2.17}$$

The reader is referred to [4] for more details.

2.2.3 Weakest link theory

This section is an adaption from [4]. The principle of the weakest link theory is that the occurrence of one failing flaw will lead to total failure in the entire specimen of a brittle material. The theory is also based on a homogeneous material in that the flaws are distributed throughout the volume of the material. By considering a material of a large number of elements n, each of volume δV which is subjected to a stress σ, the probability for the ith element failing is denoted $P_{f,i}(\sigma, \delta V)$. This probability of failure is equal for every element in the specimen, thus the probability for failure for the specimen becomes:

$$1 - P_f(\sigma, V) = \left[1 - P_f(\sigma, \delta V)\right]^n = \left[1 - \frac{V}{n}\frac{P_f(\sigma, \delta V)}{\delta V}\right]^n = \left[1 - \frac{V}{n}\phi(\sigma)\right]^n \tag{2.18}$$

where it is assumed that $P_f(\sigma, \delta V)/\delta V$ limits to $\phi(\sigma)$ with a growing n. As n approaches infinity and δV approaches zero, the probability of failure becomes:

$$P_f(\sigma, V) = 1 - \exp[-V\phi(\sigma)] \tag{2.19}$$

The argument of the weakest link does not specify any form for $\phi(\sigma)$, but Weibull assumed

$$\phi(\sigma) = \left(\frac{\sigma}{\Sigma_0}\right)^m \tag{2.20}$$

Combining Equations (2.19) and (2.20) yields the two-parameter Weibull distribution:

$$P_f = 1 - \exp\left[-V\left(\frac{\sigma}{\Sigma_0}\right)^m\right] \tag{2.21}$$

2.3 Thin plate theory

This section will state the definition of a thin plate and provide tools for calculating the stresses in simply supported and clamped plates under this assumption. Thin plate theory is generally applicable for float glass used as windows. If

- the thickness of a plate is 1/20th of its length and width, from now referred to as x and y-direction,
- the plate is homogenic, isotropic and linearly elastic,
- and the load is normal to the plate plane,

thin plate theory can be applied[12]. For thin plates, the stresses in the thickness direction z are virtually of zero magnitude. By Hooks law for plane stresses, the strains are given as:

$$\begin{aligned}\epsilon_x &= -z\frac{\partial^2 w}{\partial x^2} = \frac{1}{E}(\sigma_x - \nu\sigma_y)\\ \epsilon_y &= -z\frac{\partial^2 w}{\partial y^2} = \frac{1}{E}(\sigma_y - \nu\sigma_x)\\ \gamma_{xy} &= -2z\frac{\partial^2 w}{\partial x\partial y} = \frac{2(1+\nu)}{E}\tau_{xy}\end{aligned} \tag{2.22}$$

The strains in terms of the thickness direction are equal to zero. The stresses are:

$$\begin{aligned}\sigma_x &= -\frac{zE}{1-v^2}\left(\frac{\partial^2 w}{\partial x^2} + v\frac{\partial^2 w}{\partial y^2}\right),\\ \sigma_y &= -\frac{zE}{1-v^2}\left(\frac{\partial^2 w}{\partial y^2} + v\frac{\partial^2 w}{\partial x^2}\right),\\ \tau_{xy} &= -\frac{zE}{1-v^2}(1-v)\frac{\partial^2 w}{\partial x\partial y}.\end{aligned} \tag{2.23}$$

where E is the Young's modulus, v is the Poissons ratio and w is the transverse deformation field. w is calculated by applying boundary conditions to the plates differential equation. For a plate subjected to a uniform load q_0 the plates differential equation is given by

$$\frac{\partial^4 w}{\partial x^4} + 2\frac{\partial^4 w}{\partial x^2\partial y^2} + \frac{\partial^4 w}{\partial y^4} = \frac{q_0}{D} \tag{2.24}$$

where the plate stiffness D can be calculated by [13]:

$$D = \frac{Eh^3}{12(1-v^2)} \tag{2.25}$$

In the following subsections the plate equation w for the simply supported and clamped plates are given.

2.3.1 The clamped plate

Applying the boundary conditions applicable for a clamped plate,

$$\begin{aligned} w &= 0, \\ \frac{\partial w}{\partial n} &= 0, \end{aligned} \tag{2.26}$$

to Equation (2.24) yields

$$w(x,y) = \sum_{m=1}^{M}\sum_{n=1}^{N}(1 - cos(2m\pi x/a))(1 - cos(2n\pi y/b))\,w_{mn} \tag{2.27}$$

where a and b are the lengths in x and y-direction, respectively, and w_{mn} are coefficients found by minimizing the potential energy of the system. For the full procedure the reader is referred to [14].

2.3.2 The simply supported plate

[15] provides the plate equation for simply supported plates. Applying the relevant boundary conditions,

$$\begin{aligned} w &= 0, \\ \frac{\partial^2 w}{\partial x^2} &= 0, \\ \frac{\partial^2 w}{\partial y^2} &= 0 \end{aligned} \tag{2.28}$$

to Equation (2.24) yields

$$w(x,y) = \sum_{m=1}^{M}\sum_{n=1}^{N} sin(m\pi x/a)sin(n\pi y/b)\,w_{mn} \tag{2.29}$$

where w_{mn} are given by

$$w_{mn} = \begin{cases} \dfrac{16q_0}{Dmn\pi^6\left(\frac{m^2}{a^2}+\frac{n^2}{b^2}\right)^2}, & \text{if } m \text{ and } n \text{ are odd.} \\ 0, & \text{otherwise.} \end{cases} \tag{2.30}$$

2.4 Mechanical Properties of Glass

Several types of glass exist, and they are commonly a compound of silica (SiO_2)[16]. Two common types of glass, with composition and typical use are given in Table 2.1.

Table 2.1: Common types of glass[17]

Glass	Typical composition (wt%)	Typical uses
Soda-lime glass	79 SiO_2, 10 CaO, 15 Na_2O	Windows, bottles, etc.; easily formed and shaped
Borosilicate glass	80 SiO_2, 15 B_2O_3, 5 Na_2O	Pyrex; cooking and chemical glassware; high-temperature strength, low coefficient of expansion, good thermal shock resistance.

Under tension, glass typically has a strength of 10-100 MPa[4]. Mechanical wear of glass stem from local stresses from contact and sometimes chemical reactions. Since fracture of glass is nearly always initiated from flaws at surface level[18], different techniques of polishing can be applied to minimize flaws. These polishing techniques can be chemical, for instance as surface crystallization with a maximum strengthening factor of 17, or thermal, for instance fire polishing with a maximum strengthening factor of 200[4]. This thesis focuses on unpolished soda-lime glass often used for windows.

2.4.1 The float process

This section is an adaptation from [19], unless otherwise stated. Float glass is produced by pouring molten glass over a tin bath, ensuring uniform thickness. Once the float glass is congealed the annealing of the glass begins. This is a a controlled cooling for a uniform contraction of the glass' inner and outer sections, in order for the residual internal stresses to be relieved[20].

Before shipping the glass off to the customer the glass is cut into a size suitable for stacking and shipping. These plates are jumbo sized, usually 6×3.21 m^2, and will be referred to as the basic plates in this thesis.

During the production process the glass is exposed to mechanical wear creating surface flaws on the glass. For that reason the basic plates undergoes a control of flaw lengths, such that only plates with flaws with a maximum length is shipped to the customer. In the European standard this maximum length is 0.2 mm[21].

2.5 Blast Mechanics

When dimensioning with respect to blast loadings, it is typically with respect to the particular scenario of a high explosive detonation. Of the three principal effects of this scenario; the blast overpressures, fragment generation and shock loads from the shock wave, the blast overpressures are generally the governing factor to the structural response. This section presents the phenomena of blast pressure originating from an explosive detonation and is an adaption from [22], unless otherwise stated.

The expanding gaseous product of the sublimation of an explosion creates a shock wave propagating outwards, by the surrounding air being compressed and trying to equalize. This layer contains pressure energy and can be damaging to solids within range. In the spherical expansion of the wave, the pressure will equalize towards the air due to divergence and will also decrease due to heating of the surroundings. The expansion causes the wave to decrease in strength, velocity and lengths in duration, as Figure 2.4 illustrates. Eventually the shock wave reaches equilibrium with the air.

Almost instantly after an explosion, the peak incident overpressure P_{so} is reached, before the pressure decays exponentially over a time period t_+. Following a peak overpressure comes a usually longer time period t_- of negative pressure, i.e. pressure lower than ambient pressure P_a. Figure 2.5 illustrates this with the curve peaking at P_{so}.

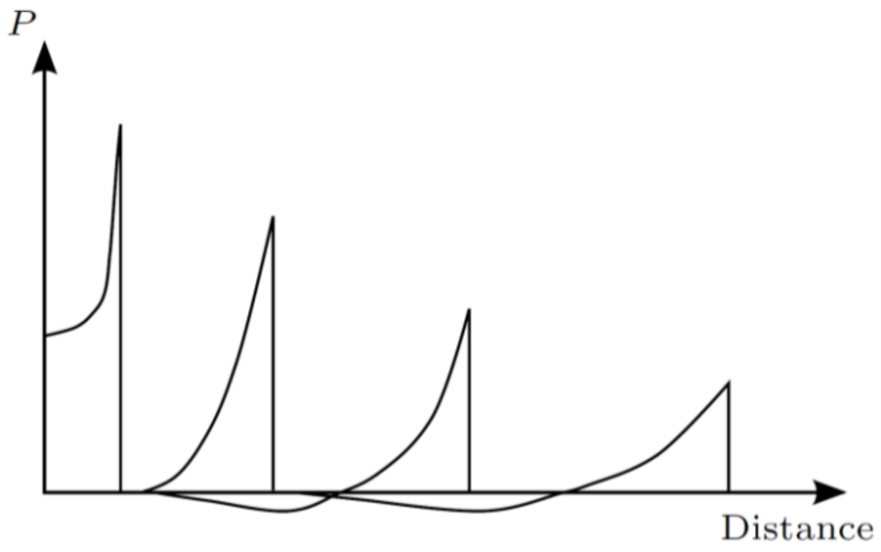


Figure 2.4: Influence of distance on the blast pressure, retrieved from [22].

A free field explosion with no nearby surface interaction is called the Friedlander waveform[23]. Once the blast wave travels a distance and interacts with a non-parallel surface, it will be reflected, always with a peak greater than the incident pressure at the same distance from the explosion. This reflected pressure determines the loading of a structure due to an explosion. Figure 2.5 illustrates the behavior of reflected pressure with a peak P_r. The Friedlander equation is frequently used to describe the reflected pressure and is given by:

$$P(t) = P_a + P_r\left(1 - \frac{t}{t_+}\right)\exp\left(\frac{-bt}{t_+}\right) \tag{2.31}$$

where b is the exponential decay coefficient.

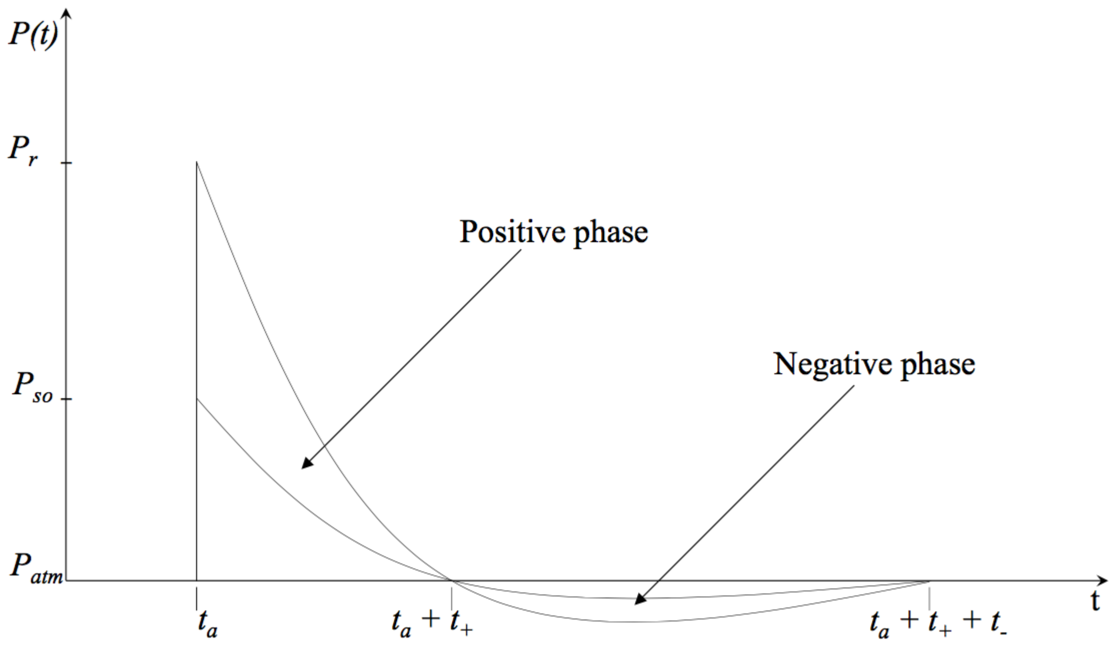


Figure 2.5: Characteristic pressure-time history for the incident blast wave from an explosion, retrieved from [22].

Chapter 3

Implementation

In this chapter the strength model adapted from [6] is thoroughly presented. First the mathematical background of the strength model is introduced, followed by a description of the general design used in the software implementations. The software implementations are generated in MATLAB, Python and FORTRAN, each with its own purpose and advantage, exploring the potential of the strength model. The final part of this chapter describes these programs in turn. The main implementations, the MATLAB and Python codes, are presented along with their limitations, optimizations, pseudo codes and asymptotic running times.

3.1 Description of the model

In 2014, Yankelevsky proposed a new model for strength prediction in glass under external loads. Traditional statistical strength prediction methods for glass uses parameters gained from experiments. As depicted in Section 2.2.2, the traditional Weibull distribution requires two parameters obtained from experiments, making the method potentially incorrect for non-tested geometries and loads, and a conversion must be used to relate the results to other loads and geometries. The model presented by Yankelevsky differs from this procedure in its independence of test series data. The idea is to predict the probabilistic strength instead of adopting a model based on specific series of experimental data.

According to the weakest link theory for brittle materials described in Section 2.2.3, failure for one surface flaw yields failure in the whole specimen. Failure of a flaw occurs when condition given by Equation (2.14) is fulfilled. Yankelevskys' model includes the properties of the flaws over the glass surface as input variables to check Equation (2.14) for every flaw under a certain load. The locations of the flaws over the plate surface will hereafter be referred to as the flaw map. No established models today provide information about the fracture origin, a feature this flaw map provides.

Surface cracks in glass originate from mechanical wear[24] and chemical exposure[25]. Over time there will be more and larger flaws, weakening the glass. Subsection 2.4.1 describes how freshly produced glass acquires cracks from mechanical stress from the handling after the event of solidification. This model examines only glass straight out of the production process and the flaws thereof.

This section is mainly an adaptation of [6], if not otherwise stated.

3.1.1 Generating the flaw map

For the general case the length and orientation of the flaws need to be randomized, thus the problem becomes one of statistics. In this regard, the following assumptions are made:

1. The maximum length of the flaws is 200 μm.
2. For every cm^2 on the surface area there is one flaw.

The flaw map is generated for a float glass of a large scale, the basic plate, which is later cut into the desired sized plates intended for structural purposes, as described in Section 2.4.1. Assumption 1 is justified by the fact that manufactured soda lime float glass undergoes controls of maximum flaw length, usually 100-300 μm[6]. These controls are for the basic plates, i.e. the product the manufacturers ship to customers. The second assumption is such that the density does not cause the flaws to interact with each other. This also complies with [26] which suggests there are 1.18 to 2.6 flaws per cm^2 in soda lime annealed glass. With respect to the distribution of flaw sizes, more assumptions are made:

3. There is only one flaw of maximum size per basic plate.

4. The smaller the flaw size, the higher the prevalence of flaws of this size.

Points 3 and 4 are included in the model through Equation (3.1)

$$\frac{N_f}{N_0} = e^{\frac{\delta}{\mu}} \tag{3.1}$$

where

N_0 is the number of flaws in the plate,

δ is size of a flaw in the plate,

N_f is the number of flaws in the plate less than or equal to the size of δ, μ is called the characteristic flaw size, a parameter found by setting and $\delta = \delta_{max}$, which only one flaw fulfills, ie. $N_f = 1$, and further $\delta = 0$ which all flaws fulfills, ie. $N_f = N_0$. By substituting these conditions into Equation (3.1) and reordering the terms, the characteristic flaw size is determined by the following equation:

$$\mu = \frac{\delta_{max}}{\ln(N_0)} \tag{3.2}$$

Figure 3.1 illustrates the exponential relationship described by Equation (3.1) between the occurrences of different flaw sizes, by an example of a plate with 100 flaws.

Further, every flaw is assigned a random orientation θ between 0 and π. The stress parallel with the flaw is given by [27]:

$$\sigma_{x1} = \frac{\sigma_{xx} + \sigma_{yy}}{2} + \frac{\sigma_{xx} - \sigma_{yy}}{2} \cos\left(2\theta\right) + \tau_{xy} \sin\left(2\theta\right) \tag{3.3}$$

However, Mode I fracture occurs when the stress σ in Equation (2.12) is normal to θ. By referring to this σ as σ_{y1} for now, the sum of σ_{x1} and σ_{y1}, and σ_{xx} and σ_{yy} should be the same. Combining this relation and Equation (3.3) the equation for the stresses perpendicular to the flaw becomes:

$$\sigma_{y1} = \sigma_{xx} + \sigma_{yy} - \frac{\sigma_{xx} + \sigma_{yy}}{2} + \frac{\sigma_{xx} - \sigma_{yy}}{2} \cos\left(2\theta\right) + \tau_{xy} \sin\left(2\theta\right) \tag{3.4}$$

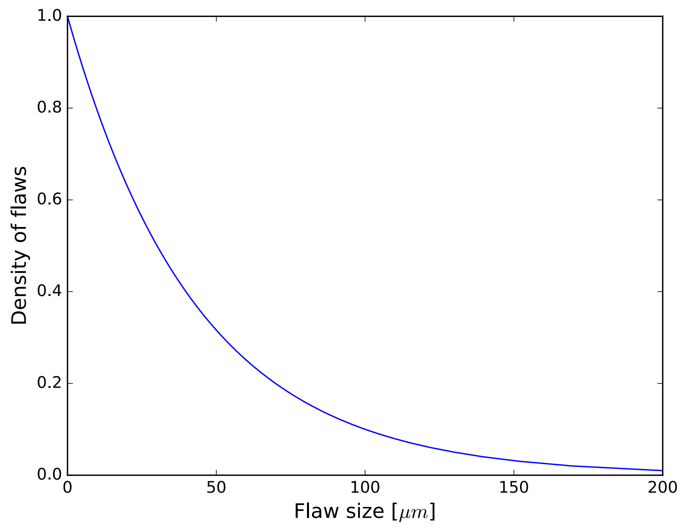


Figure 3.1: The prevalence of flaws with length equal or less than some length, for a plate with 100 flaws.

3.1.2 The analysis procedure

In order to find the strength of a glass plate a Monte Carlo method is used. In accordance with the step-by-step Monte Carlo procedure given in Section 2.2.1, the analysis procedure becomes the following:

1. The domain of inputs for this method are glass plates which are identical in terms of ge-

ometry, material properties, boundary conditions and loading.

2. Other inputs to be generated are the flaw maps, as described in Section 3.1.1. The flaws are distributed uniformly over basic plates with a density of one flaw per square centimeter. The basic plates with their flaw maps are divided into plates fitting the domain.

3. With a plate subjected to a given load, every flaw in the plate is checked for failure with Equation (2.14). This is repeated for different loads generated according to a search method, until there is only one single failing flaw in the plate. Further, this is repeated for every plate in the domain.

4. The probability distribution for the critical stresses and other parameters of interest are eventually obtained.

3.2 General design

The model has been implemented in three programming languages, each language having a different purpose. The MATLAB code is introductory work with analytical solutions for a few simple cases, chosen on the basis of it being a simple programming language for procedural programming. The finite element analysis software Abaqus FEA is used for numerical simulations of the chosen problems, where the glass response in terms of the stresses is retrieved. A Python script is used for the post-processing of the results, as it is compatible with Abaqus FEA. Abaqus FEA enables more complicated loading scenarios, and the Python code is therefore more versatile than the MATLAB code. Finally, FORTRAN was used for creating an Abaqus subroutine visualizing crack propagation.

In the following sections the model implementations are explained in detail. All implementations uses the material properties given in Table 3.1. Initially, the geometry correction factor Y was chosen to be 0.6625, which corresponds to a half-penny shaped flaw, i.e. where the depth is half the length. This geometry correction factor was found by setting ϕ equal to $\frac{\pi}{2}$ in Equation 2.11. This geometry correction factor differs from the one used by Yankelevsky, however the same depth and length ratio was chosen.

Table 3.1: Material properties[28] [29]

Module of Elasticity	70000 MPa
Poissons ratio	0.22
Density	2.5E-09 $\frac{tons}{mm^3}$
Critical toughness	0.75 MPa m$^{\frac{1}{2}}$

A simplification occurring in all implementations is the basic plate. Usually the basic plate is constructed in the jumbo size of 6×3.21 m^2, but for simplicity the area is scaled uniformly similarly to the plate to be analyzed, though in a greater dimension approximate to the jumbo size. This way the basic plate area is comprised of a whole number of the plate area in question, has the same shape and the prevalence of maximum sized flaws is approximately the same as with the 6×3.21m^2 plate. Figure 3.2 illustrates the proportions of a quadratic implemented basic plate (red) and the general jumbo sized plate (blue).

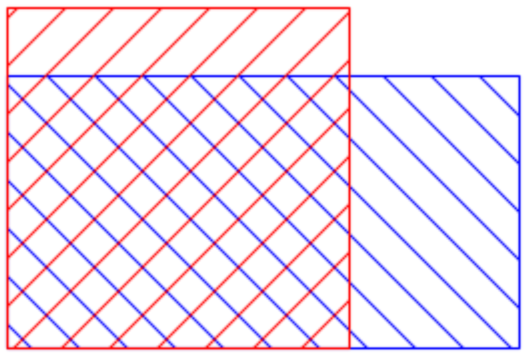


Figure 3.2: Demonstration of the proportions of a jumbo sized plate of 6×3.21m^2 and an implemented basic plate used in a analysis of 4×4m^2.

3.3 MATLAB implementation

The MATLAB code consist of analytical solutions of three distinct combinations of loads and boundary conditions. These are uniform loads perpendicular to simply supported and clamped plates, and in-plane uniform load on the sides of a plate. In this section these solutions will be referred to as cases.

3.3.1 Limitations

The MATLAB code calculates the stresses in the plate, requiring the displacement function of the particular combination of load and boundary conditions, see Sections 2.3.2 and 2.3.1. Serving as an analytical solution with the purpose of verifying the other codes, it is considered sufficient to limit the application to three cases.

The stresses for the two most complex cases, the simply supported and clamped plates, are calculated on the basis of thin plate theory, as given in Section 2.3. The geometry of the plates must be in accordance with the requirements such that this theory can be applied. Since these plates' stresses are approximated through series expansion, some deviation from analytical solution is expected.

The implementation only handles linear behavior and thus disregards nonlinear behavior such as geometry nonlinearities from large displacements.

3.3.2 Main structure

This section describes the structure of the MATLAB implementation, rounded off with the simplified pseudo formulation, given in Algorithm 1. The reader is referred to Appendix A for the whole source code.

The code loops through a number of basic plates and generates two individual flaw maps for each of these, one for each side of the plate to account for possible failure at both sides. Each basic plate is cut into smaller plates which are further looped through. Each of the smaller plates are applied with load in a while loop that will continue until the point where only a single flaw fails for the plate examined. To find this point the critical load search method, described in Section 3.3.3, is used. Within the while loop every flaw in the current plate is looped through, checking whether this flaw will fail for the current load or not.

Eventually there is only one flaw that fails. However, the load for which this flaw failed might not be its critical load. Therefore the flaw is subject to a convergence criterion to approximate the lowest load it will fail for. This approximation also uses the critical load search method. As convergence has occurred according to the criterion, information about the failing flaw is saved for the Monte Carlo analysis, and the next plate is examined.

After all plates are accounted for the analysis writes information about the critical stresses, locations of failure and deflections to a text-file for Monte Carlo post-processing.

Algorithm 1: Pseudo formulation of the main structure of the MATLAB code.

```
define initial conditions;
for all elements/flaws in plate do
    calculate stresses and store in matrix;
end
for all basic plates do
    generate flaw map;
    for all plates cut out of one basic plate do
        retrieve flaws for this plate from flaw map;
        set guessing value for load;
        while there not only a single failing flaw do
            for every relevant flaw in the plate do
                retrieve stresses from matrix;
                check for failure;
            end
            if there is not only one failing flaw then
                apply critical load search method for a new loop;
                update relevant flaws in the plate;
            else
                while the critical load is not converged do
                    apply critical load search method until the failing flaw is converged;
                end
                register the first failing flaw;
            end
        end
    end
end
```

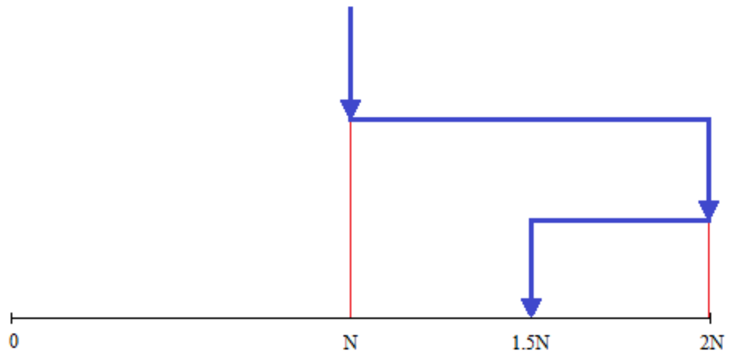
3.3.3 Critical load search method

The MATLAB implementation of the model benefits from a customized search algorithm for finding the critical load. With an average performance of $\theta(\log_2 n)$, the binary search algorithm[30] was chosen. The algorithm, adapted to finding the critical load, is in this section explained and further formulated in pseudo code in Algorithm 2.

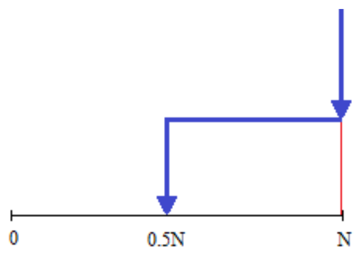
For the first plate, the critical load is estimated. If there is no failure, the loading level is doubled until there is failure. When failure occur, the load is set down half the difference between the current and the last tested loading level. The idea is to find the critical stresses by testing the mean between the lowest stresses causing failure and the highest stresses with no failing events. Eventually this method will converge to the critical point of a single failing flaw.

Figure 3.3(a) illustrates the algorithm in a situation where no failure occurs at the initial load, such that the load is doubled and failure occurs. The load is further set to the mean of the non-failure and failure inducing loads.

If failure occurs at the initial guessed value the load is halved, as demonstrated in Figure 3.3(b).



(a)



(b)

Figure 3.3: Illustration of the binary search algorithm searching for a given value.

Algorithm 2: Pseudo formulation of the critical load search method.

```
retrieve array of current applied load, the previous applied load and lowest applied load
  with failure registered, and information if the flaw failed for current load;
if failure for current load then
    if this is the first tested load then
        half the load;
    else
        if the current load is less than the previous then
            half the load;
        else
            subtract the load with half the difference of current load and previous load;
        end
    end
else
    if this is the first tested load then
        double the load;
    else
        add to current load half the difference between lowest failing load and current
          load;
    end
end
```

3.3.4 Optimization

Depending on the size of the plate and number of plates in the Monte Carlo analysis, the number of computations is potentially large. This section presents some cost-saving measures taken to optimize the code. One of these measures were applying a search method for finding the critical load, see Section 3.3.3. Other measures are listed below:

- When there are failure in multiple flaws for a load, the failed flaws are registered and are the only flaws assessed from here on, as they are singled out as candidates for the first failing flaw. This is applicable to the MATLAB code since the solutions are behaving linearly.

- When finding the critical stresses for one plate, the process starts with a hard-coded estimated value for the load. This estimate is the same for every geometry implemented, although the value might be far from the critical load. In order to speed up the process of finding the critical loads for the plates, it is assumed the load is approximate to the critical load q_{crit} of the previous plate in the loop. The inital estimated load for the current plate is set to $q_{crit} * 1.3$ in an attempt to avoid the guessing value resulting in no failure. Thus, at the first run for the current plate the guessing value is likely to result in some failed flaws, eliminating other flaws as candidates for first failing flaw. Also, since the value is within a close range of the likely critical load, the number of candidates will be accordingly low. This optimization affect all but the first plate in the loop.

- For two of the cases, the simply supported and the clamped plate, the stresses are not uniform over the plate and needs to be calculated at the point of every flaw. For every load tested in the analysis, the plate equation and the stresses need to be recalculated, which is time consuming. The plate equations for the simply supported and clamped plates is given in Equation (2.3). w_{mn} is the only coefficient depending on the load q_0, and since the x and x-coordinates remain constant for every flaw, the plate equation is linear for a flaw. Separating w_{mn} from the expression, the remains of the expression can be calculated prior to process of the main structure, see Section 3.3.2. Thus computations are saved and only w_{mn} needs recalculation for every load on each plate.

3.3.5 Asymptotic running time

Analysis of the asymptotic run time of the MATLAB implementation yields a worst case scenario of

$$\mathcal{O}\left(P \log_2\left(\frac{q_f}{TOL}\right) e\right) \tag{3.5}$$

where P represent the number of plates, q_f the highest guessed load inducing failure, TOL the tolerance of the convergence criteria and e the number of flaws in a plate.

3.4 Python implementation

The Python program differs from the MATLAB program by the stresses being retrieved from the output database of an Abaqus simulation. This allows for flexibility in terms of geometry, boundary conditions and loads. The requested stress data lies in an .odb-file which Python reads with an imported odbAccess package[31]. Abaqus generates this data for a specified number of time frames, and desired data in between these time frames needs to approximated. This approximation is done linearly. Thus, the more time frames, the more accurate the results for combinations of boundary conditions and loads causing nonlinear behavior. For linear behavior, the approximation between time frames will provide identical results regardsless of the number of time frames.

The Python code is general and will handle linear as well as nonlinear behavior. The outline of this section is for the nonlinear case as that angle will provide a better understanding for the structure of the code. However, as glass can behave linearly until the point of failure, the code has potential to run significantly faster, and therefore a description of handling these cases is provided in Section 3.4.4.

3.4.1 Limitations

Some knowledge of the expected value and variance is beneficial when simulating the problem in Abaqus to make sure all plates will fail for the maximum load. If not the analysis will be incomplete by not including the plates failing for the highest loads.

With respect to time it is also of great benefit to limit the maximum load in Abaqus, as the critical load search method in Section 3.3.3 is not effective in the Python code. This is due to the time consumption related to reading field data from the Abaqus output database. This is further elaborated in Section 3.4.3.

The current version of the Python code does not output the critical load, as loads can be applied in a number of ways and the code is general. It will however store the percentage closeness the first failing flaws are to their non-failure time steps. Using the Abaqus simulation to obtain the loads at the time steps, the critical load can be calculated in the Monte Carlo post-processing of the results. An example code for this is included in at the end of Appendix B.

As stresses are retrieved at the time steps of the numerical analysis, an interpolation of the stresses over the time steps is required to find the critical stresses. As results from nonlinear behavior improves with higher time frame density, it is assumed the time steps are set so close that linear interpolations of the stresses are sufficient.

Since the flaws lie in the center of every surface square centimeter, a simplification requiring either a 1×1 cm^2. The script also allows a fine 2×2 mm^2 mesh by retrieving the stresses of the center element in every square centimeter, as illustrated in Figure 3.4. By specifying $position = CENTROID$, stresses can be retrieved from the center of the 1×1 cm^2 mesh or from the center of the center element of every square centimeter for a 2×2 mm^2 mesh.

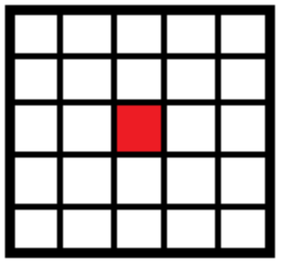


Figure 3.4: The red element in this 2 mm^2 mesh is the point where the stresses are acting on the flaws, and is thus from where the stresses are retrieved.

3.4.2 The main structure

The Python script imports the odbAccess package that enables reading from the relevant .odb-file of a simulation. The data from this Abaqus analysis will together with each individual flaw map represent the plates in the analysis.

The length and orientation of the flaws for the Python script is read from a text file. The flaw information is generated for plates cut from basic plates, and the text files in this thesis has been generated through the MATLAB script. The reader is referred to the Appendix A and B for details regarding the generating of flaws and the retrieval of these, respectively. Based on the amount of data in this text file, the program calculates the number of plates in the analysis.

Depending on the loading situation, the time step are looped through in either a ascending or descending order. What is meant by loading situation is further elaborated in Section 3.4.3. For a time step, stresses from elements corresponding to the first failing flaw candidates for all plates are read from the .odb-file. The stresses are saved in a list.

Further are every flaw in every plate looped through, and the list of stresses is used to check if failure occurs for any of the flaws. If failure occur for a flaw, it is saved in the new list of first failing flaw candidates for the next time step.

Eventually the first failing flaw(s) for a plate is found at some time step. Depending on the density of the time steps, there may be more than one flaw failing for a plate. Which is the first failing flaw is determined by which flaw lies closest to the non-failing time step stresses by percentage, assuming stresses increase linearly with load. To calculate this, the stresses are first needed converged by a version of the binary search algorithm, as also in Section 3.3.3, in the interval of non-failure stresses and failure stresses.

After assessing which flaw fails first for every plate, information about the failure is stored in a text file for a Monte Carlo analysis.

The pseudo formulation of the main structure is divided into two parts and presented in Algorithm 3 and 4 as Part 1 and Part 2, respectively. Part 1 finds the first failing flaw(s) of a plate for stress data extracted from the output database. Part 2 is the stage of converging the stresses for all failed flaws to find the one first failing flaw for every plate. The parts do not go into depth in the handling of the numerous specific cases, nor software validation procedures. The reader is referred to the Appendix B for the whole logical code.

Algorithm 3: Pseudo formulation for Part 1 of the main structure of the Python code.

```
open connection to odb-job;
define initial conditions;
retrieve flaw lengths and angles from text file;
while not all plates have first failing flaws do
    for all element/flaws candidates do
        find $\sigma_{xx}, \sigma_{yy}, \tau xy$ at this time frame and save in array structure;
    end
    for all plates searching for 1st failing flaw do
        for all element/flaw(s) candidates do
            retrieve length and angle for flaw(s);
            retrieve $\sigma_{xx}, \sigma_{yy}, \tau xy$ for this element from array structure;
            calculate the flaw(s) $\sigma$;
            if this is the first time frame tested then
                discard the flaw with lowest $\sigma$ as candidate;
            end
            calculate whether this flaw fails or not;
            if flaw fails then
                save information in array structure;
            else
                discard this flaw as candidate;
            end
        end
    end
end
```

Algorithm 4: Pseudo formulation for Part 2 of the main structure of the Python code.

```
for all plates do
    for all failed flaws do
        if failed flaw belongs to plate then
            retrieve stresses for the time frame with no fail for this flaw;
            while no convergence do
                apply critical search method 3.3.3 until convergence criteria is met;
            end
            save critical stress for this flaw in this plate in array structure;
        end
    end
    for all converged first failing flaws do
        calculate how close the critical stress is to the non-failing frame stress by
          percetage;
        if this this percentage is the lowest registered then
            register this flaw as best candidate thus far;
        end
        save information of the first failing flaw for this plate in text file
    end
end
```

3.4.3 Optimization

Since it can be time consuming reading field data from the output database, the structure of the Python code is more efficient the less field data is retrieved from Abaqus. Hence, it is advantageous to process all plates in the analysis for the same stresses, opposite to the MATLAB code.

Some loading conditions are optimized by the plates are tested for all time steps in descending order to eliminate non-candidates. Thus, it is advantageous to limit the maximum load to eliminate more non-candidates. This optimization is applicable for cases of geometrically linear behavior, or generally for cases where the stresses in the elements are constant or increasing. For oscillating stresses on the other hand, it is necessary to search for the first failing flaw in an ascending order of the time steps, and no flaws can be eliminated as first failing flaw candidates.

3.4.4 Linear solutions

Depending on geometry, boundaries and loads, a glass plate can behave geometrically linear until the point of failure. For these solutions it is not necessary to read the stresses from the output database at all time frames. By retrieving the last time frame with the maximum load generated in Abaqus, the search method in Section 3.3.3 will linearly and more effectively find the critical stresses, yielding the same result as with multiple time frames. Since the Python program handles the general case, both nonlinear and linear, reducing the number of time frames read needs to come from the .odb-file. In other words, field data output should be specified in Abaqus to be created only at step time for the linear solutions, in order to be processed most effectively.

3.4.5 Run time

An analysis of the asymptotic worst case run time was performed. The worst case performance of the program will occur if all flaws fails at all time steps. Thus, running the program with T timesteps, P plates, e elements, a tolerance TOL with the binary search algorithm and failure first found for load q_f, the worst case performance of the main structure will be

$$\mathcal{O}(Pe\big(T + \log_2\left(\frac{q_f}{TOL}\right)\big)) \tag{3.6}$$

3.5 FORTRAN implementation

The FORTRAN code was implemented with the purpose of visualizing fracture in glass. It is a VUSDLFD subroutine, which enables the simulation of fracture as the critical stresses as reached. For the source code the reader is referred to Appendix C.

Because of the desire of simulating the first failing flaw and the early consequential crack propagation, this event is more likely isolated by using small time steps. An explicit method is preferred to an implicit method when the time steps are small and many, avoiding having to solve a large set of equations at each iteration[32]. VUSDLFD is an explicit user subroutine allowing for changing user defined material properties based on field variables and was hence chosen for the problem of finding the failing flaws[31]. The user defined material properties are further elaborated in Section 3.5.1.

As with the Python code, all combinations of loads, boundary conditions and thin plate geometries are applicable with Abaqus simulations. The mesh size corresponds to the density of flaws, in other words, as there is one flaw per cm^2, the mesh size is one cm^2. What differentiates the Python and FORTRAN code is the Python code being probabilistic, running a Monte Carlo analysis on several flaw maps, while the FORTRAN code only uses a single random case with one flaw map. I.e. the FORTRAN code is not a software implementation of Yankelevskys' strength model, but uses the flaw map property of this model.

3.5.1 Subroutine structure

The input file of an Abaqus job includes a text file with the following history variables:

1. a: size of the flaw in this element
2. $theta$: the orientation of the flaw in this element
3. $nfail$: number of time steps this element has failed

4. K/K_c: defines failure for this elements if value reaches 1. In that event $nfail$ increases by 1.

5. $status$: when $nfail = 5$ status is set to failure and the element is eroded.

These variables are retrieved at each time step for every element in the subroutine, as well as the stresses at the integration points. The code is designed for elements of a single integration point, for instance S4R[33], such that there is only one value per stress variable on each side of the element.

K is calculated from a, $theta$ and the stresses. Further, K/K_c is updated for this element at this time step. If K/K_c is greater than 1, the flaw has failed, and the value of $nfail$ increases by 1. If K/K_c is less than 1, $nfail$ is initialized to 0. After the subroutine is run a number of times and the flaw has failed 5 times in a row, such that $nfail$ is 5, the flaw is considered failed.

The reason for requiring 5 failures in a row before acknowledging the flaw as failed, is the elastic waves travelling through structures in Abaqus/Explicit, potentially causing the stresses to momentarily leap [34]. Such a leap may in turn cause the subroutine to register a failed flaw. It is however assumed unlikely that this will happen five times in a row, and in that event the subroutine can be certain the flaw has failed. Finally, when $nfail$ has the value of at least 5 for the integration point at all section points element by $status$ to 0. When the status variable is 0 it means the element is inactive and is thus eroded[35]. Once an element is eroded the stresses in the specimen will relocate some and typically concentrate around the deleted flaw. Therefore surrounding elements are increasingly susceptible to failure once a neighbor element has failed, and may cause a chain of element failures. This chain is comparable with crack propagation and can be used to illustrate this phenomena.

A limitation to the model is posed by Abaqus, by the Status variable requiring failure for all section points for every integration points. This does not reflect the phenomena of glass fracturing exactly, as it is usually initiated from only one side of the glass. However, the FORTRAN implementation is only preliminary work exploring the possible usage of the model, and for visualizing crack propagation it is considered adequate for this thesis.

Chapter 4

Verification

This chapter presents the verifications of the software implementations in MATLAB, Python and FORTRAN. In developing the model, two possible sources of error exist: Faults in the mathematical foundation and mistakes in the software implementation of the mathematical foundation. In the previous chapter, the mathematical background developed by Yankelevsky was presented, and will in this chapter be assumed to be correct. In this section, emphasis is put on the software implementations.

The MATLAB code is verified by comparing stresses and deflections with corresponding Abaqus simulations. The next step is verifying the Python and FORTRAN programs by comparing results with the MATLAB code. Throughout the sections describing these verifications, a $1 \times 1 \times 0.006$ m^3 geometry is used in the analyses. The uniform loads in the simply supported and clamped plates are simulated in Abaqus by applying a pressure on the face of the plate. The stretched plate are simulated by applying "Shell Edge Loads" on one horizontal and one vertical edge, while the other edges are simply supported.

4.1 Preliminary Controls

In order to avoid software bugs, continuous quality controls of the implementations were performed. Some of these controls were:

- Mathematical errors in the implementation of the stresses in the MATLAB code was avoided by the use of Symbolic Math Toolbox in the Command Line.
- Comparison of system out prints of the implementations.
- Comparison of graphical plots displaying preliminary results of the implementations.
- Interpretation of results in comparison with Abaqus/Implicit.

4.2 Verification of the MATLAB code

For the MATLAB code to find the first failing flaw and calculate the strength of a glass plate, it is essential that the stresses are calculated correctly. To verify the behaviour of the MATLAB plate, comparisons are made with corresponding cases in Abaqus. As the MATLAB code behaves geometrically linear, nonlinear behavior is excluded from the Abaqus analyses. The three cases of the analytical solution: the uniformly loaded simply supported plate, the uniformly loaded clamped plate and the stretched plate, are considered sufficient in verifying the code. These are compared with corresponding simulations in Abaqus on the center deflection w_c for the simply supported and clamped plates, maximum displacement δ_{max} for the stretched plate, and the center stresses $\sigma_{xx,c}$, $\sigma_{yy,c}$ and $\tau_{xy,c}$. In addition, a small study regarding the behavior of plates under biaxial and uniaxial tension was performed.

The MATLAB code was run for a single simply supported plate of dimension $1 \times 1 \times 0.006$ m^3 with 49 series in Naviers plate solution. For the same dimensions and the resulting critical load q_c, an Abaqus/Implicit simulation was run. The results are presented in Table 4.1.

Table 4.1: Data from MATLAB for the simply supported in Section 4.2

	q_c [kPa]	w_c [mm]	$\sigma_{xx,c}$ [MPa]	$\sigma_{yy,c}$ [MPa]	$\tau_{xy,c}$ [MPa]
MATLAB	4.4	-13.46	32.77	32.77	0.005
Abaqus/Implicit	4.4	-13.56	33.06	33.06	0.0049

Further, the MATLAB code was run for a single clamped plate with 49 series in the double cosine series. For the critical load q, a corresponding Abaqus/Implicit simulation was run. The results are presented in Table 4.2.

Table 4.2: Data from MATLAB for the clamped plate in Section 4.2

	q_c [kPa]	w_c [mm]	$\sigma_{xx,c}$ [MPa]	$\sigma_{yy,c}$ [MPa]	$\tau_{xy,c}$ [MPa]
MATLAB	6.2	-5.92	22.67	22.67	0.0054
Abaqus/Implicit	6.2	-5.92	22.20	22.20	0.0053

At last, the MATLAB code was run for a plate biaxially stretched with equal force on one horizontal and one vertical edge. For the critical load q_c, a corresponding Abaqus/Implicit simulation was run. The results are presented in Table 4.3.

Table 4.3: Data from MATLAB for the stretched plate in Section 4.2

	q_c [MPa]	δ_{max} [mm]	$\sigma_{xx,c}$ [MPa]	$\sigma_{yy,c}$ [MPa]	$\tau_{xy,centre}$ [MPa]
MATLAB	25.39	0.36	25.39	25.39	0
Abaqus/Implicit	25.39	0.28	25.39	25.39	-1.52E-16

The results coincide and the small deviations seen is likely to reduce with more series for the plate solution for the simply supported and clamped plate. For the stretched plate the stresses are almost identical, although there are some deviation in the displacement δ_{max}. However, this parameter is merely showing the behavior of the plate and will not affect the failing flaw calculation, which is dependent on the stresses.

An additional study was performed, verifying the behavior of the MATLAB implementation of the model with regards to glass behavior under biaxial and uniaxial tension. The MATLAB code applies loads to two edges of the plate, a horizontal and a vertical edge. A load q is applied to one of these edges, while the load applied to the other corresponds to the product of q and a factor. Thus for a factor of 1 the edges are equally loaded, while for a factor of 0 there is uniaxial tension in the plate. Table 4.4 presents the critical loads q_c from running the MATLAB code for a $1 \times 1 \times 0.006$ m^3 stretched plate with these factors. 992 plates were used in the analyses. The results indicate that q must be higher in order for a uniaxially loaded plate to fracture compared to a biaxially loaded plate. This is a natural response due to the increased probability of finding a critically large flaw oriented perpendicular to the maximum resultant stress in the plate. This indicates that the MATLAB program has successfully implemented typical glass behavior.

Table 4.4: Critical loads q_c for a $1 \times 1 \times 0.006$ m^3 plate exposed to biaxial and uniaxial stretching.

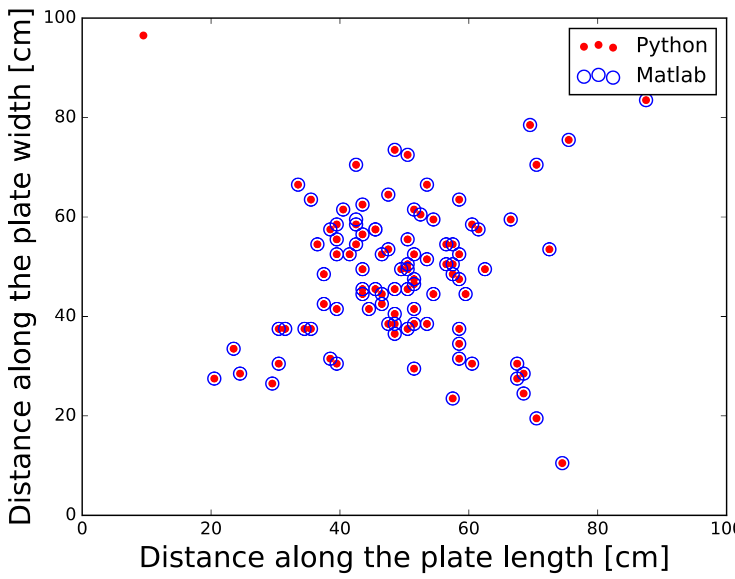
Factor	q_c [MPa]
1	69.66
0	77.32

4.3 Verification of the Python code

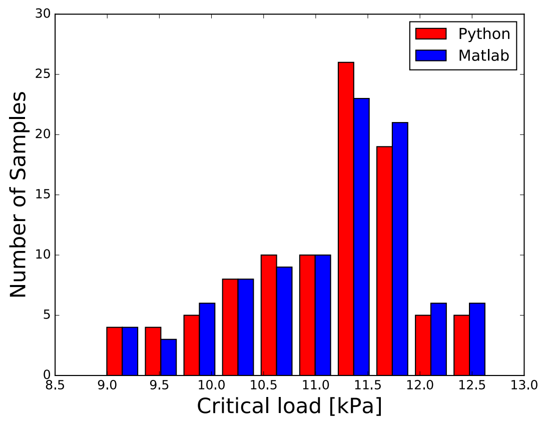
The Python code is capable of handling more complicated solutions than the MATLAB code. Verification of the Python code is performed by comparing the three analytical solutions from MATLAB to corresponding solutions produced by Abaqus and Python. It is assumed that the Python program will handle any loading situation correctly if it produces coinciding results to the three MATLAB cases. Histograms of both critical loads and deflections, and also the map of the first failing flaws will be compared. The mean critical loads $q_{c,avg}$ and mean center deflections $\delta_{c,avg}$ are given in Table 4.5.

To produce the same results in the MATLAB and Python programs, it is essential the same flaw maps are used. Therefore the flaw map information is gathered in a textfile, produced by the MATLAB code while running its analysis, and later retrieved for the analysis in Python. It is also important to turn off nonlinear effects for the simulation in Abaqus as the analytical solutions does not account for this.

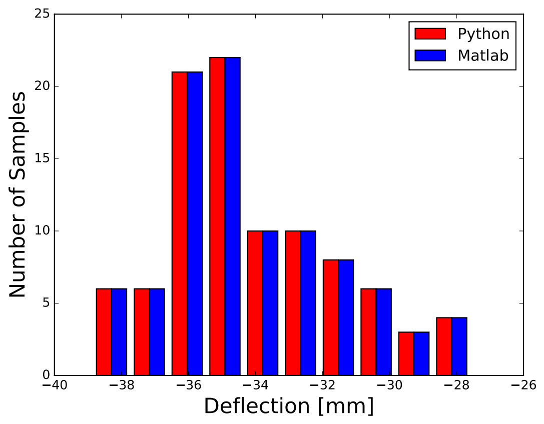
In this sections Monte Carlo analyses, 96 plates have been examined for every case. For the simply supported and clamped, 30×30 series are used in the analytical solutions. The MATLAB script will find the critical point of failure by converging with respect to critical load, as is described in Section 3.3.2. The Python code will however converge with respect to stresses, as Section 3.4.2 describes. Since the loads and stresses are proportional, a convergence criterion of 0.0002 has been applied for the critical loads and the stresses for the MATLAB code and Python code, respectively, in order to receive compliant results.



(a) First failing flaws.



(b) Critical loads with bin size 0.36 kPa.

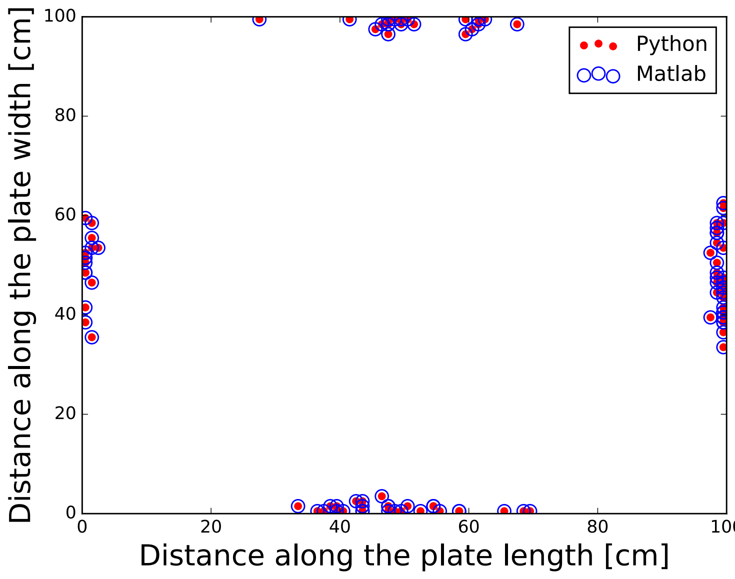


(c) Deflections with bin size 1.11 mm.

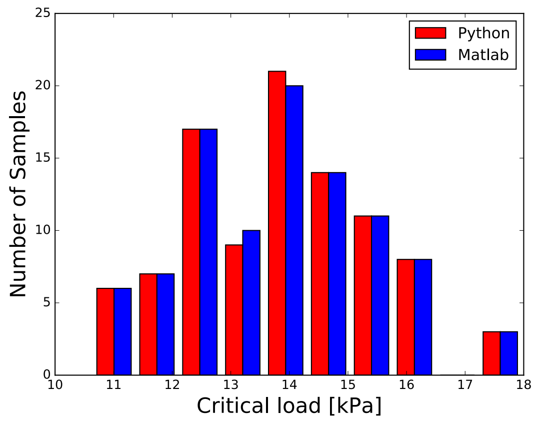
Figure 4.1: Diagrams for an analysis of 96 simply supported plates.

Figure 4.1(a) shows the distribution of the first failing flaws of the 96 plates in the simply supported plate analysis. They gather around the center of the plate, the area of the highest stress concentrations. The results from the analytical solution and the Python programme coincide, with one single deviation.

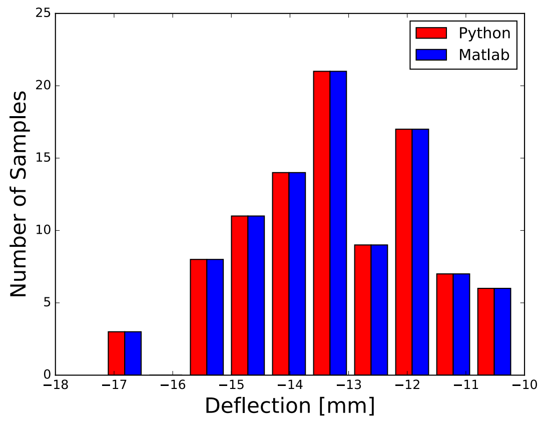
Figure 4.1(b) and Figure 4.1(c) show the density of the critical loads and center deflections for the failing simply supported plates. Visual inspection suggest distributions show similar curves and magnitudes. Mean values from both figures are given in Table 4.5 also suggesting coinciding results, only with a slight deviation in the mean deflection.



(a) First failing flaws.



(b) Critical loads with bin size 0.76 kPa.

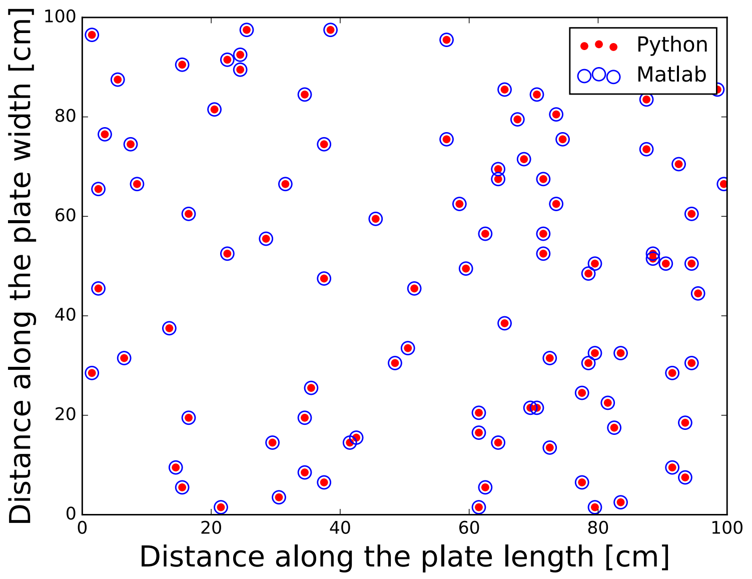


(c) Deflections with bin size 0.69 mm.

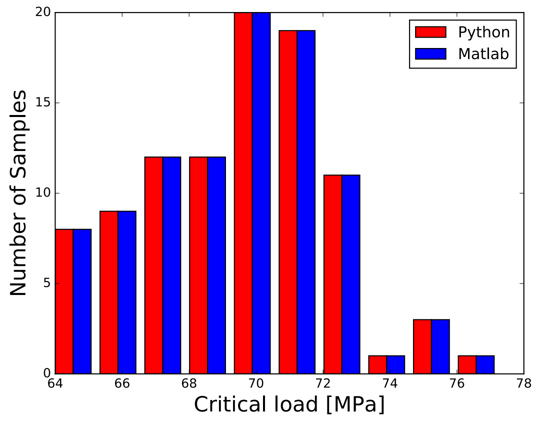
Figure 4.2: Diagram for an analysis of 96 clamped plates.

Figure 4.2(a) shows the distribution of the first failing flaws of the 96 plates in the clamped plate analysis. The flaws concentrate along the middle of the edges, which is the area of the highest stress concentrations for the clamped plate. The overall result yields satisfactory consistency.

Figure 4.2(b) and Figure 4.2(c) show the density of the critical loads and deflections for the failing clamped plates, while the mean values from both figures are given in Table 4.5. A slight deviation in the critical loads is seen by visual inspection, although the mean critical loads given in Table 4.5 are equal. There is a slight deviation in the mean deflection values, however the deflection distributions indicate completely coinciding results. The results show good correspondence between the analytical solution and the Python code.



(a) First failing flaws.



(b) Critical loads with bin size 1.5 MPa.

Figure 4.3: Diagram for an analysis of 96 stretched plates.

Figure 4.3(a) and 4.3(b) show the relevant values for verification for the stretched plate. The plate is stretched with uniform load over one horizontal and one vertical edge, while the remaining edges are simply supported. The stresses are thus uniform throughout the plate and probability for failure is equal at any point, as is reflected in Figure 4.3(a).

The location of the failing flaws of the Python fully match the results from the analytical solution. Also the critical loads coincide with the MATLAB code. Including deflection was considered unnecessary in verification of the stretched plate as the analytical solution has no distribution for this parameter. The mean center deflection, as well as critical load, given in Table 4.5, presents the coinciding results with the analytical solution for the stretched plate.

Table 4.5: Mean critical load and deflection for Section 4.3

	Simply supported plate		Clamped plate		Stretched plate	
	$q_{c,avg}$ [kPa]	$\delta_{c,avg}$ [mm]	$q_{c,avg}$ [kPa]	$\delta_{c,avg}$ [mm]	$q_{c,avg}$ [MPa]	$\delta_{c,avg}$ [mm]
MATLAB	11.1	34.11	13.8	13.19	69.42	0
Python	11.1	34.13	13.8	13.21	69.42	4.19E-16

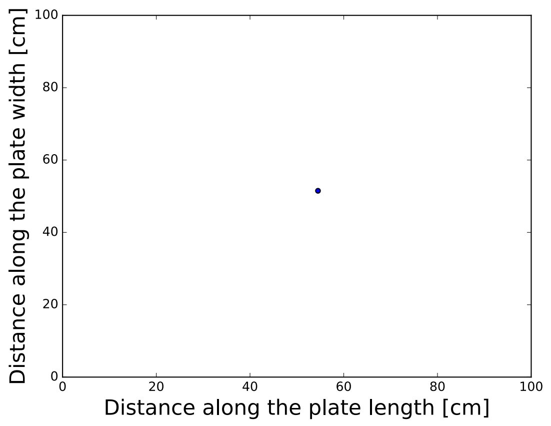
The small deviations seen is likely due to different approaches of calculating stresses. While the analytical solutions three cases uses Hooks law[36] for the stretched plate and the thin plate theory given in 2.3 for the simply supported and clamped plates, Abaqus uses finite element theory.

4.4 Verification of the FORTRAN code

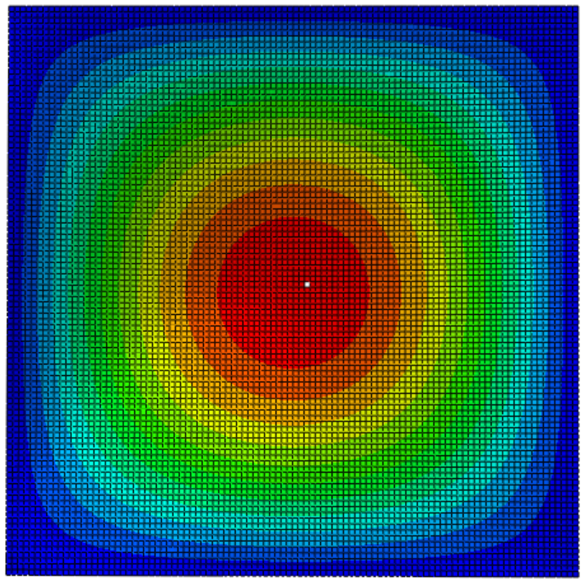
The FORTRAN code is a VUSDLFD subroutine called when running a job in Abaqus, with purpose of visualizing the failed flaws by element erosion. The verification process in this section will be conducted by comparing the location of the first failing flaws, center stresses and deflections of the analytical solution and the FORTRAN solution for the simply supported, clamped and stretched plate. The same flaw maps are used in the analyses. The first failing flaws are visualized by element erosion in Abaqus, acquired by changing the Status Variable to the K/K_{crit} variable and specifying element deletion when its value is equal to or greater than 1.

All simulations are run quasi-statically in Abaqus/Explicit with a total time of 0.1 seconds. Attempting to avoid elastic vibrations, the load is ramped up with a "Smooth step"-amplitude. Nonlinear effects is not included in Abaqus for compliance with the analytical solution.

Figure 4.4 shows that the same flaw is critical both for the analytical solution and the FORTRAN code for a simply supported plate. The flaw is located close to the center of the plate, the point of maximum deflection and stresses. These values are given in Table 4.6 and shows the FORTRAN codes consistency with the analytical solution for the simply supported plate.



(a) Analytical solution.



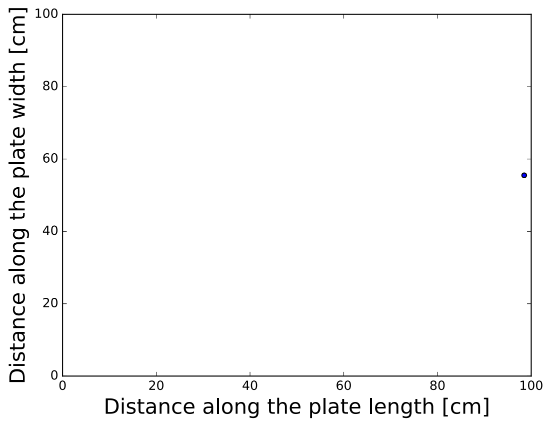
(b) Abaqus subroutine. The colors indicate deflection magnitude where red signifies the greatest magnitude.

Figure 4.4: First failing flaws for the simply supporetd plate in Section 4.4.

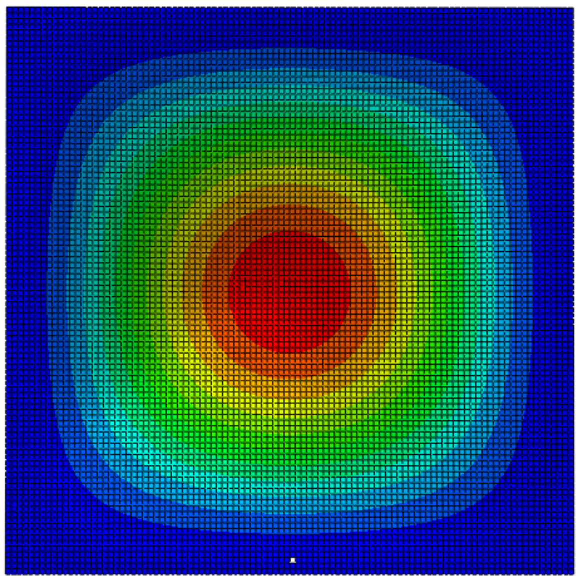
Table 4.6: Data from MATLAB for the simply supported plate in Section 4.4

	w_{centre} [mm]	$\sigma_{xx,centre}$ [MPa]	$\sigma_{yy,centre}$ [MPa]	$\tau_{xy,centre}$ [MPa]
FORTRAN	-26.37	63.96	63.96	0.0126
MATLAB	-26.12	63.77	63.77	0.0096

The results corresponding to the clamped plate is given in Figure 4.5 and Table 4.7. By visually inspecting Figure 4.5 it becomes clear that the clamped plate in this analysis fails for different flaws. The analytical solutions failure starts with a flaw along the right vertical end of the plate, while the subroutine fails for a flaw along the bottom horizontal edge. Further, the failure stresses and deflections in the center of the two solutions differ, as depicted in Table 4.7, although not drastically. Several factors may be the cause of this. Although the effect of elastic vibrations are attempted suppressed, as discussed in Section 3.5, they may still be present and affecting the analysis. A finer mesh along the boundaries of the plate may result in more accurate results, however this is not supported by the current version of the FORTRAN code.



(a) Analytical solution.



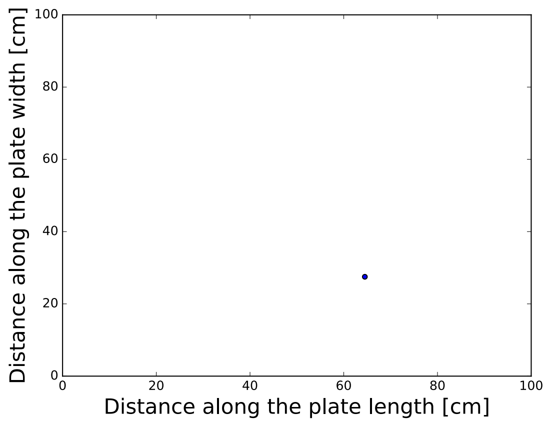
(b) Abaqus subroutine. The colors indicate deflection magnitude where red signifies the greatest magnitude.

Figure 4.5: First failing flaws for the clamped plate in Section 4.4.

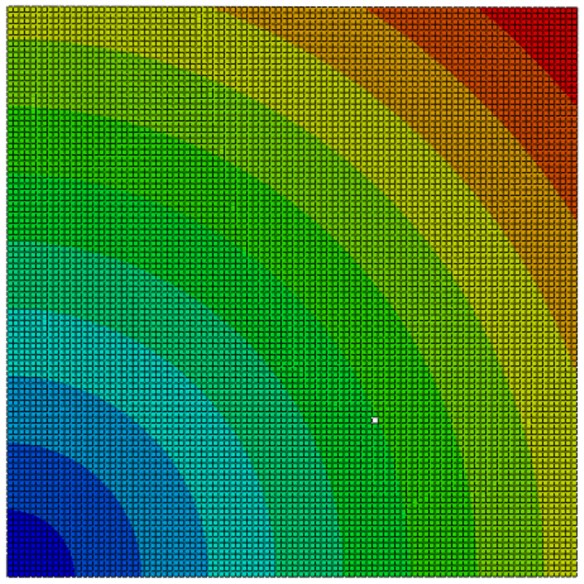
Table 4.7: Data from the FORTRAN subroutine for the clamped plate in Section 4.4

	w_{centre} [mm]	$\sigma_{xx,centre}$ [MPa]	$\sigma_{yy,centre}$ [MPa]	$\tau_{xy,centre}$ [MPa]
FORTRAN	-10.70	38.95	38.95	0.043
MATLAB	-9.30	34.85	34.85	0.008

Figure 4.6 and Table 4.8 presents the results for the stretched plate verification analysis. The stretched plate is in Abaqus constructed with the left and bottom edges simply supported, while right and top edges are applied shell edge loads. Figure 4.6 shows coinciding results for the stretched plate in terms of the location of the flaw. Also the critical stresses of the analysis are consistent with those of the analytical solution, as presented in Table 4.8.



(a) Analytical solution.



(b) Abaqus subroutine. The colors indicate deflection magnitude where red signifies the greatest magnitude.

Figure 4.6: First failing flaws for the stretched plate in Section 4.4.

Table 4.8: Data from FORTRAN subroutine for the stretched plate in Section 4.4

	w_{centre} [mm]	$\sigma_{xx,centre}$ [MPa]	$\sigma_{yy,centre}$ [MPa]	$\tau_{xy,centre}$ [MPa]
FORTRAN	3.2E-15	52.97	52.97	0.0005
MATLAB	0	52.72	52.72	0

Good correspondence was found with the simply supported and stretched plate, while deviation occurred with the clamped plate. The clamped plate failed with center stresses deviating by approximately 10.52% suggesting the plate fails at a reasonable stress state, although not in exact correspondence with the results of the analytical solution. Deviations also occurred for some of the clamped plates in the 96-plates analyses of the Python code verification, however with significantly smaller deviation of the center stresses. The overall result suggests the code can be used for visualizing failing flaws, although deviations of some degree may occur.

Chapter 5

Parametric Studies

This chapter presents the parametric studies performed in this thesis. The purpose of these studies is documenting the behavior of the strength model with respect to geometric parameters and in that regard discuss its validity. Results include critical loads presented in histograms and cumulative probability diagrams. These distributions have been compared with a fitted Weibull distribution, the traditional distribution for glass strength, as described in 2.2.2. They have also been compared with a fitted Normal distribution, which has been discussed whether is a more accurately descriptive distribution for the problem[6]. The parameters for these fitted distributions, the mean value μ, standard deviation σ, Weibull modulus m and the Weibull scale parameter σ_0, are given in a separate table for every study.

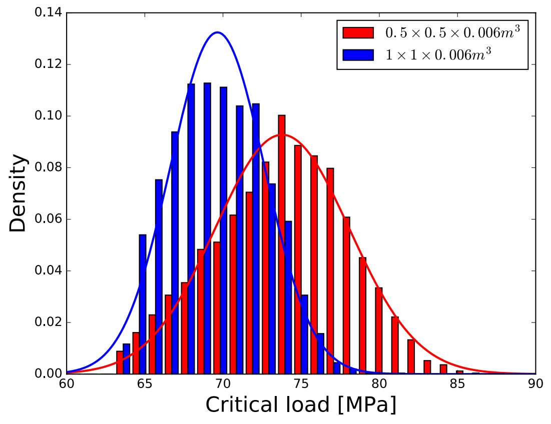
All analyses in this chapter uses quadratic glass plates which are exposed to tensile stresses from "Shell Edge Loads" at one vertical and one horizontal edge in Abaqus/Implicit, i.e. in-plane stretching. The remaining edges are simply supported.The analyses in this chapter uses 2400 plates as this is considered sufficient for documenting the variation in behaviors of the plate for the different studies. The studies are performed with the Python program.

5.1 Plate Geometry

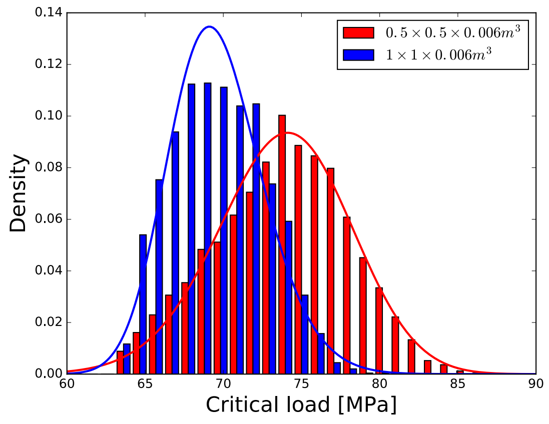
Two plates of sizes $1 \times 1 \times 0.006$ m^3 and $0.5 \times 0.5 \times 0.006$ m^3 are subjected to in-plane stretching. Figure 5.1 shows the critical load diagrams for the large and the small plate and Table 5.1 presents the parameters for the Normal and Weibull distributions fitted to the critical load data of the examined plates in this section. The large plate experiences failure for smaller loads than the small plate. This agrees with the theory depicted in Section 2.1.3, saying that with more flaws present on the specimen it is more likely one is a critical flaw. Since the density of flaws is the same for the two geometries examined in this section, there are more flaws on the bigger plate, and in average a higher prevalence of large flaws. In other words, a large plate will have more weak points.

There is potential for the effect of large plates having lower strength due to more weak points to be even greater than what the current version of the model predicts. With a density of one flaw per cm^2, the likelihood of the flaws' stress concentrations affecting another significantly is minimal and therefore this is not part of the implementations. However, the structured density of flaws in this model is merely a simplification and does not reflect a realistic flaw map. A realistic flaw map is unstructured and instances of the amplifying and shielding effect on the stresses, as described in Section 2.1.3, are likely to occur. The phenomenon of two flaws creating the amplifying effect will by average happen every some number of flaws, thus the more flaws in a plate the more probable these incidents occur.

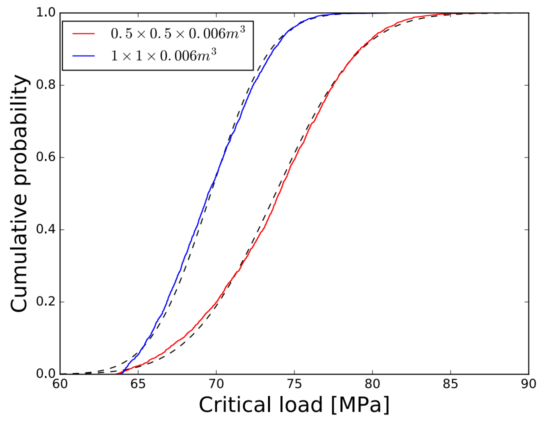
Figures 5.1(a) and 5.1(c) show the Normal distribution fitted to the resulting probability and cumulative probability distributions of the current study, while Figures 5.1(b) and 5.1(d) are fitted with the Weibull distribution. The dashed lines in the cumulative distribution diagrams represent the fitted curves. Both curves seem to yield similar results, which is in accordance with the findings of [5]. However, the Weibull distribution provides a better description of the steep rise in the bottom of the distribution for the bigger plate. Visually comparing the Normal and Weibull cumulative distribution curves, the overall best fit seems to be the Weibull distribution for the results in the current study.



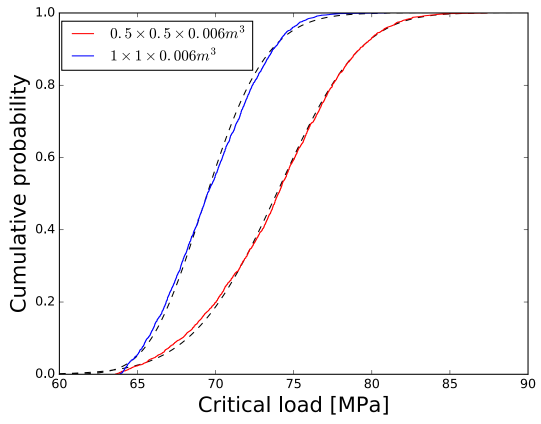
(a) Normal probability density distribution curves.



(b) Weibull probability density distribution curves.



(c) Cumulative normal distribution curves.



(d) Cumulative Weibull distribution curves.

Figure 5.1: Distributions of critical loads for plates of $1 \times 1 \times 0.006\text{m}^3$ and $0.5 \times 0.5 \times 0.006\ \text{m}^3$, fitted with Normal and Weibull distribution curves. The bin size is 1.1 MPa for the histograms.

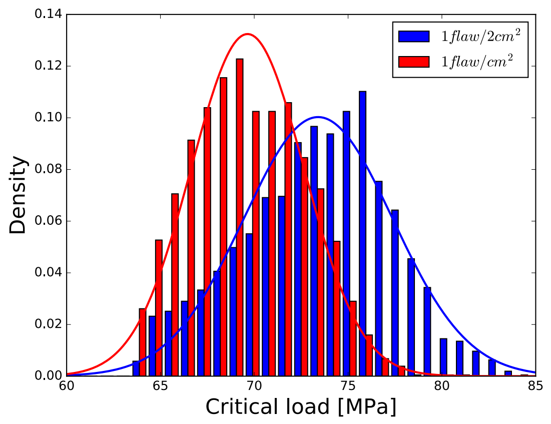
Table 5.1 presents the parameters of the Normal and Weibull distributions fitted to the critical load data in MPa. As given in Section 2.2.2 the Weibull shape parameter m is assumed a material constant. The results in Table 5.1 contradicts this, as the largest plate with the lower strength has a higher m compared with the smallest plate, despite having the same material properties. m is a measure of the scatter of the distribution, where a narrow distribution corresponds to a high value of m[4]. By inspecting Figure 5.1 it becomes clear the small specimen has a larger scatter in strength. This scatter is a consequence of the lower prevalence of large flaws in the plates, meaning the mean critical flaw length is smaller such that higher stresses is in average required for the flaws to fail. It can also be seen that the distribution ranges from approximately the same interval start, and this is because the small plates that does contain large flaws will fail for the same loads as the large plate. This indicates that less flaws in a specimen will lead to a lower m.

Table 5.1: Parameters for the Normal and Weibull distributions for the different plate geometries in Section 5.1.

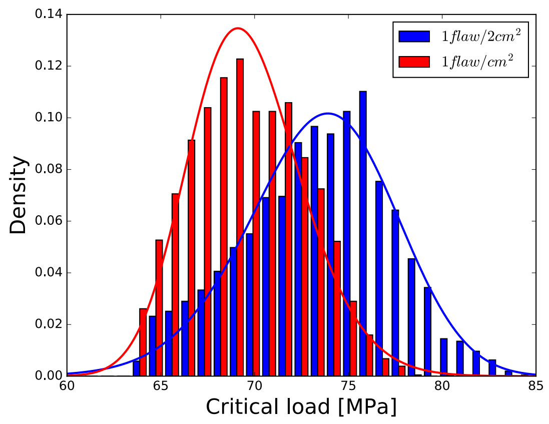
	Normal		Weibull	
Geometry	μ	σ	m	σ_0
$1 \times 1 \times 0.006\text{m}^3$	69.64Mpa	3.01MPa	23.61	71.10MPa
$0.5 \times 0.5 \times 0.006\ \text{m}^3$	73.78MPa	4.30MPa	18.64	76.23MPa

5.2 Flaw Density

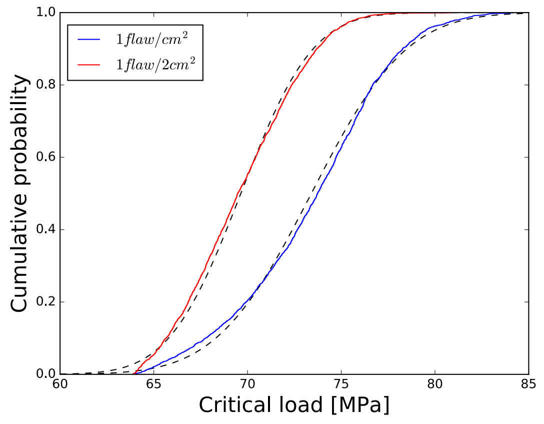
The implementations have used a structured flaw density of 1 flaw per cm^2. However, a preliminary study on flaw density performed in [26] suggests a density of one flaw per 1.18-2.60 cm^2, thus a reduced number of flaws per plate in comparison with the current model. A parametric study was performed on a $1 \times 1 \times 0.006$ m^3 plate, with a density of one flaw per 2 cm^2, which is in the range of the results of the study mentioned above. Figure 5.2 shows the critical load distributions of the implemented model and the study in question. Table 5.2 shows mean critical load for the denser flaw map is 69.6441 MPa and for the less dense concentration of flaws 73.4041 MPa. According to the results, with a lower the concentration, a glass plate will in average handle more loads and tensile stresses. This can be explained with the same logic as expressed in Section 5.1, that the prevalence of critical flaws is by average lower with less flaws. The plate in the current study has half the amount of flaws than the implemented model, thus the average critical load increases.



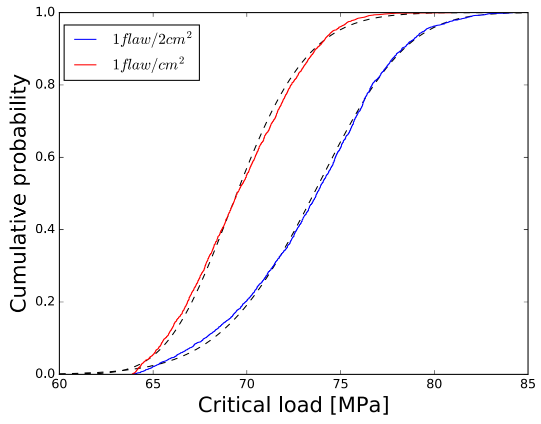
(a) Normal probability density distribution curves.



(b) Weibull probability density distribution curves.



(c) Cumulative normal distribution curves.



(d) Cumulative Weibull distribution curves.

Figure 5.2: Distributions of critical loads for $1 \times 1 \times 0.006\text{m}^3$ plates of 1 flaw per cm^2 and 2 cm^2, fitted with Normal and Weibull distribution curves. The bin size is 0.83 for the histograms.

Table 5.2: Parameters for the Normal and Weibull distributions for the different flaw densities ρ in Section 5.2

	Normal		Weibull	
ρ	μ	σ	m	σ_0
1 flaw/2cm^2	73.40MPa	3.97MPa	20.32	75.25MPa
1 flaw/cm^2	69.64MPa	3.01MPa	23.61	71.10MPa

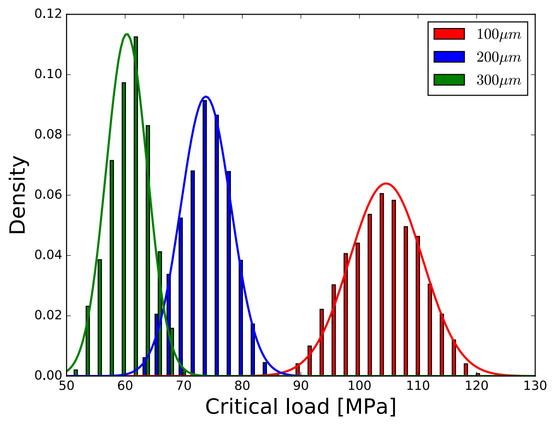
The dashed lines in Figure 5.2 represent the fitted cumulative Normal and Weibull distributions. The fitted distributions manage to fit the data from the analyses well. The sharp rise from the foot of the distribution for the plate with a denser flaw map is best captured by the Weibull distribution. By visually inspecting the cumulative distributions the fitted Weibull distribution overlaps the actual distribution to a greater extent than the normal distribution. This seems to especially be the case for the more dispersed curve, i.e. the plate with one flaw per 2cm^2.

The Weibull modulus m for the two plates analyzed in this section is given in Table 5.2. This study contradicts the assumption of the m being a material constant as the values are ranging from 20.3287 for the less dense flaw map to 23.6135 for the more dense, despite having the same material properties. As in Section 5.1, the distributions starts at approximately the same critical load value, but the plate holding less flaws has a larger scatter due to fewer instances of plates with large flaws. Hence, with less flaws in a plate m will decrease.

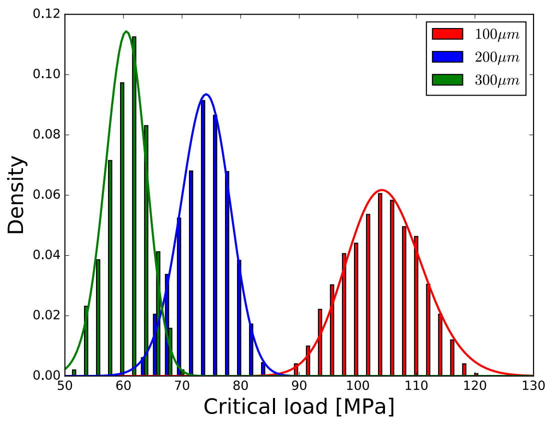
5.3 Flaw Geometry

Yankelevskys' model assumes half-penny shaped flaws, a maximum length of 200μm and a geometry correction factor Y of 1.12. For a half-penny shaped flaw the geometry correction factor Y is 0.6625, by Equation (2.10). The strength model in this thesis has applied this geometry correction factor. However, as Yankelevsky argues in [6], it is uncertain if the penny shape is a realistic shape for a flaw. Three parametric studies will be performed on different flaw geometry properties; length, geometric correction factor and finally depth. All analyses in this section uses a plate geometry of $0.5 \times 0.5 \times 0.006$ m^3. The dashed lines in the cumulative probability distribution diagrams represent fitted Weibull and Normal distributions.

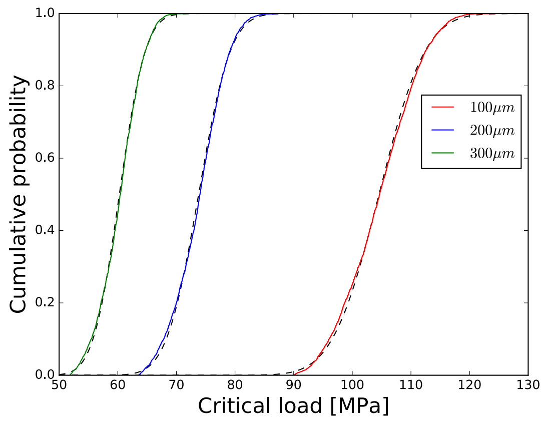
5.3.1 Flaw length



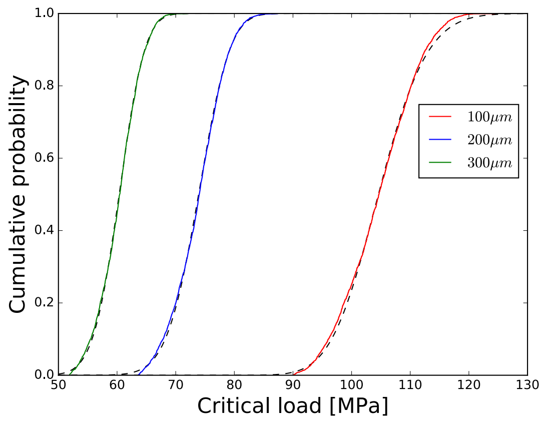
(a) Normal probability density distribution curves.



(b) Weibull probability density distribution curves.



(c) Cumulative normal distribution curves.



(d) Cumulative Weibull distribution curves.

Figure 5.3: Distributions of critical loads for $0.5\times0.5\times0.006\text{m}^3$ plates with maximum flaw lengths 100, 200 and 300 μm, fitted with Normal and Weibull distribution curves. The bin size is 2 MPa for the histograms.

Table 5.3: Parameters for the Normal and Weibull distributions for the different flaw lengths $2c$ in Section 5.3.1

	Normal		Weibull	
$2c$	μ	σ	m	σ_0
100μm	104.55MPa	6.24MPa	18.04	108MPa
200μm	73.78MPa	4.30MPa	18.64	76.23MPa
300μm	60.27MPa	3.51MPa	18.54	62.27MPa

Maximum flaw lengths of 300μ m and 100μm was applied to the model. Figure 5.3 compares the critical loads of the model of maximum flaw length 200μm to the 300μm and 100μm models. The mean critical loads of the plates are given in Table 5.3, showing that with a smaller maximum flaw length the glass plates in the model will tolerate more load. This is in accordance with observations from experiments. From Figure 5.3 it can also be observed that the variance of critical loads increases with an increased mean critical load. This is explained mathematically. for a small flaw size the stresses perpendicular to the flaw needs to be correspondingly high for failure to occur. With a percentage change in a small flaw length, the percentage change of the critical stress for this flaw will be greater than for the same percentage change in a large flaw length. This is why the variance is greater for smaller flaw lengths.

The non-equal spacing between the mean critical loads despite the equal spacing between the flaw lengths is explained by Equation (2.12). Here the toughness K is linearly proportional with the stress and therefore the load, and it is proportional to the square root of the depth a, which is the half flaw length. This relation is not linear.

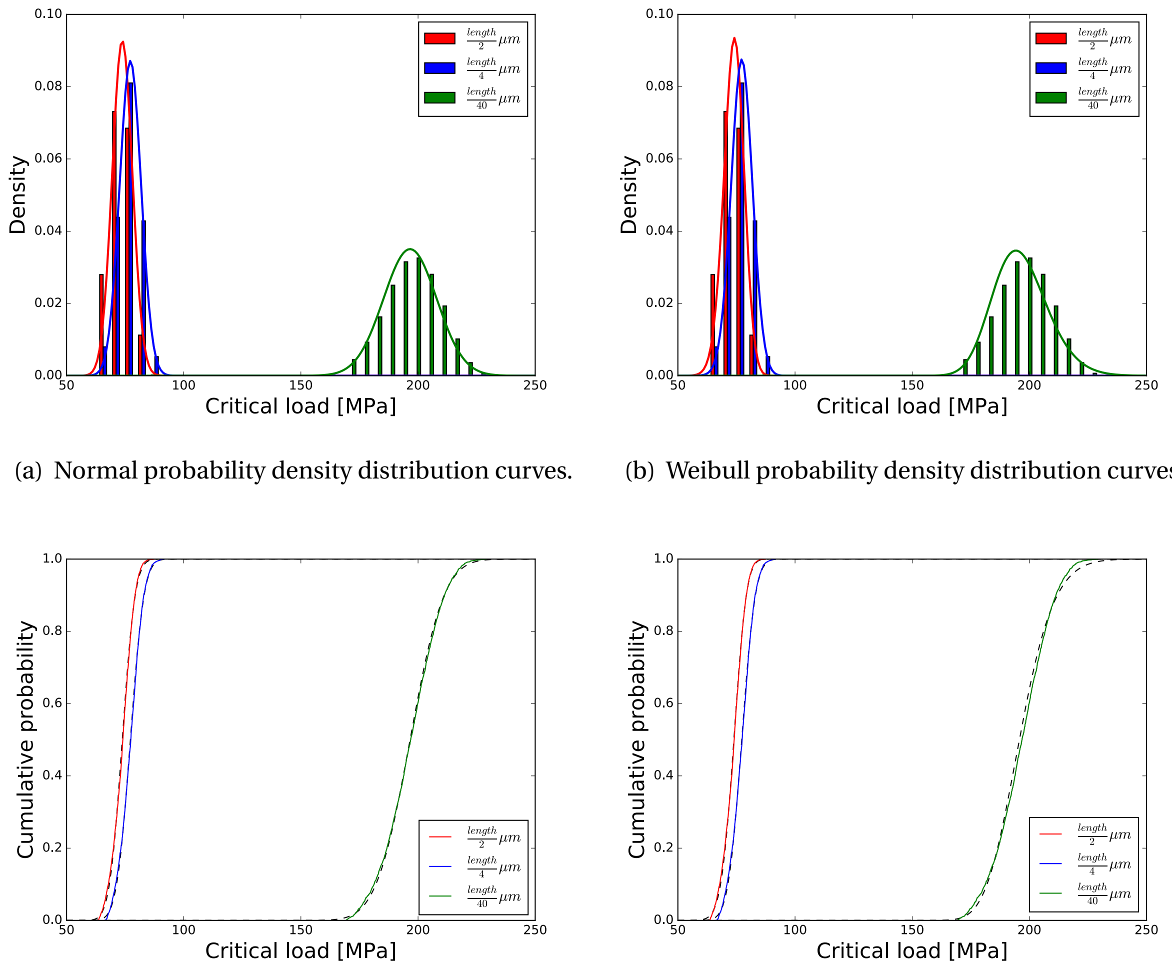
The fitted distributions in Figures 5.3(a) and 5.3(b), and the cumulative distributions of Figures 5.3(c) and 5.3(c) show similar results for the Normal and Weibull distribution. For the flaws lengths 200μm and 300μm, it is uncertain which distribution provides the most accurate fitting, but for the flaw of 100μm the Normal distribution is more precise. This is due to the Weibull distribution failing to capture the behavior at the bottom end of this distribution, which is noticed as a typical behavior for the Weibull distribution in [37].

5.3.2 Flaw depth

The depths a included in this study are are $a = c/2$ and $a = c/20$, where c is the half-length of the flaw. These depths have by Equation (2.10) the corresponding geometry correction factors $Y = 0.89$ and $Y = 1.12$, respectively. The results for the models with these flaw depths, along with the half-penny shaped flaws, are presented in Figure 5.4.

Figure 5.4 and the mean critical loads in Table 5.4 shows that the model finds higher strength for plates with more shallow flaws, which is the natural response. With a higher mean critical load it can be observed that the variance increases. This is explained by the same logic as with the same phenomena occurring in Section 5.3.1. Some percentage change in the depth of a shallow flaw results in a larger leap of the critical stresses than for the same percentage change in the depth of a deep flaw.

Normal and Weibull distributions are fitted to the data in Figure 5.4, yielding similar results and good approximations. For the two most shallow flaws, it is hard to conclude the most descriptive distribution by visual inspection, but for the most shallow flaw the Normal distribution provides the best fit. This becomes clear by inspecting the cumulative probability distributions.



(a) Normal probability density distribution curves.

(b) Weibull probability density distribution curves.

(c) Cumulative Normal distribution curves.

(d) Cumulative Weibull distribution curves.

Figure 5.4: Distributions of critical loads for $0.5x0.5x0.006m^3$ plates of depths 1/2th, 1/4th and 1/40th of the flaw length, fitted with Normal and Weibull distribution curves. The bin size is 5 MPa for the histograms.

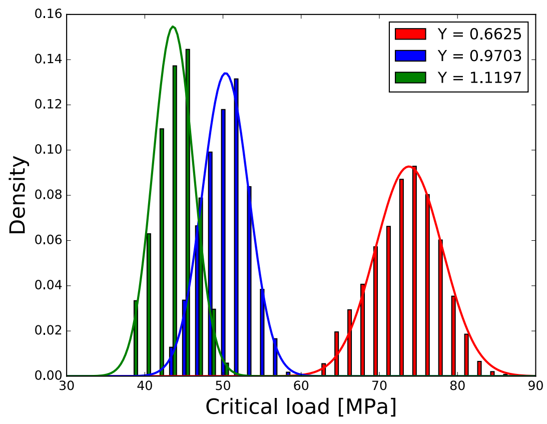
Table 5.4: Parameters for the Normal and Weibull distributions for the different flaw depths a in Section 5.3.2

	Normal		Weibull	
a	μ	σ	m	σ_0
$\frac{length}{2}$	73.78MPa	4.30MPa	18.64	76.23MPa
$\frac{length}{4}$	77.22MPa	4.57MPa	17.89	79.56MPa
$\frac{length}{40}$	196.59MPa	11.38MPa	18.71	201.941MPa

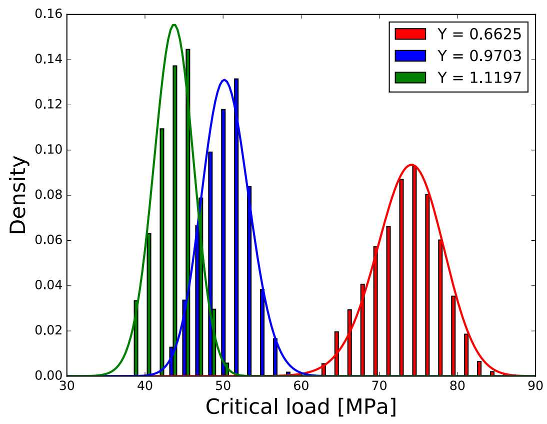
5.3.3 Geometry correction factor

The geometry correction factors of 0.9703 and 1.1197 were applied to the model without altering the depth a. Figure 5.5 shows the resulting critical loads, and the mean values are given in Table 5.5. The results indicate higher values of Y induce failure for smaller loads with a linear relationship. This it due to both Y and σ being proportional to K_c, and the loads proportional relationship with σ in this case. For geometric nonlinearity-inducing loading conditions, this proportionality will be between the geometry correction factor and stresses, but not the load. Further, the probability distributions show a higher variance for the lowest Y. This is because the product of the lowest Y and the percentage change of some flaw size results in a larger percentage change in the critical stress for that flaw, than for a higher Y and the same percentage change of the same flaw size.

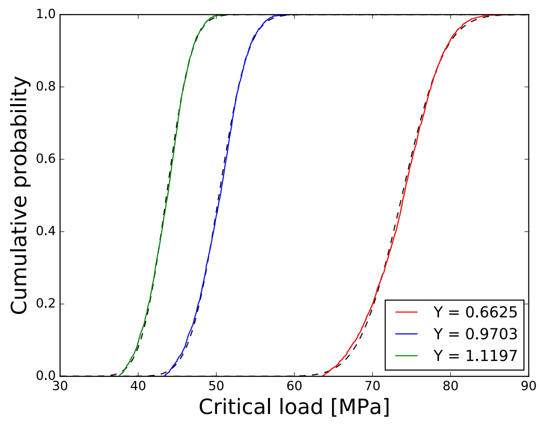
By studying Figure 5.5 it becomes clear the fitted Normal and Weibull distribution yield both good and similar results. Visually inspecting the graphs to find the better fit of the Normal and Weibull distributions yields inconclusiveness.



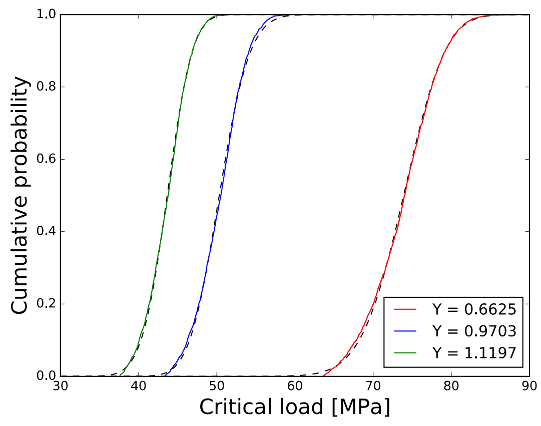
(a) Normal probability density distribution curves



(b) Weibull probability density distribution curves.



(c) Cumulative normal distribution curves.



(d) Cumulative Weibull distribution curves.

Figure 5.5: Distributions of critical loads for $0.5 \times 0.5 \times 0.006\text{m}^3$ plates with Geomtry correction factor 0.6625, 0.9703 and 1.1197, fitted with Normal and Weibull distribution curves. The bin size is 1.75 MPa for the histograms.

Table 5.5: Parameters for the Normal and Weibull distributions for the different geometry correction factors Y in Section 5.3.3

	Normal		Weibull	
Y	μ	σ	m	σ_0
0.6625	73.78	4.30	18.64	76.23
0.97	50.33	2.97	18.22	52.02
1.12	43.61	2.57	18.22	45.08

5.4 Summary of the Parameter Studies

- By
  - decreasing the span of the plate,
  - decreasing the density of the flaws,
  - decreasing the maximum size, either depth of length, of the flaws,
  - increasing the geometry correction factor

  the strength of the glass will increase.
- The higher loads a glass plate typically can bear, the greater variance of strength.
- The assumption of the Weibull modulus m being a material constant was contradicted in the studies relating to flaw density and the span of the plate. With more flaws on a plate geometry m would increase.
- The studies did not indicate the Weibull modulus m being dependent on flaw geometry.
- The Normal distribution and Weibull distribution yield similar results.
- This parameter study found the Weibull distribution most descriptive of more of the resulting distributions than the Normal distribution.

Chapter 6

Case Studies

This chapter presents the case studies performed in this thesis. The case studies was performed with the purpose of examining the behavior of the model under different loading conditions, with reference to performed experimental tests. The analysis in the first sections are run for 2400 plates with the Python code. In the final section a case study is performed to visualize crack propagation, and for this the FORTRAN code was used.

6.1 Four point bending

A series of four point bending tests of different geometries were performed in the laboratories of the department of Structural Engineering and reported in [7]. This section presents a case study on two of these geometries with the purpose of comparing the behavior of the model with some experimental results. The results are fitted with Normal and Weibull distributions to comment on the best fit. In the cumulative distributions diagrams these distributions are represented by dashed lines. Of the experiments that was performed it was the smallest and largest geometry is considered the most interesting, in order to also document the models response to different geometries compared with actual cases. The small plate is 100mm in length, 20mm in width and 4mm in depth, while the large plate is 300mm in length, 60mm in width and 4mm in depth.

Four cylinders of diameter 6mm supports and loads the glass specimens, and have a length of three times the width of the specimens. Two cylinders are supporting the plate from underneath and two cylinders are loaded on top. The supporting cylinders are placed 10mm from the ends of the plate. The free body diagram of the test setup is given in Figure 6.1.

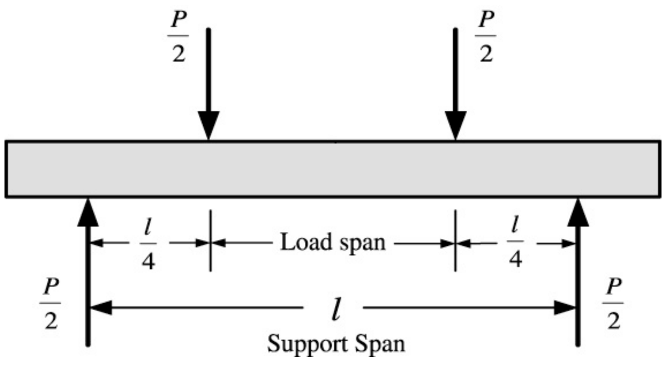


Figure 6.1: The test setup for the four point bending tests in Section 6.1.

The stresses are given by the equation

$$\sigma = \frac{3}{4}\frac{PL}{bd^2} \tag{6.1}$$

where L represents the support span as shown in Figure 6.1, P the applied load, b the plate width and d the plate height. Further, the deflection w is given by

$$w = \frac{11}{786}\frac{PL^3}{EI} \tag{6.2}$$

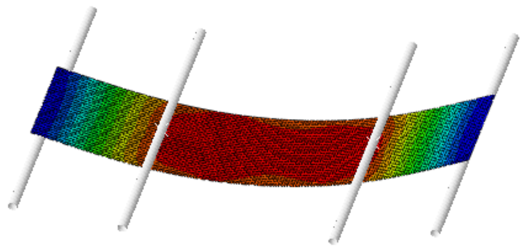
where E is the module of elasticity and I is the moment of inertia.

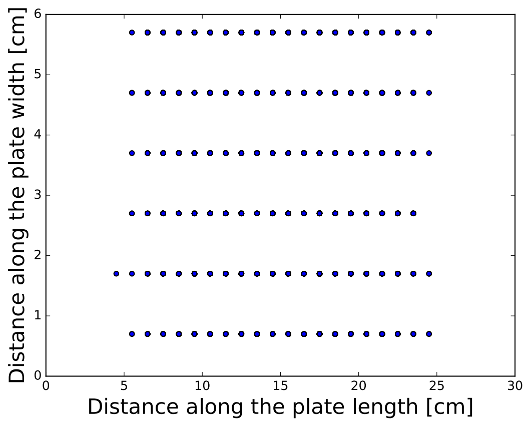
The tests are simulated quasi-statically in Abaqus/Explicit with a total time of 0.1 seconds. A surface-to-surface contact algorithm is applied between analytical rigid cylinders and three dimensional glass plate of S4R shell elements. A mesh size of 2 mm^2 is applied to the glass specimens. In the performed laboratory test, the cylinders were not restrained from rotating and this is imitated for the loaded cylinders by a frictionless interaction property. However, to prevent the glass specimen from sliding through the supporting cylinders, their interaction were given a friction coefficient of 0.6, under the assumption that this will not cause the simulation to deviate significantly from the real tests. The load applied to the top cylinders are ramped up with a "smooth step" amplitude. They are constrained from moving in any direction but vertically. The supporting cylinders are constrained from moving in any direction. In order to run the 2400-analysis faster with the Python code, the solution for linear geometrically behaving cases described in Section 3.4.4 was chosen, although the simulation behaves nonlinearly for the plates experiencing the great deflections at failure. The consequential effect on the results is however assumed small.

The following sections present the results in terms of general observations, observations regarding the shape of the distribution with respect to the Normal and Weibull distributions, and finally a discussion on possible sources of discrepancy between the model and the physical tests. The 2400-plates Python analysis is referred to as analysis results, while the experimental test series results is referred to as test results.

6.1.1 Large sample

The simulated four point bending test of the large sample is given in Figure 6.2(a). The load applied in the simulation is large in order to cover the range of critical loads for a plate of this geometry, and hence the deflection is visibly large as well. The colors visualizes the stresses, with red signifying the highest stress at the center of the plate, in accordance with Equation (6.1). The blue area visualizes that the outer 10mm of the plate are stress free. An illustration of the positions of the critical flaws of the 2400-plates analysis is presented in Figure 6.2(b), illustrating the high stresses in the area of the plate around and inbetween the loaded cylinders causes the flaws in this area to propagate and be the source of failure.


(a) Simulated four point bending for the large sample in Section 6.1.



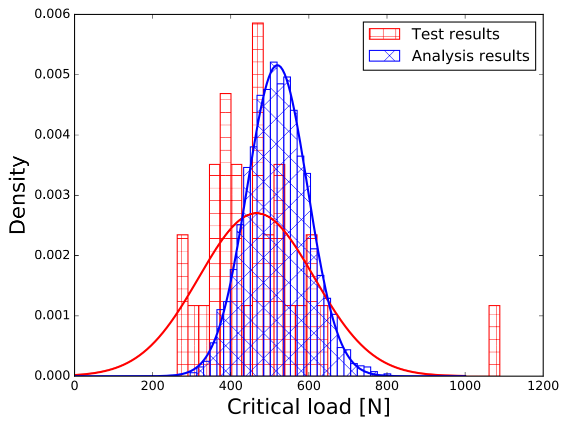
(b) Location of failures in the analysis.

Figure 6.2: The large sample.

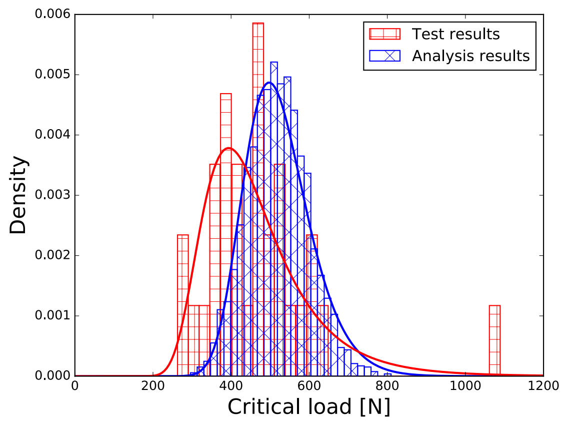
Figure 6.3 and Figure 6.4 show the distributions of the critical loads and deflections at failure, respectively, of the 2400-plates numerical analysis and the 31-plates test series from [7]. All graphs are fitted with Normal and Weibull distribution curves. The mean critical loads and deflections are given in Table 6.1, showing the analysis results yielding higher values for both compared to the test results. The mean critical load of the analysis results deviate with 11,47% from the test results, while the mean deflection deviate with 23,58%. The range of critical load curves start at approximately the same value, but the analysis curve ranges further than the test series curve, with the exception of the one test with high critical load. The variances of the deflection curves are quite similar, as is also reflected with the standard deviations in Table 6.1.

By analyzing the Figure 6.3, it is clear neither of the fitted Normal or Weibull distributions manages to capture the critical loads of the experimental test results well. This is because the experimental results stems from a series of only 31 plates which in most cases will yield a rough distribution curve. By the looks of Figure 6.3(a), it seems that the Normal distribution fails to follow the curve at the bottoms of the experimental test series distribution, while the Weibull distribution has a closer approximation shown in Figure 6.3(b). By visually inspecting the cumulative distribution curves, the case is the opposite for the distribution generated from the 2400-plates analysis. The Weibull distribution fails to fit the bottom end of the distribution, opposite to the

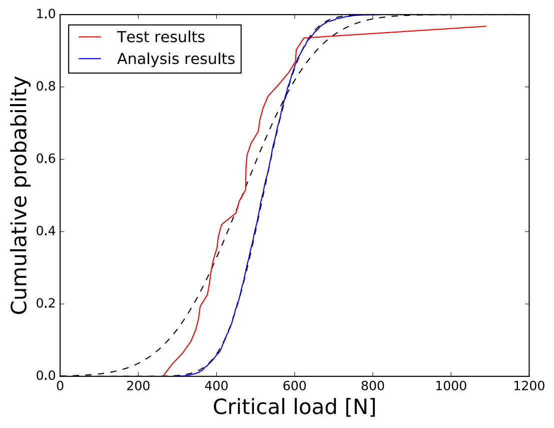
Normal distribution, and thus the latter is the better fit for the analysis results. Table 6.1 presents the Normal and Weibull parameters for the fitted critical loads distributions. The one deviating critical load for from the test series causes the Weibull modulus m to decrease, as seen in Table 6.1, as it widens the fitted Weibull distribution curve. The trouble the fitted distributions has had in fitting the experimental test series data is likely greatly influenced by the large critical value deviating from the distribution body. It is questionable if this deviating value is due to errors in the testing and should be discarded. Supporting this is that no particularly large deflection value is measured, seen by inspection of Figure 6.4, despite one plate being subjected to an particularly high load. An outcome of this is the different values for m for the test series deflections and critical loads, as seen in Table 6.1, where m is significantly lower for the critical loads as a result of the wide fitted Weibull distribution due to the deviating high load.



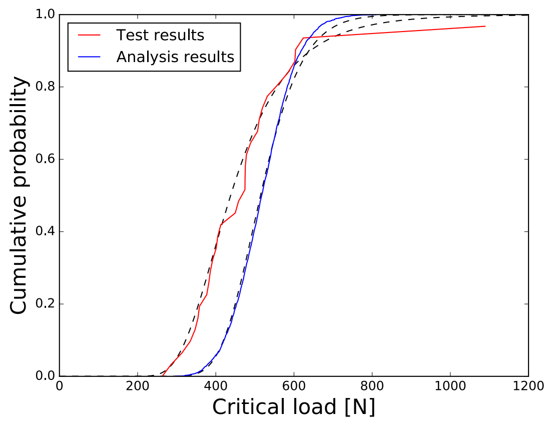
(a) Normal probability density distribution curves.



(b) Weibull probability density distribution curves.



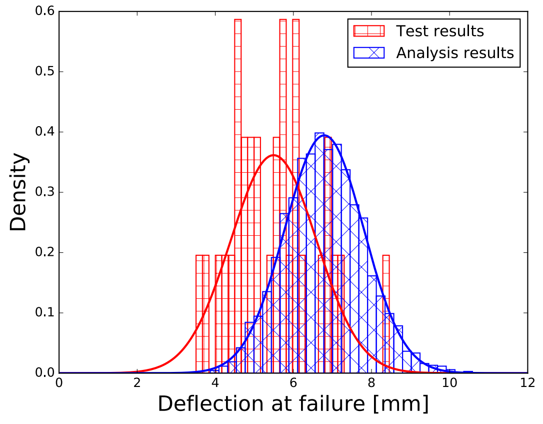
(c) Cumulative normal distribution curves.



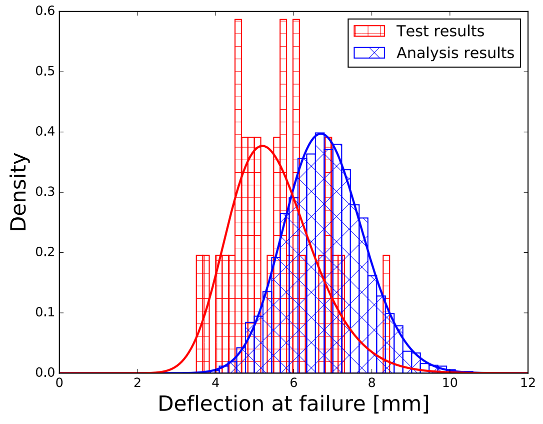
(d) Cumulative Weibull distribution curves.

Figure 6.3: Distributions of the critical loads for the large sample, fitted with Normal and Weibull distribution curves. Analysis results bin size is 17.4N and the test results bin size is 25N for the histograms.

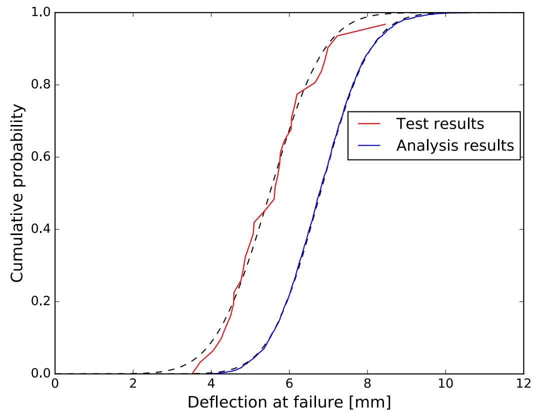
By assessing the deflections of the test series distribution in Figure 6.4, the rough curve of a small test series is evident. Also with deflection it seems that the Normal distribution fails to capture the steep bottoms of the test series distributions, and thus the Weibull distribution seems to more accurately capture the behavior. With the 2400-plates analysis the Normal and Weibull distributions yield visually similar results. Table 6.1 presents the Normal and Weibull parameters for the fitted deflection distributions.



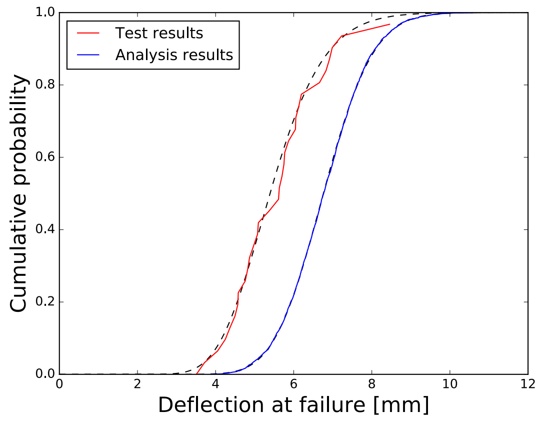
(a) Normal probability density distribution curves.



(b) Weibull probability density distribution curves.



(c) Cumulative normal distribution curves.



(d) Cumulative Weibull distribution curves.

Figure 6.4: Distribution of deflections at failure for the large specimen, fitted with Normal and Weibull distribution curves. Analysis results bin size is 0.23 mm and test results bin size is 0.17 mm for the histograms.

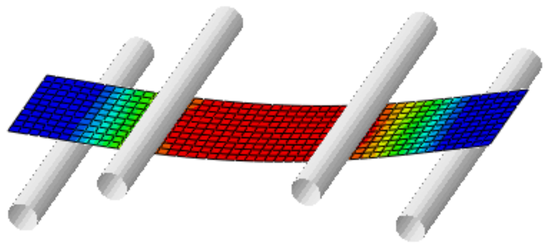
The deviation seen between the test series and the analysis of the model can be due to a number of factors. The parameters assumed for the model, for instance the penny-shaped flaw or maximum flaw length, may the inappropriate. Implementing a larger maximum flaw length would for instance return lower average load at failure, however it is not certain that this is the point where the model fails to imitate the physical behavior exactly. It is likely the simplification and measure for a speedy 2400-plates analysis, using the solution for geometrically linear solutions, have influenced the results to some degree, as was expected, although the effect was assumed to be small. The experimental tests may also be the source of deviation. For the medium samples in the analyses found in [7], which are not included in this thesis, unexpected results was explained by coarse edges of the glass. Edge-flaws are typical present from the cutting from the basic plate[38], and is not included in the current implementation of the model, which assumes only failure from surface flaws on the span of the plate. However, this is an adequate assumption since big glasses such as windows glasses are treated to reduce edge flaws, avoiding this problem[39]. This was a measure that was not taken for the samples used in he tests given in [7]. Additionally, the experiments are run for only 31 plates, resulting in an uneven distribution curve of the scattered results. Running tests on another 31 plates may yield a different result, and thus proper validation of the model can only be conducted with an extensive series of tests[6].

Table 6.1: Normal and Weibull parameters for the large specimen in the 2400-plates analysis in Section 6.1.1.

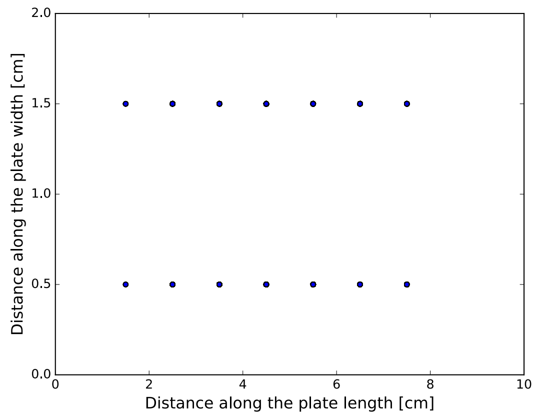
	Normal		Weibull	
	μ	σ	m	σ_0
Analysis critical load	519.31N	77.32N	7.09	553.18N
Analysis deflection	6.79mm	1.01mm	7.09	7.23mm
Test critical load	465.88N	147.61N	2.99	517.63N
Test deflection	5.49mm	1.10mm	5.18	5.95mm

6.1.2 Small sample

Figure 6.5(a) presents the simulated four point bending test for the small sample, with the stresses visualized by colors and red signifying the largest stresses, in accordance with Equation (6.1). The blue signifies a stress free state. The location of the failing flaws in the 2400-plates analysis is shown in Figure 6.5(b), in the area with the highest stresses, as shown in Figure 6.5(a). However, the location of the failing flaws is visibly not symmetric over the plate as for the large samples. This might be due to a fault in the simulation. In order to induce failure in all plates in the analysis, the applied load in the simulation is so large (4000N) that the plate nearly escapes its supporting cylinders, starting at the left cylinder. Consequently, the deflection and stresses on this side are slightly larger than for the right and thus instances of failures are wrongfully induced on this side. This effect is however considered to be small compared to the size of the analysis, and an investigation revealed that there was only 5 such deviating instances out of the 2400 failures of the analysis. Controls of maximum center stresses with Equation (6.1) and center deflection with Equation (6.2) compared to the equivalent values from the Abaqus simulation yielded somewhat lower values for the simulation corresponding to errors of 7.8 % and 6.6 %, respectively. This is however considered adequate. The occurrence of these error values indicates nonlinear geometric behavior for the applied load of 4000N. This leads to a nonlinear relationship between stresses and deflections.



(a) Simulated four point bending for the small sample in Section 6.1.

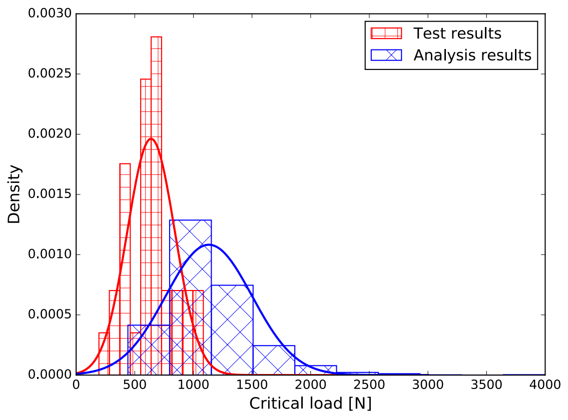


(b) Location of failures in the analysis.

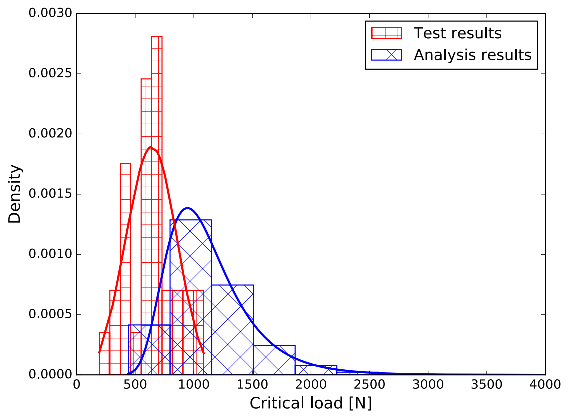
Figure 6.5: The small sample.

Figures 6.6 and 6.7 presents the critical loads and deflections at failure for the test series and the 2400-plates analysis. All distributions are fitted with the Normal and Weibull distributions. The mean critical loads and deflections are given in Table 6.2. The mean critical loads are clearly deviating as the value for the test series is nearly half the value of the analysis performed by the model. Mean deflection at failure, on the other hand, does not differ significantly and yields rather good results. The variance of the critical loads of the analysis results are greater than of the test results, while for deflection the variances are similar. For the 2400-plates analysis results, there was one instance of a plate that failed for a significantly higher load and deflection than the rest of the values associated with the distributions bodies, as can be seen in Figures 6.6 and 6.7.

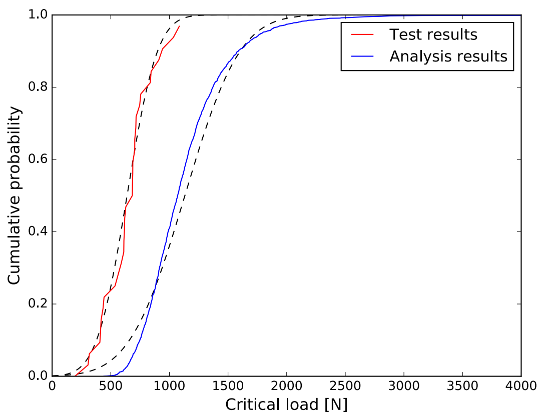
By visually inspecting Figure 6.6 it is hard to conclude the better descriptive distribution fit for the test series' critical loads. For the analysis results on the other hand, the Weibull distribution provides the better description of the curve, as the Normal distribution fails to capture the behavior at the bottom of the curve.



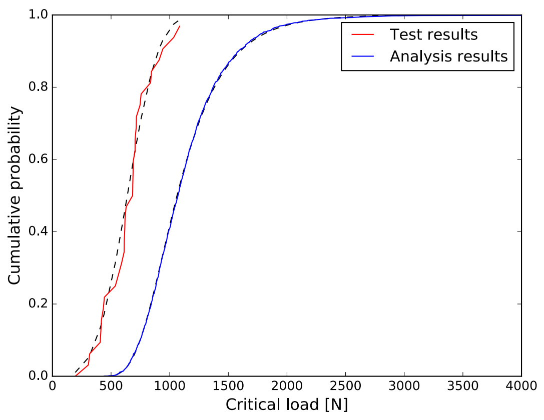
(a) Normal probability density distribution curves.



(b) Weibull probability density distribution curves.



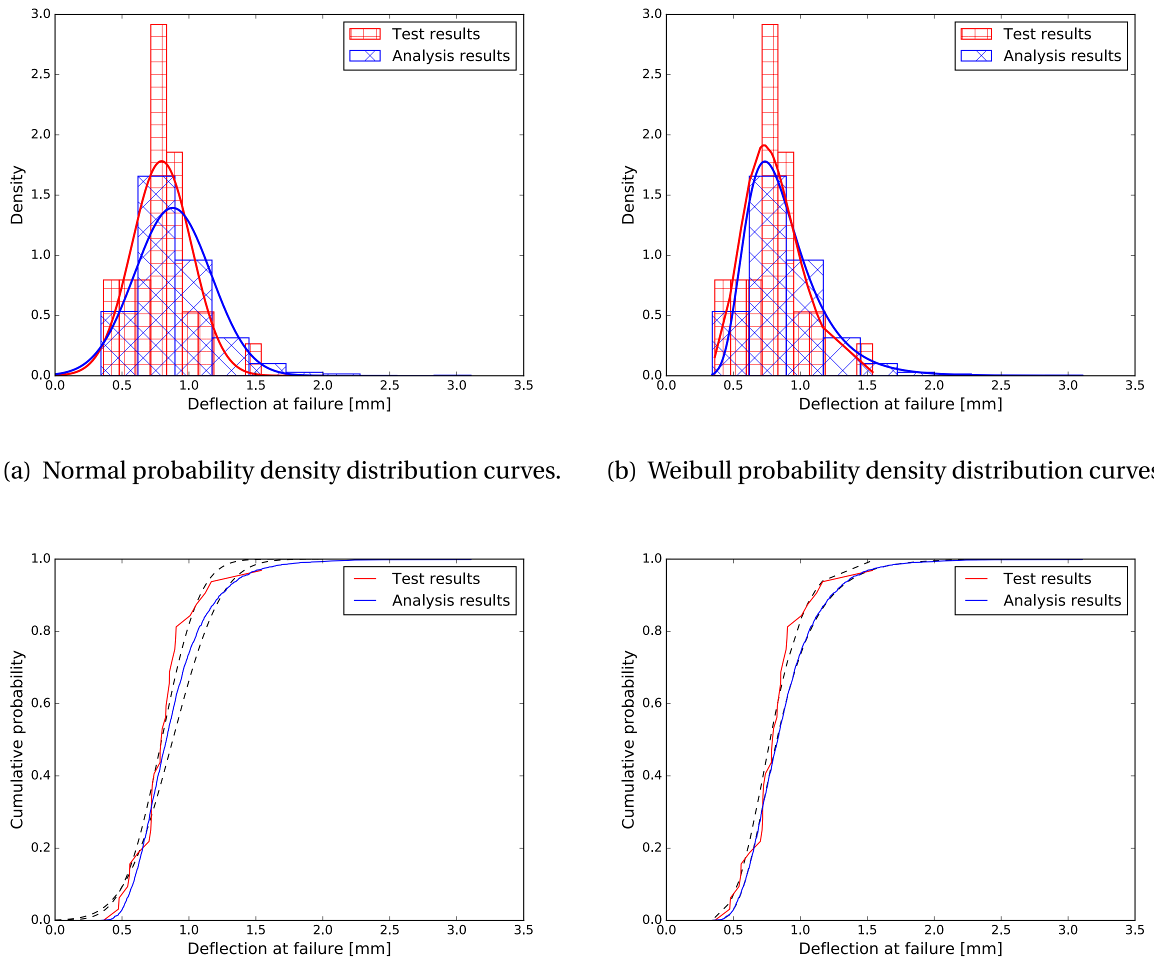
(c) Cumulative Normal distribution curves.



(d) Cumulative Weibull distribution curves.

Figure 6.6: Distributions of the critical loads for the small sample, fitted with Normal and Weibull distribution curves. Analysis results bin size is 370.4 N and test results bin size is 90.9 N for the histograms.

The Normal distribution fits of the deflection distribution curves from both the test series and the 2400-plates analysis are not providing a good description of the data, as seen in Figure 6.7. The fitted Weibull distribution manages to capture the behavior to a greater extent and is the overall better fit for the deflection distributions.



(a) Normal probability density distribution curves. (b) Weibull probability density distribution curves. .

(c) Cumulative Normal distribution curves. (d) Cumulative Weibull distribution curves.

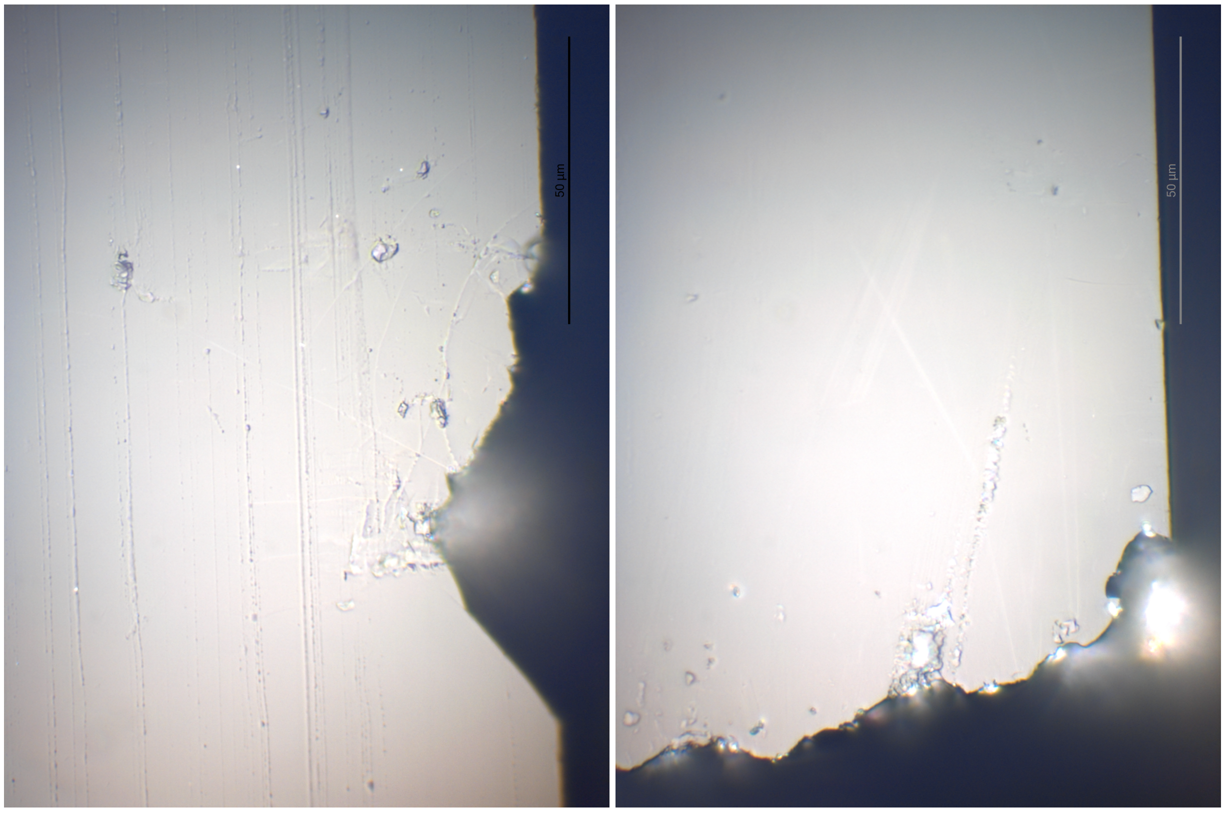
Figure 6.7: Distributions of the deflections at failure for the small specimen, fitted with Normal and Weibull distribution curves. Analysis results bin size is 0.26 mm and test results bin size is 0.12 mm for the histograms.

As with the deviations seen in the results of the large samples in Section 6.1.1, there are multiple possible reasons for the deviating results seen with the small sample. All the possible sources of discrepancy mentioned discussing the results for the large sample applies to the small sample as well, with emphasis on the edge flaws. For a smaller geometry, less mid-plate surface flaws are present. Assuming there is some density of flaws along the edges of glass, with a greater circumference to area ratio for the glass plate the edge-flaws' influence increases. For the large sample this ratio is equal to 0.04 while for the small sample ratio it is 0.12. In other words, the edge-flaws to mid-plate surface flaws ratio increases with the small sample and thus the edge-flaws may be more dominant in determining the strength in this case. This can explain how the model performs better with the large sample compared to the small. Supporting this theory are microscope pictures taken of the small samples prior to testing, shown in Figure 6.8, revealing some rough edge flaws. These flaws may have been present in the large samples as well as they were not controlled with a microscope like the small samples were. However, if the edge flaws of the large samples are of the same sizes as in the small samples, the argument of a higher edge-flaws to mid-plate surface flaws holds.

Another possible source of the discrepancies is mentioned in [7]. Some results were disregarded from the performed test series due to the discovery of a loose fastening mechanisms in the test setup, producing unrealistic results. This was adjusted while the testing was ongoing. [7] received a lower mean Young's modulus from the experiment than both expected and compared to the larger samples, and blames the loose fastening mechanism for this. Therefore, it can seem not all faulted results were disregarded and thus may be the cause of some deviation in the experimental results presented in this section as well.

Table 6.2: Normal and Weibull parameters for the small specimen in Section 6.1.2.

	Normal		Weibull	
	μ	σ	m	σ_0
Analysis critical load	1131.79N	368.57N	3.00	1259.46
Analysis deflection	0.87mm	0.23mm	3.00	0.97
Test critical load	639.01N	203.23N	3.47	710.36N
Test deflection	0.79mm	0.22mm	3.58	0.87mm



(a) (b)

Figure 6.8: Edge flaws on the small sample.

6.1.3 Summary of the Four Point Bending analysis

- For large geometries the model produces results similar to the test series data.
- It is likely that edge flaws become dominant for small geometries with untreated edges, a factor not considered by the model, and hence the model results will naturally deviate.
- The Weibull distribution describes both the experimental test data and the analysis data more accurately than Normal distributions.
- As also seen in the plate geometry parameter study in Section 5.1, with a smaller geometry the specimen endures higher loads. Due to a lower prevalence of large flaws in small specimens, the scatter in strength increases. This is reflected by the Weibull modulus m

from the analysis results presented in Table 6.1 and 6.2, where m is significantly higher for the small specimen in comparison with the large. This indicates treating m as a material constant is erroneous.

- The similarity in the spread of the distributions for the larger specimens indicates that the exponential distribution of flaw lengths being a good model for flaw length occurrence.

6.2 Blast wave

This section examines the response of the model in a blast loading situation. This study is based on some experiments that was conducted at the SIMlab shock tube facility at the Institute of Structural Engineering in the fall of 2016. The test setup is similar to the illustration in Figure 6.10. The 400×400mm^2 plate is clamped between rubber supports at the outer edges such that the inner 300×300mm^2 area is untouched by the rubber. The thickness of the glass is 4mm. It was reported that the glass was exposed to a blast load of 79 kPa without failing. However, an equivalent experiment yielded failure for approximately 83 kPa. The calibrated values for the Friedlander equation explained in Section 2.5 is given in Table 6.3. Figure 6.9 shows the corresponding load history.

Table 6.3: Friedlander Equation calibrated values.

P$_r$	79 kPa
t$_+$	13 ms
b	7.7E-01

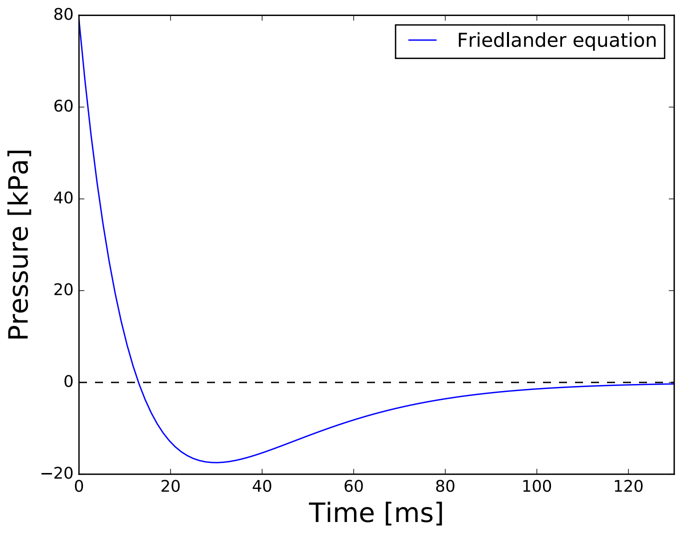


Figure 6.9: The pressure curve corresponding to the calibrated values for the Friedlander equation in Table 6.3.

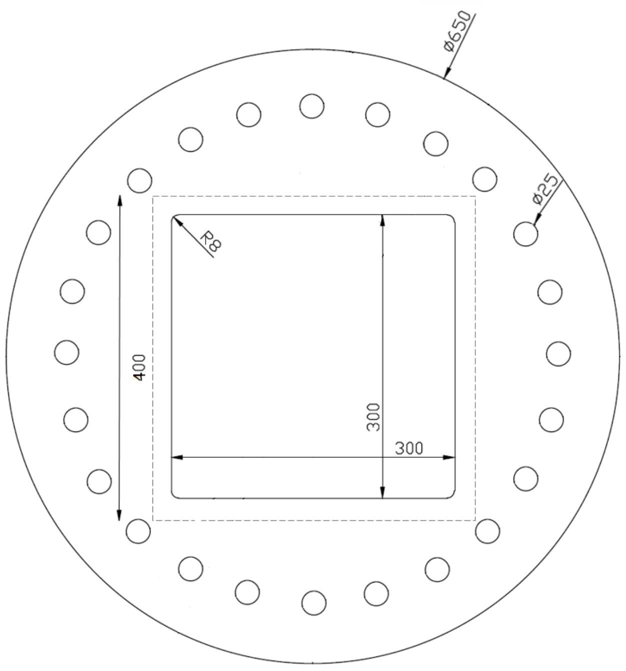


Figure 6.10: Test setup, retrieved from [40].

The experiment was simulated in Abaqus/Explicit. The geometry and boundary conditions of the plate was approximated by using a 350m x 350m simply supported plate, simulated as a 3D deformable shell. The boundary conditions was determined on the basis that the clamped areas of the glass' was able compress the rubber to some degree. A pressure load was applied to the model, with the purpose of failing all plates in the subsequent Monte Carlo analysis. The load was ramped up to maximum P during the first 4% of the total time of 0.1 seconds, such that the load is applied almost instantly. The amplitude uses five points in representing the blast load curve, as can be seen in Figures 6.11 and 6.12. As also seen in these figures is the oscillating stresses which requires an ascending order of the time steps, not eliminating any flaws as first failing flaw candidates, by use of the Python code.

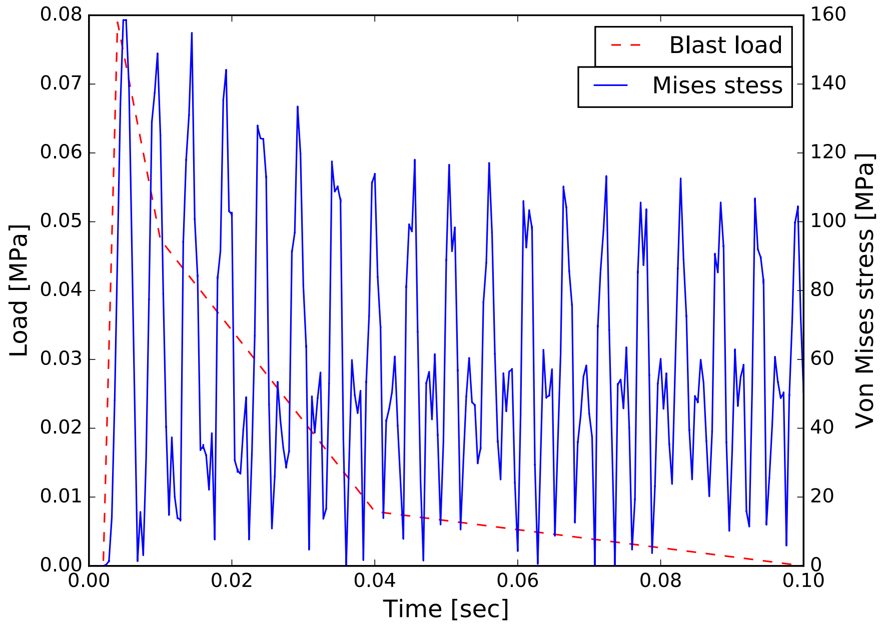


Figure 6.11: Pressure versus center von Mises stresses for the 79 kPa blast load.

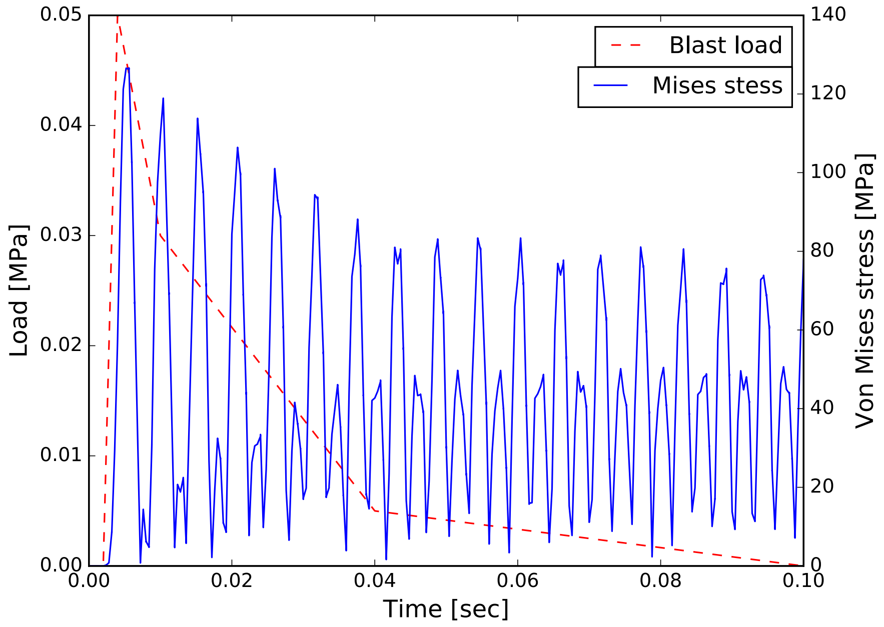
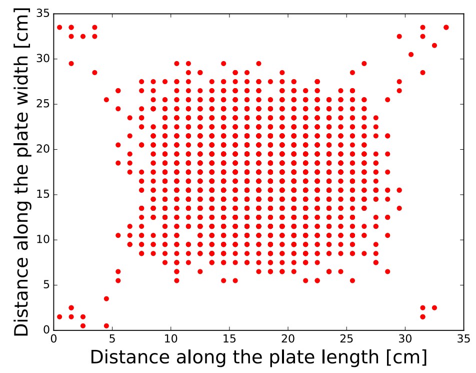


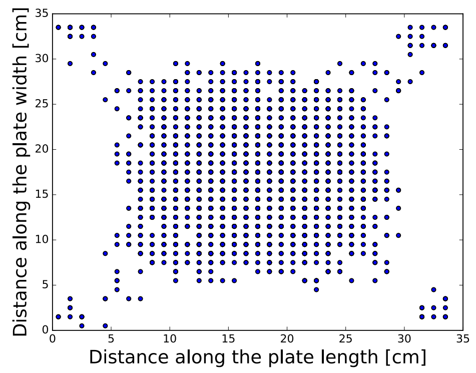
Figure 6.12: Pressure versus center von Mises stresses for the 50 kPa blast load.

Two different peak loads of 79 kPa and 50 kPa amplitudes were simulated with Abaqus. The blast load of 79 kPa was chosen in reference to the SIMlab experiment. This load induced failure for all plates in the analysis with many failing at peak load, as can be seen in Figure 6.13(d), and therefore the 50 kPa load was further tested to examine the behavior for lower loads. The same 2400 flaw maps were used in both analyses. The loads are plotted together with the resulting center stresses over time in Figures 6.11 and 6.12. The peak load amplitude for both diagrams occurs after 0.004 seconds, while the Mises stress reaches its greatest value at 0.005 seconds. This is due to the delay of deformation from the almost instantly applied blast load.

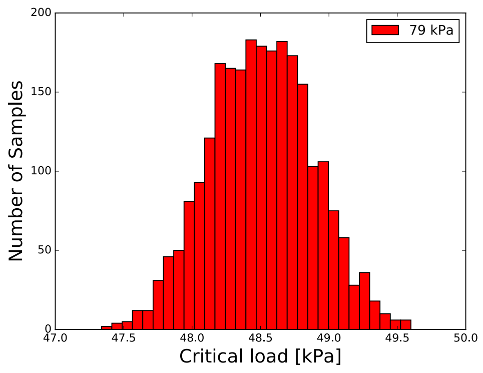
Figure 6.13 shows the resulting locations of initial fracture from the first failing flaws and the critical loads and critical tensile stresses diagrams for both loads. Figures 6.13(a) and 6.13(b) show the locations of fracture occur in the same areas, although the figures implies a greater spread of first failing flaws for the higher load. This is natural as the loads are applied at a higher rate. The curves of tensile stresses at failure have approximately the same shape and mean value. As the location of initial fracture are in the same areas, in the center and in the corners of the plate, it is likely that many of the same flaws are critical in both analyses. Thus it is reasonable that the critical tensile stress diagrams are similar. The main difference of the two cases lies with the critical loads at failure. For the greater load, the initial stresses at peak load, prior to the delayed stress peak are for many plates critical, and hence failure occurs at impact. There are plates not failing at impact, as seen in Figure 6.13(d), and they can have failed both before or after the event of the peak load. For the case with a peak load of 50 kPa, the stresses occurring at maximum applied load are not critical for any of the plates and thus no failure occurs for 50 kPa. The following response of the plate will however induce failure as the delayed stresses are greater, as can be seen in Figure 6.12. Therefore the distribution will have a more symmetric body, as seen in Figure 6.13(c), as opposed to the case where plates fails instantly at peak load. By examining the critical load intervals for both blast loads in Figures 6.13(d) and 6.13(c) and comparing to the stresses and loads over time in Figures 6.11 and 6.12, it seems all plates fails before or at the point where the stresses are the highest.



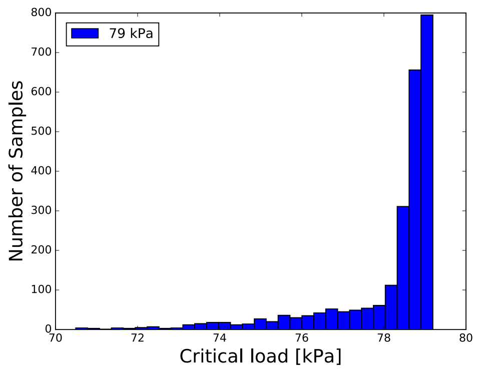
(a) Flaw map (50 kPa load)



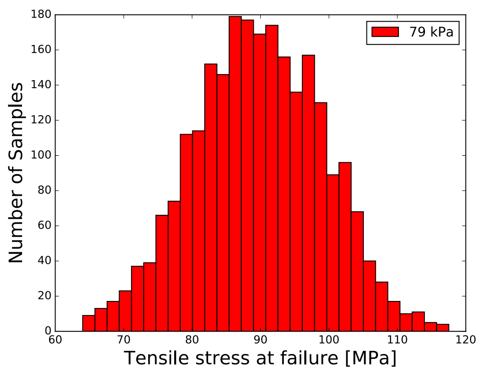
(b) Flaw map (79 kPa load)



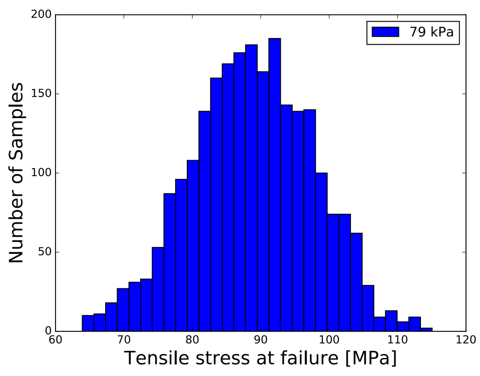
(c) 50 kPa load. Bin size: 0.07 kPa.



(d) 79 kPa load. Bin size: 0.28 kPa



(e) 50 kPa load. Bin size: 1.82 MPa



(f) 79 kPa load. Bin size: 1.7 MPa

Figure 6.13: The $350 \times 350 \times 4$ mm^3 SSP exposed to blast waves of 50 kPa and 79 kPa.

As mentioned, two experimental tests with blast loads of 79 kPa and 83 kPa have been performed, where only the latter induced failure in the glass. This is not consistent with the analysis results where all 2400 plates in the analysis run with the Python code failed after being applied with a 50 kPa blast load. This may due to an inappropriate geometry and boundary conditions in the simulation. The maximum deflection in the experimental test with applied load 79 kPa was approximately 6 mm while the maximum deflection in the simulation with the same load applied was 11,4 mm. This indicates a more stiff behavior in the experimental tests. It is not certain that a simply supported specimen of $350\times350\times4$ mm^3 is appropriate. Clamped edges may provide a better description of the behavior of the plates in the experimental tests. Reducing the plate geometry to $300\times300\times4$ mm^3 may also enhance the simulations in regards to the experiments. It is also uncertain whether the thickness of the plate were 4mm or a few mm thinner. Reducing the thickness will however result in reduced stiffness of the plate and lead to a greater deflection, i.e. this is not what causes the simulations to behave differently than the plates in the experiment.

6.3 The FORTRAN code applied to the case studies

The FORTRAN code was applied to two of the case studies in this chapter with the purpose of visualizing crack propagation. These were the large sample four point bending test of Section 6.1.1 and the blast wave case of 79 kPa in Section 6.2.

Four point bending test

Due to the limitation of the program deleting elements only when all section points of an integration point are fulfilling the requirements for failure, as explained in Section 3.5, no elements were deleted with respect to the Status field variable for the large sample in the four point bending test. This is because the always compressive stresses on the top surface of the plate will not induce failure, only the tensile stresses on the bottom. Thus visualizing crack propagation was impossible in this case with the current version of the subroutine. However, one-sided failure was detected and by specifying in Abaqus/Viewer that element deletion should occur at this point, it is possible to visualize the origin of failure with the FORTRAN code. Figure 6.14 shows the location of failure on a color map showing the Mises stresses with red signifying the highest stresses.

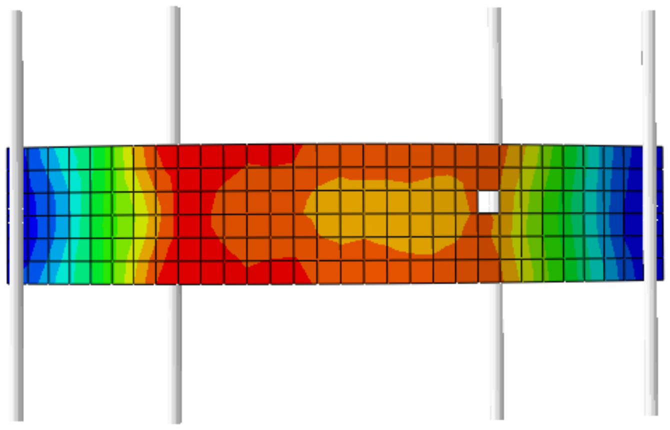


Figure 6.14: The first failing flaw visualized with the FORTRAN subroutine.

Blast wave

For the blast wave case, the plate displacement and the stresses will oscillate. At some point with this behavior, there are tensile stresses on both sides of an element of the plate which causes failure and the requirement for element erosion is fulfilled. From here the stresses relocates and concentrates around the eroded element and consequently neighbor elements are eroded. Figure 6.15 shows this effect illustrating crack propagation over the plate after exposure to a blast load. The colors visualizes the Mises stress concentrations, with red signifying the highest stresses, showing increased stresses in neighbor elements to the crack. It is emphasized that this case of crack propagating might not be physically correct as fracture in fact occurs from one side, while the FORTRAN code only recognized the fracture when both sides of the elements registers fracture.

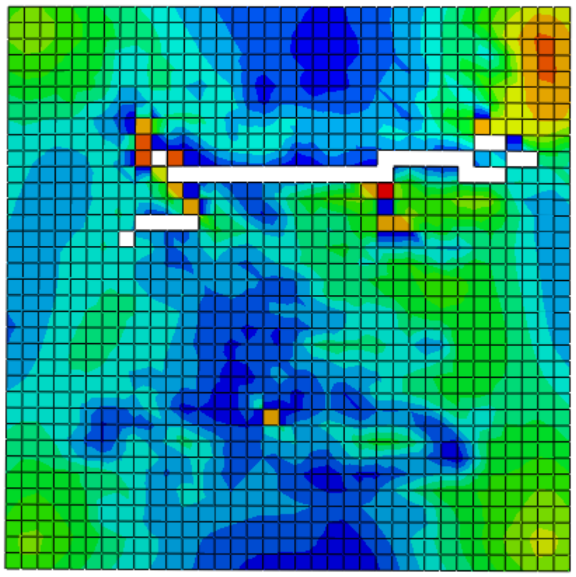


Figure 6.15: Crack propagation visualized with the FORTRAN subroutine.

Chapter 7

Concluding remarks

This thesis presents the foundation, implementation and verification of a strength model for float glass, first presented by Yankelevsky in [6] and further developed herein by using a geometry correction factor corresponding to the flaw depth to length ratio and including the flaws' orientations. The model provides information of the origin of failure in the glass and the probability distribution as a result of the analysis and not assumed a priori.

In order to thoroughly explain the framework of the strength model, the mathematical and mechanical foundation was first presented, as well as background on the the glass production process. Grouped together with the explanation of the model, was the rendering of the three software versions of the model written in MATLAB, Python and FORTRAN. This structure was chosen because of the tight relationship between the implementations and the strength model. The main implementations, namely the MATLAB and Python scripts, are elaborated with limitations and optimizations, and pseudo codes are provided for a better understanding of the main structure of the programs.

Further, the verification processes of the three implementations was presented. The MATLAB code was implemented with the purpose of serving as an analytical solution in verifying the Python and FORTRAN codes. It was verified through comparing center stresses and deflections with corresponding solutions in Abaqus FEA. The verification of the Python code yielded satisfying results. As did the FORTRAN code, except for one discrepancy which was reasoned to not be associated with the model or faults in the source code.

After having verified the Python code, a parameter study was performed on this version of the model, documenting its behavior with respect to geometric variables. The response was in accordance with natural behavior of failing glass. Additionally it was found that for the glass plates with less flaws, fitted Weibull distributions would have a lower Weibull modulus m, suggesting it is inaccurate to assume m is a material constant.

Further, a selection of case studies of glass in different loading conditions were performed. The model was compared with results from four point bending tests conducted by Brekken and Ingier[7], yielding acceptable results for the large specimen, however poor for the small specimen. This was reasoned by some rough edge flaws from cutting the glass becoming dominant in the small geometry, which was supported by microscopic pictures of the specimen from the experiments. The response of a glass plate exposed to different blast loads was also examined. The results showed that glass plates exposed to high loads is likely to fail at peak load, while for lower loads failure is likely to occur after peak load.

The work presented in this thesis shows that the Yankelevskys' strength model for float glass has great potential for processed glass plates. Ideas for further development of the model is presented in Chapter 8.

Chapter 8

Future work

The software development of the model in this thesis was limited by time, thus some ideas for possible improvement remain unexplored. This chapter introduces some ideas for possible improvements on both the model and the software implementations of the model.

Clustering method

As described in Section 3.4, the Python and FORTRAN codes read stress values from the center of every element. Since the model has a structured flaw density with one flaw in the center of every cm^2, these codes required initially a mesh of cm^2 elements in the Abaqus simulations. For the small specimen in the four point bending test in Section 6.1 this mesh turned out to be too coarse for the contact problem between the glass plates and the rolling cylinders. The distance between the nodes became too large such that the cylinders lost contact with plate nodes as they began rolling. The quick solution chosen for the Python script, which was used in this four point bending test, was to develop the code to accept a 2×2 mm^2 mesh and to retrieve the center stresses from the center element of every cm^2. However, similar problems may still arise for small geometries, and accuracy of the finite element solution will improve with a finer mesh, thus removing the rigid requirement of either the a cm^2 or a 2×2 mm^2 mesh is desirable.

This can be achieved by implementing a clustering method, interpolating the stresses between the elements. This allows for stresses to be retrieved at a point on the plate, independent of the mesh[41]. A clustering method would improve both the Python and FORTRAN codes.

Interacting flaws

The strength model implemented in this thesis uses a flaw map with a density of one flaw per cm^2. This is a simplification made to avoid interaction between flaws such that a single flaw is responsible for the local conditions causing failure. However, as discussed in Section 5.1, a structured flaw map is not realistic, and in an unstructured flaw map instances of flaws interacting are likely to occur. Applying to the model the theory related to the shielding and amplifying effect of flaws, presented in Section 2.1.3, may yield more realistic results. This model extension is suggested implemented to explore the model's potential. If no clustering method is implemented, it makes sense to use an unstructured mesh in the Abaqus simulations for the Python and FORTRAN codes.

Validation

The purpose of this thesis was exploring the potential of Yankelevskys' model. Performing a validation process on the software implementations of the MATLAB and Python codes would reveal whether the model is actually viable for determining glass strength. A validation process was not included as the data at hand was considered non-sufficient to provide proper validation of the implementations. To conclude the model implementations are behaving equally to real situations, extensive experimental test series should be used to compare. Four point bending test data of approximately 30 samples per test was compared to model results in Section 6.1, but this number was considered too small to conclude the model was behaving correctly.

Bibliography

[1] Nina Ivanovna Min'ko and Vladimir Mikhailovich Nartsev. Factors affecting the strength of the glass (review). *Middle-East Journal of Scientific Research*, 18(11):1616–1624, 2013.

[2] L Afferrante, M Ciavarella, and E Valenza. Is weibull's modulus really a material constant? example case with interacting collinear cracks. *International Journal of Solids and Structures*, 43(17):5147–5157, 2006.

[3] J Jeong, H Adib-Ramezani, and G Pluvinage. Tensile strength of the brittle materials, probabilistic or deterministic approach? *Strength of materials*, 38(1):72–83, 2006.

[4] J.B. Wachtman, W.R. Cannon, and M.J. Matthewson. *Mechanical Properties of Ceramics*. Wiley, 2009.

[5] Bikramjit Basu, Devesh Tiwari, Debasis Kundu, and Rajesh Prasad. Is weibull distribution the most appropriate statistical strength distribution for brittle materials? *Ceramics International*, 35(1):237–246, 2009.

[6] David Z. Yankelevsky. Strength prediction of annealed glass plates - a new model. *Engineering Structures*, 79:244–255, 2014.

[7] KA Brekken and TP Ingier. Modelling of window glasses exposed to blast loading. Master's thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2016.

[8] TL Anderson. *Fracture Mechanics, Fundamentals and Applications*. Taylor and Francis, Boca Raton, FL, 3rd edition, 2011.

[9] M.F. Ashby and D.R.H. Jones. *Engineering Materials 1: An Introduction to Properties, Applications and Design.* Number v. 1 in Butterworth-Heinemann. Butterworth-Heinemann, 2011.

[10] Wikipedia. Fracture mechanics — wikipedia, the free encyclopedia, 2016. [Online; accessed 2-November-2016].

[11] Wikipedia. Monte carlo method — Wikipedia, the free encyclopedia, 2016. [Online; accessed 07-November-2016].

[12] D. Cook and C. Young. *Advanced Mechanics of Materials.* Pearson, 2nd edition, 1998.

[13] A. Reyes. Lecture 13: Klassisk tynnplateteori. Online; accessed 10-November-2016, November 2013.

[14] RL Taylor and S Govindjee. Solution of clamped rectangular plate problems. *Communications in Numerical Methods in Engineering*, 20(10):757–765, 2004.

[15] A. Reyes. Lecture 14: Plateberegninger. Online; accessed 10-November-2016, November 2013.

[16] Wikipedia. Glass — wikipedia, the free encyclopedia, 2017. [Online; accessed 2-January-2017].

[17] MF. Ashby and DR. Jones. *Engineering Materials 2, an introduction to microstructures, processing and design.* Butterworth Heinemann, 2nd edition, 1998.

[18] H.J. Herrmann and S. Roux. *Statistical models for the fracture of disordered media.* Random materials and processes. North-Holland, 1990.

[19] The float process. [Online; accessed 15-November-2016].

[20] Wikipedia. Annealing (glass) — wikipedia, the free encyclopedia, 2016. [Online; accessed 26-September-2016].

[21] EN. Glass in building - basic soda lime silicate glass products - Part 2: Float glass. Standard, CEN, 2004.

[22] V Aune, T Børvik, and M Langseth. Impact mechanics: An introduction to blast mechanics. Unpublished draft., 2016.

[23] Wikipedia. Blast wave — wikipedia, the free encyclopedia, 2016. [Online; accessed 19-November-2016].

[24] JK Lancaster. Crack propagation and particle detachment in the wear of glass under elastic contact conditions. *Interface Dynamics, Tribology Series*, 12:111–119, 1988.

[25] Naimeh Khorasani. *Design principles for glass used structurally*. Department of Building Science, Univ., 2004.

[26] Andrew A Wereszczak, Mattison K Ferber, and Wayne Musselwhite. Method for identifying and mapping flaw size distributions on glass surfaces for predicting mechanical response. *International Journal of Applied Glass Science*, 5(1):16–21, 2014.

[27] Wikipedia. Plane stress — wikipedia, the free encyclopedia, 2016. [Online; accessed 6-December-2016].

[28] Oriel Goodman and Brian Derby. The mechanical properties of float glass surfaces measured by nanoindentation and acoustic microscopy. *Acta Materialia*, 59(4):1790–1799, 2011.

[29] Jianghong Gong, Yufeng Chen, and Chunyan Li. Statistical analysis of fracture toughness of soda-lime glass determined by indentation. *Journal of Non-Crystalline Solids*, 279(2):219–223, 2001.

[30] T.H. Cormen. *Introduction to Algorithms*. MIT Press, 2009.

[31] Abaqus 6.14 documentation, 2014. Online; accessed 1-Desember-2016.

[32] KM Mathisen. Solution of the dynamic equilibrium equations by explicit direct integration. Online; accessed 1-November-2016, 2015.

[33] Y. Chen and University of South Carolina. *Optimization of the Hybrid RC/FRP Beam System*. University of South Carolina, 2007.

[34] FS Loureiro, JEA Silva, and WJ Mansur. An explicit time-stepping technique for elastic waves under concepts of green's functions computed locally by the fem. *Engineering Analysis with Boundary Elements*, 50:381–394, 2015.

[35] Abaqus Inc. Lecture 9, material damage and failure, 2005. Online; accessed 10-Desember-2016.

[36] F Irgens. *Fasthetslære*. Tapir akademisk forlag, 7th edition, 2006.

[37] FA Veer, Pieter Christiaan Louter, and FP Bos. The strength of annealed, heat-strengthened and fully tempered float glass. *Fatigue & Fracture of Engineering Materials & Structures*, 32(1):18–25, 2009.

[38] Maria Lindqvist, Marc Vandebroek, Christian Louter, and Jan Belis. Influence of edge flaws on failure strength of glass. In *12th International Conference on Architectural and Automotive Glass (Glass Performance Days 2011)*, pages 126–129. Glass Performance Days, 2011.

[39] Joseph B Kelly. Edging glass sheets with diamond wheels, November 11 1986. US Patent 4,621,464.

[40] Vegard Aune, Egil Fagerholt, Magnus Langseth, and Tore Børvik. A shock tube facility to generate blast loading on structures. *International Journal of Protective Structures*, 7(3):340–366, 2016.

[41] CA Duarte, TJ Liszka, and WW Tworzydlo. Clustered generalized finite element methods for mesh unrefinement, non-matching and invalid meshes. *International journal for numerical methods in engineering*, 69(11):2409–2440, 2007.

Appendix A

MATLAB source code

```
function [N] = basicPlateSpan(Length, width)
%@INPUT:
%Length: Length of plate [m]
%width: width of plate [m]
%@OUTPUT:
%N: N*Length*width = close to 6*3.21m, where N=integer

desiredArea = 6*3.21;
approx = desiredArea/(Length*width);
approx1 = ceil(approx);
approx2 = floor(approx);
while(rem(sqrt(approx1),1) ~= 0)
    approx1 = approx1 + 1;
end
while(rem(sqrt(approx2),1) ~= 0)
    approx2 = approx2 - 1;
end
if (abs(((sqrt(approx1)*Length*width)-desiredArea)/desiredArea) >= abs(((sqrt(approx1)*Length*width
    ↪ )-desiredArea)/desiredArea))
    N = sqrt(approx2);
else
    N = sqrt(approx1);
end
end
```

```
function [flawInfoMatrix] = generatePlate(Length,width,maxFlawSize)
    % @INPUT:
    %Length: Length of basic plate (integer) [m]
    %width: width of basic plate (integer) [m]
    %maxFlawSize: max size of flaw [m]
    % @OUTPUT:
    %flawInfoMatrix: matrix where rows represent a flaw of random Length[m]
    %and angle[radians].

    %Generating flaw map
    plateArea = Length*width;
    numFlaws = plateArea/10^(-4);
    charFlawSize = maxFlawSize/log(numFlaws);
    flawInfoMatrix = zeros(int32(numFlaws),2);
    a = 1/numFlaws;
    for index = 1:numFlaws
        flawLength = (charFlawSize*(log(numFlaws) - log(numFlaws*((1-a)*rand(1)+a))))/2;
        flawAngle = pi*rand(1);
        flawInfoMatrix(index,1) = flawLength;
        flawInfoMatrix(index,2) = flawAngle;
    end
end
```

```
function [result, sigma] = willFlawPropagate(stresses, theta, a, Y, Kcrit)
%This function calculates whether a flaw will propagate or not.
% @INPUT:
% stresses: array of local plane stress components [N/m^2]
% theta: orientation of flaw [radians]
% a: length of flaw [m]
% @OUTPUT:
% result: true if flaw fails, false if it does not
% sigma: the stress normal to flaw



    sigma = stresses(1) + stresses(2) - (0.5*(stresses(1) + stresses(2)) + 0.5*(stresses(1)-
        ↪ stresses(2))*cos(2*theta) + stresses(3)*sin(2*theta));

    if (sigma < 0) %only tensile stresses please
        result = false;
        return;
    end
    K = Y*sigma*sqrt(pi*a);

    if(Kcrit <= K)
      result = true;
    else
      result = false;
    end
end
```

```
%-----------------------------------------------------------------------
%Initial conditions in terms of loading situation and geometry.
%With 'ST' a factor for the load on edge compared to the
%other can be chosen.
%-----------------------------------------------------------------------
loadSit = 'ST'; % SSP(simply supported) or CP(clamped) or ST(stretched)
if (strcmp(loadSit,'ST'))
   factor = 0;
end
width = 1; %[m]
Length = 1; %[m]
depth = 0.006; % [m]
numFlawsSP = width*100*Length*100;
midX = Length*100*0.5; %[cm]
midY = width*100*0.5; %[cm]
%------------------------------------------------------------------------
%Parameters
%------------------------------------------------------------------------
E = 70*10^9;
v = 0.22;
D = (E*(depth)^3)/(12*(1-v^2));
geoFactor = 0.6625;
Kcrit = 0.75*10^6;
%------------------------------------------------------------------------
%Series expansion
%------------------------------------------------------------------------
numMSeries = 30;
numNSeries = 30;
totNumSeries = numNSeries*numMSeries;
```

```
%------------------------------------------------------------------------
%Generating the basic plate parameters.
%------------------------------------------------------------------------
numBP = 62; %number of basic plates in analysis
N = basicPlateSpan(Length, width);
bpWidth = N*width; %m
bpLength = N*Length; %m
maxFlawSize = 200*10^(-6); %m
numSP = N*N; %number of small plates per basic plate
%------------------------------------------------------------------------
%Generating stiffness matrix for the calculation for clamped plate.
%------------------------------------------------------------------------
if (strcmp(loadSit,'CP'))
   Kmatrix = K(Length,width,numMSeries,numNSeries);
end
%------------------------------------------------------------------------
%Calculating part of the plate equation for clamped and simply
%supported.
%------------------------------------------------------------------------
xPositions = zeros(int32(Length*100),1);
yPositions = zeros(int32(width*100),1);
sigmaXXm = cell(1,numFlawsSP);
sigmaYYm = cell(1,numFlawsSP);
tauXYm = cell(1,numFlawsSP);
for numFlaw = 1:numFlawsSP
    xindicator = mod(numFlaw,(Length*100));
    if (xindicator == 0)
        xindicator = Length*100;
    end
    yindicator = ceil(numFlaw/(Length*100));
    X = xindicator-0.5;
    Y = yindicator-0.5;
    xPositions(numFlaw) = X;
    yPositions(numFlaw) = Y;
    if (strcmp(loadSit,'SSP')|| (strcmp(loadSit,'CP')))
        sigmaXXv = zeros(1,totNumSeries);
        sigmaYYv = zeros(1,totNumSeries);
        tauXYv = zeros(1,totNumSeries);
        index = 1;
        X = X*10^-2;
        Y = Y*10^-2;
        for n = 1:numMSeries;
            for m = 1:numNSeries;
                    if (strcmp(loadSit,'SSP'))
                        sigmaXXv(index) = -(depth*0.5*E)/(1-v^2) * (-(m^2*pi^2*sin((pi*m*X)/(Length
                            ↪ ))*sin((pi*n*Y)/(width)))/(Length^2) + v*-(n^2*pi^2*sin((pi*m*X)/(
                            ↪ Length))*sin((pi*n*Y)/(width)))/(width^2));
                        sigmaYYv(index) = -(depth*0.5*E)/(1-v^2) * (-(n^2*pi^2*sin((pi*m*X)/(Length
                            ↪ ))*sin((pi*n*Y)/(width)))/(width^2) + v*-(m^2*pi^2*sin((pi*m*X)/(
                            ↪ Length))*sin((pi*n*Y)/(width)))/(Length^2));
                        tauXYv(index) = -(depth*0.5*E)*(1-v)/(1-v^2) * ((m*n*pi^2*cos((pi*m*X)/(
                            ↪ Length))*cos((pi*n*Y)/(width)))/(Length*width));
                    elseif (strcmp(loadSit,'CP'))
                        sigmaXXv(index) = -(depth*0.5*E)/(1-v^2) * (-(4*pi^2*m^2*cos((2*pi*X*m)/
                            ↪ Length)*(cos((2*pi*Y*n)/width) - 1))/Length^2 + v*(-(4*pi^2*n^2*cos
                            ↪ ((2*pi*Y*n)/width)*(cos((2*pi*X*m)/Length) - 1))/width^2));
                        sigmaYYv(index) = -(depth*0.5*E)/(1-v^2) * (-(4*pi^2*n^2*cos((2*pi*Y*n)/
                            ↪ width)*(cos((2*pi*X*m)/Length) - 1))/width^2 + v*(-(4*pi^2*m^2*cos
                            ↪ ((2*pi*X*m)/Length)*(cos((2*pi*Y*n)/width) - 1))/Length^2));
                        tauXYv(index) = -(depth*0.5*E)*(1-v)/(1-v^2) * ((4*pi^2*m*n*sin((2*pi*X*m)/
                            ↪ Length)*sin((2*pi*Y*n)/width))/(Length*width));
                    end
                    index = index + 1;
            end
        end
        sigmaXXm{numFlaw} = sigmaXXv;
        sigmaYYm{numFlaw} = sigmaYYv;
        tauXYm{numFlaw} = tauXYv;
    end
end

%--------------------------------------------------------------------------
%MAIN STRUCTURE
%--------------------------------------------------------------------------
failedFlawInfo = [];
forMonteCarlo = [];
check = [];
```

```
for bP = 1:numBP
    %-------------------------------------------------------------------
    %Generating the flaw map for both sides of the current
    %basic plate.
    %-------------------------------------------------------------------
    flawInfo = generatePlate(bpLength, bpWidth, maxFlawSize);
    flawInfo2 = generatePlate(bpLength,bpWidth,maxFlawSize);
    for sP = 1:numSP
        %---------------------------------------------------------------
        %Isolating the flaws for this plate
        %---------------------------------------------------------------
        startIndex = 1+(sP-1)*numFlawsSP;
        flawsInSP = flawInfo(startIndex:(startIndex+numFlawsSP-1),:);
        flawsInSP2 = flawInfo2(startIndex:(startIndex+numFlawsSP-1),:);
        %---------------------------------------------------------------
        %Generate text file for the explicit Abaqus model
        %---------------------------------------------------------------
        if (sP==1)
            partname = 'Plate-1';
            filename = sprintf('initial_conditions%d.inp',sP);
            fp = fopen(filename,'w');
            fprintf(fp,'*initial␣conditions,type=SOLUTION\n');
            fprintf(fp,'**Elnb,␣␣␣␣␣␣␣␣␣␣␣␣a,␣␣␣␣␣␣␣theta,␣nfail,K/Kcrit,status\n');
            for i=1:size(flawsInSP)
                fprintf(fp,[partname '.%g,%6d,%6d,%6d,%7d,%6d\n'],i,flawsInSP(i,1)*1000,flawsInSP(i
                    ↪ ,2),0,0.0,1.0);
            end
            fclose(fp);
        end
        %---------------------------------------------------------------
        % Generate text files for Python post processing
        %---------------------------------------------------------------

          filename = 'flawInfoForPlate';
          filenametxt = strcat(filename,'.txt');
          if (exist(filename, 'file') && (bP == 1) && (sP == 1))
              disp('file␣was␣deleted');
              delete filenametxt;
          end
          if (bP == 1 && sP == 1)
              file = fopen(filenametxt,'w');
          end
          for i = 1:size(flawsInSP,1)
              fprintf(file,'%1.9f␣%1.5f\r\n',flawsInSP(i,1),flawsInSP(i,2));
          end
          if (bP == numBP && sP == numSP)
              fclose(file);
          end

          filename2 = 'flawInfoForPlate-2';
          filenametxt2 = strcat(filename2,'.txt');
          if (exist(filename2, 'file') && (bP == 1) && (sP == 1))
              disp('file␣was␣deleted');
              delete filenametxt2;
          end
          if (bP == 1 && sP == 1)
              disp('?PNER␣FIL2');
              file2 = fopen(filenametxt2,'w');
          end
          for i = 1:size(flawsInSP,1)
              fprintf(file2,'%1.9f␣%1.5f\r\n',flawsInSP2(i,1),flawsInSP2(i,2));
          end
          if (bP == numBP && sP == numSP)
              disp('LUKKER␣FIL');
              fclose(file2);
          end
      %--------------------------------------------------------------

      %--------------------------------------------------------------
        %Initiating estimated critical load for current plate and
        %necessary flaws for loading situation.
        %--------------------------------------------------------------
        failure = false;
        if (strcmp(loadSit,'ST'))
            if (sP ==1)
                qList = [40000];
            else
                qList = [qList(1)*1.3];
            end
        elseif (strcmp(loadSit,'SSP')|| (strcmp(loadSit,'CP')))
            if (sP == 1)
                qList = [100];
            else
                qList = [qList(1)*1.3];
            end

        end
        flawsToBeExamined = [1:size(flawsInSP,1)].';
        if (strcmp(loadSit,'ST')|| (strcmp(loadSit,'CP')))
            flawsToBeExamined2 = [1:size(flawsInSP2,1)].';
        else
            flawsToBeExamined2 = [];
        end
        %-------------------------------------------------------------
        %Finding the first failing flaw.
        %-------------------------------------------------------------
        while (failure == false)
            qn = qList(1);
            failedFlawCount = 0;
            if (strcmp(loadSit,'SSP'))
                wn = zeros(1,totNumSeries);
                counting = 1;
                for n = 1:numNSeries
                    for m = 1:numMSeries
                        if (mod(m,2)~=0 && mod(n,2)~=0)
                            wn(counting) = (16*qn)/(D*m*n*(pi^6)*((m/(Length))^2 + (n/(width))^2)
                                ↪ ^2);
                        end
                        counting = counting + 1;
                    end
                end
            elseif (strcmp(loadSit,'CP'))
                bload(1:(numMSeries*numNSeries)) = ((qn*Length^4)/(4*(pi^4)*D));
                wn = (Kmatrix\bload.').';
            end

            if (~isempty(flawsToBeExamined))
```

```
            for index = 1:size(flawsToBeExamined,1)
                flaw = flawsToBeExamined(index);
                if (strcmp(loadSit,'SSP')|| (strcmp(loadSit,'CP')))
                    sigmaXseries = sigmaXXm{flaw};
                    sigmaYseries = sigmaYYm{flaw};
                    tauXYseries = tauXYm{flaw};

                    sigmax = wn*sigmaXseries.';
                    sigmay = wn*sigmaYseries.';
                    tauxy = wn*tauXYseries.';

                    stresses = [sigmax sigmay tauxy];
                elseif (strcmp(loadSit,'ST'))
                    stresses = [qn qn*factor 0];
                end
                %------------------------------------------------------------
                %If flaw fails for this load, save this in array.
                %------------------------------------------------------------
                [failbool, stress] = willFlawPropagate(stresses, flawsInSP(flaw,2), flawsInSP(
                    ↪ flaw,1), geoFactor, Kcrit);
                if (failbool == true)
                    failedFlawCount = failedFlawCount + 1;
                    failedFlawInfo = [failedFlawInfo;[flaw xPositions(flaw) yPositions(flaw)
                        ↪ stress flawsInSP(flaw,1) flawsInSP(flaw,2) 0]];
                end
            end
        end
        if (~isempty(flawsToBeExamined2))
            for index = 1:size(flawsToBeExamined2,1)

                flaw = flawsToBeExamined2(index);

                if (strcmp(loadSit,'SSP') || (strcmp(loadSit,'CP')))
                    sigmaXseries = sigmaXXm{flaw};
                    sigmaYseries = sigmaYYm{flaw};
                    tauXYseries = tauXYm{flaw};

                    sigmax = -wn*sigmaXseries.';
                    sigmay = -wn*sigmaYseries.';
                    tauxy = -wn*tauXYseries.';

                    stresses = [sigmax sigmay tauxy];
                elseif (strcmp(loadSit,'ST'))
                    stresses = [qn qn*factor 0];
                end
                %-------------------------------------------------------
                %If flaw fails for this load, save this in array.
                %-------------------------------------------------------
                [failbool, stress] = willFlawPropagate(stresses, flawsInSP2(flaw,2), flawsInSP2
                    ↪ (flaw,1), geoFactor, Kcrit);
                if (failbool == true)
                    failedFlawCount = failedFlawCount + 1;
                    failedFlawInfo2 = [failedFlawInfo2;[flaw xPositions(flaw) yPositions(flaw)
                        ↪ stress flawsInSP2(flaw,1) flawsInSP2(flaw,2) 1]];
                end
            end
        end
        %-------------------------------------------------------
        %If there is only one flaw failing, converge
        %until its critical load is found. Find deflection
        %in that process.
        %-------------------------------------------------------
        registeredChange = 0;
        if (failedFlawCount == 1)
            if (isempty(failedFlawInfo))
               failedFlawInfo = failedFlawInfo2;
               registeredChange = 1;
            end
            failure = true;
            criticalStress = (Kcrit)/(geoFactor*sqrt(pi*failedFlawInfo(5))); %Pa
            qList = findCriticalPressureAlgorithm(qList,1);
            convergence = false;
            while (~convergence)
                disp(stress);
                wn = zeros(1,totNumSeries);
                counting = 1;
                deflection = 0;
                if (strcmp(loadSit,'SSP')|| (strcmp(loadSit,'CP')))
                    if (strcmp(loadSit, 'SSP'))
                        for n = 1:numNSeries
                            for m = 1:numMSeries
                                if (mod(m,2)~=0 && mod(n,2)~=0)
                                    wn(counting) = (16*qList(1))/(D*m*n*(pi^6)*((m/(Length))^2
                                        ↪ + (n/(width))^2)^2);
                                    deflection = deflection + (wn(counting)*sin((m*pi)/2)*sin((
                                        ↪ n*pi)/2));
                                end
                                counting = counting + 1;
                            end
                        end
                    else
                        bload(1:(numMSeries*numNSeries)) = ((qList(1)*Length^4)/(4*(pi^4)*D));
                        wn = (Kmatrix\bload.').';
                        index = 1;
                        for m = 1:numNSeries
                            for n = 1:numMSeries
                                deflection = deflection + wn(index)*(1-cos(m*pi))*(1-cos(n*pi))
                                    ↪ ; %ikke 100% sikker p? komposisjonen av wn og om dette
                                    ↪ da stemmer!!
                                index = index + 1;
                            end
                        end
                    end
                    sigmaXseries = sigmaXXm{failedFlawInfo(1)};
                    sigmaYseries = sigmaYYm{failedFlawInfo(1)};
                    tauXYseries = tauXYm{failedFlawInfo(1)};

                    if (registeredChange == 1 && strcmp(loadSit, 'CP'))
                        wn = -wn;
                    end
                    stresses = zeros(1,3);
                    stresses(1) = wn*sigmaXseries.';
                    stresses(2)= wn*sigmaYseries.';
                    stresses(3)= wn*tauXYseries.';
                    %----------------------------------------------------------
                    %Get center stresses for output.
                    %----------------------------------------------------------
```

```
                if (mod(numFlawsSP/2,Length*100)==0)
                    midsigmax = wn*sigmaXXm{(numFlawsSP/2)+ (Length*100*0.5)}.';
                    midsigmay = wn*sigmaYYm{(numFlawsSP/2)+ (Length*100*0.5)}.';
                    midtauxy = wn*tauXYm{(numFlawsSP/2)+ (Length*100*0.5)}.';
                else
                    midsigmax = wn*sigmaXXm{numFlawsSP/2}.';
                    midsigmay = wn*sigmaYYm{numFlawsSP/2}.';
                    midtauxy = wn*tauXYm{numFlawsSP/2}.';
                end
            elseif (strcmp(loadSit,'ST'))
                stresses = [qList(1) qList(1)*factor 0];
            end
            [failbool, stress] = willFlawPropagate(stresses, failedFlawInfo(6),
                ↪ failedFlawInfo(5), geoFactor, Kcrit);
            if (abs(stress - criticalStress)/criticalStress <= 0.0002)
                forMonteCarlo = [forMonteCarlo;[failedFlawInfo(2) failedFlawInfo(3) stress
                    ↪ qList(1) deflection ]]; %qList(1) her er feil!!!
                check = [check; sP failedFlawInfo(1) failedFlawInfo(6) failedFlawInfo(5)
                    ↪ failedFlawInfo(7) stresses(1) stresses(2) stresses(3)]
                convergence = true;
            end
            qList = findCriticalPressureAlgorithm(qList,failbool);
        end
    %-------------------------------------------------------------
    %More than one failure for the current plate. Register these
    %failed flaws, and eliminate others as first failing flaw
    %candidate. Reduce load. Empty failed flaws info array.
    %-------------------------------------------------------------
    elseif (failedFlawCount > 1)
        qList = findCriticalPressureAlgorithm(qList,1);
        if (~isempty(failedFlawInfo))
            flawsToBeExamined = failedFlawInfo(:,1);
        else
            flawsToBeExamined = [];
        end
        if (strcmp(loadSit,'ST')|| (strcmp(loadSit,'CP')))
            if (~isempty(failedFlawInfo2))
                flawsToBeExamined2 = failedFlawInfo2(:,1);
            else
```

```
                        flawsToBeExamined2 = [];
                    end
                end
            else
                qList = findCriticalPressureAlgorithm(qList,0);
            end
            failedFlawInfo = [];
            if (strcmp(loadSit,'ST')|| (strcmp(loadSit,'CP')))
               failedFlawInfo2 = [];
            end
        end
    end
end
%--------------------------------------------------------------------
%Save information of the first failing flaws of the plates in a
%text-file for post-processing.
%--------------------------------------------------------------------
file = fopen('matlabvalues.txt','w');
fprintf(file, 'Deflection[mm],␣Load[MPa],␣x[cm],␣y[cm]');
for i=1:size(forMonteCarlo,1)
    fprintf(file,'%1.9f␣%1.5f␣%1.5f␣%1.5f␣\r\n',forMonteCarlo(i,end)*10^3,forMonteCarlo(i,4)*10^-6,
        ↪  forMonteCarlo(i,1), forMonteCarlo(i,2));
end
fclose(file);

disp('Mean␣critical␣load');
disp(sum(forMonteCarlo(:,4)*10^-6)/size(forMonteCarlo(:,4),1));
```

Appendix B

Python source code

```
def file_len(fname):
        """Calculates the length of a .txt-file"""
        with open(fname) as f:
                for i,l in enumerate(f):
                        pass
                return i+1

def findFailStress(xxfail, yyfail, xyfail, xxnofail, yynofail, xynofail):
        """Linearizes stresses and returns the middle value. """
        factor = 0.5
        xx = xxnofail + factor*(xxfail-xxnofail)
        yy = yynofail + factor*(yyfail-yynofail)
        xy = xynofail + factor*(xyfail-xynofail)
        return [xx, yy, xy]

def unique_rows(a):
        """Returns unique rows of a numpy array."""
        a = np.ascontiguousarray(a)
        unique_a = np.unique(a.view([('',a.dtype)]*a.shape[1]))
        return unique_a.view(a.dtype).reshape((unique_a.shape[0], a.shape[1]))

def column(matrix, i):
        """Given an index, returns the corresponding column in a matrix."""
    return [row[i] for row in matrix]
```

```
def binary_search(array, target):
        """Searches for a target value in an index by binary search. Returns the index."""
    lower = 0
    upper = len(array)
    while lower < upper:
        x = lower + (upper − lower) // 2
        val = array[x]
        if target == val:
            return x
        elif target > val:
            if lower == x:
                break
            lower = x
        elif target < val:
            upper = x

def back_search(failuresAtThisFrame, plate):
        """Returns a list of the indexes of the target value plate in the list
        failuresAtThisFrame. Starts at the last index which needs to be a plate−value."""
        arr = column(failuresAtThisFrame,0)
        index = −1
        stopIndex = len(arr)
        indexlist = np.array([index])
        iteration = 0
        while(abs(index)!=stopIndex):
                index = index − 1
                if (int(arr[index])==plate):
                        indexlist = np.vstack((indexlist, [index]))
                else:
                        index = stopIndex
        return indexlist

def width_search(failuresAtLastFrame, plate):
        """Returns a list of all indexes of target value plate in failuresAtLastFrame."""
        arr = column(failuresAtLastFrame,0)
        startIndex = binary_search(arr,plate)
        if (startIndex == None):
                print 'There␣are␣no␣failing␣flaws␣for␣this␣plate'
                quit()
        plusIndex = startIndex + 1
        minusIndex = startIndex - 1
        plusOK = False
        minusOK = False

        indexlist1 = np.array([])
        indexlist2 = np.array([startIndex])
        while(plusOK == False):
                if (plusIndex < len(arr)):
                        if (int(arr[plusIndex]) == plate):
                            indexlist2 = np.vstack((indexlist2, [plusIndex]))
                            plusIndex = plusIndex + 1
                        else:
                            plusOK = True
                else:
                        plusOK = True
        while(minusOK == False):
                if (minusIndex >= 0):
                        if (int(arr[minusIndex]) == plate):
                            if (indexlist1.size == 0):
                                indexlist1 = np.array([minusIndex])
                            else:
                                indexlist1 = np.vstack((indexlist1, [minusIndex]))
                            minusIndex = minusIndex - 1
                        else:
                            if (indexlist1.size != 0):
                                indexlist1 = np.fliplr([indexlist1])[0]
                                indexlist1 = np.vstack((indexlist1,indexlist2))
                            else:
                                indexlist1 = indexlist2
                            minusOK = True
                else:
                        if (indexlist1.size != 0):
                            indexlist1 = np.fliplr([indexlist1])[0]
                            indexlist1 = np.vstack((indexlist1,indexlist2))
                        else:
                            indexlist1 = indexlist2
                        minusOK = True
        return indexlist1
```

```
from odbAccess import *
from math import *
import numpy as np
#=========================================================================
##########                    DEFINE NECESSARY INITIAL CONDITIONS
#=========================================================================
#-------------------------------------------------------------------------
#Fine mesh (allows mesh of 2 square millimeters)
#-------------------------------------------------------------------------
finemesh = False
#-------------------------------------------------------------------------
#Forward search appropriate for some nonlinear cases. stopAtTimeFrame = -1 if all time
#frames are to be checked.
#-------------------------------------------------------------------------
nonlinear = True
stopTimeFrame = -1
#-------------------------------------------------------------------------
#Hardcoded values for mode I failure for glass
#-------------------------------------------------------------------------
Kcrit = 0.75*pow(10,6) #0.75*pow(10,6)NM^-3/2, men m  bli  Nmm^-3/2
Y = 0.6625
#-------------------------------------------------------------------------
#Access Abaqus job and create a variable referring to the last time step for Step-1
#-------------------------------------------------------------------------
odbfile = 'Plate.odb'
odb = openOdb(odbfile,readOnly=True)
instance = odb.rootAssembly.instances['PLATE-1']
#-------------------------------------------------------------------------
#Initialize plate geometry and element list
#Make sure the length is initialized as x-axis length and width as y-axis width
#-------------------------------------------------------------------------
numElementsInFile = len(instance.elements)
lengthCM = 35
widthCM = 35
numElements = lengthCM*widthCM
if (finemesh == False):
        if (numElementsInFile != numElements):
                print 'Element␣size␣not␣of␣1␣square␣centimeter'
                odb.close()
        relevantElements = np.arange(numElements)
else:
        numElementsInMesh = numElements*25
        if (numElementsInFile != numElementsInMesh):
                print 'Element␣size␣not␣of␣2␣square␣millimeters'
                odb.close()
        el_length = lengthCM*5
        el_width = widthCM*5
        cm_el_length = lengthCM;
        cm_el_width = widthCM;
        relevantElements = np.zeros(numElements)
        count = 0
        for elL in range(1,cm_el_width+1):
                if (elL == 1):
                        index = el_length*2 -2
                else:
                        index = index + el_length*4
                for elB in range(1,cm_el_length+1):
                        index = index + 5
                        relevantElements[count] = index
                        count = count + 1
#-----------------------------------------------------------------------------------------
#Open textfile with flaws. Register number of plates in text file.
#-----------------------------------------------------------------------------------------
file = open('flawInfoForPlate.txt','r').readlines()
file2 = open('flawInfoForPlate-2.txt','r').readlines()
numLines = file_len('flawInfoForPlate.txt')
numPlatesNext = np.arange(numLines/numElements);
open('flawInfoForPlate.txt','r').close()
open('flawInfoForPlate-2.txt','r').close()

#===========================================================================
############################################################################
#===========================================================================


#-----------------------------------------------------------------------------------------
#Initializing lists carrying the failures registered.
```

```
#------------------------------------------------------------------------------
failuresAtThisFrame = np.array([])
failuresAtLastFrame = np.array([])
finalFailures = np.array([])
areAllPlatesFailing = np.zeros(len(numPlatesNext))
#------------------------------------------------------------------------------
#Initializing
#------------------------------------------------------------------------------
if (nonlinear == True):
        frameIndex = -1
else:
        frameIndex = 0
allPlatesHaveFirstFailingFlaws = 0
sigmaXX1 = np.zeros(numElements)
sigmaYY1 = np.zeros(numElements)
tauXY1 = np.zeros(numElements)
sigmaXX2 = np.zeros(numElements)
sigmaYY2 = np.zeros(numElements)
tauXY2 = np.zeros(numElements)
#------------------------------------------------------------------------------
#Finding the first failings flaws. This will loop through the time steps where field
#data is generated and retrieve the stresses. There might be more than one failing
#flaw per plate after this while loop has continued running.
#------------------------------------------------------------------------------
while (allPlatesHaveFirstFailingFlaws == 0):

        if(nonlinear ==True):
                frameIndex = frameIndex + 1
        else:
                frameIndex = frameIndex - 1
        print frameIndex
        #----------------------------------------------------------------------
        # Stopping the time step search at last time step
        #----------------------------------------------------------------------
        if ((odb.steps['Step-1'].frames[frameIndex] == odb.steps['Step-1'].frames[0] and nonlinear
            ↪ ==False) or (odb.steps['Step-1'].frames[frameIndex] == odb.steps['Step-1'].frames[
            ↪ stopTimeFrame] and nonlinear==False)):
                print 'Final␣time␣step␣is␣reached␣and␣this␣is␣the␣very␣last␣frame␣loop'
                allPlatesHaveFirstFailingFlaws = 1
```

```
lastFrame = odb.steps['Step-1'].frames[frameIndex]
stresses = lastFrame.fieldOutputs['S']
#------------------------------------------------------------------------------
#Retrieving stresses for different cases.
#------------------------------------------------------------------------------
if (relevantElements.shape != (2L,)):
        for elem in relevantElements:
                if ((frameIndex == -1 and nonlinear==False) or (nonlinear==True and
                    ↪ frameIndex == 0)):
                        el = int(elem)
                        thiselement = stresses.getSubset(region = instance.elements[el],
                            ↪ position=CENTROID)
                        sigmaXX1[el] = thiselement.values[0].data[0]*pow(10,6)
                        sigmaYY1[el] = thiselement.values[0].data[1]*pow(10,6)
                        tauXY1[el] = thiselement.values[0].data[3]*pow(10,6)

                        sigmaXX2[el] = thiselement.values[1].data[0]*pow(10,6)
                        sigmaYY2[el] = thiselement.values[1].data[1]*pow(10,6)
                        tauXY2[el] = thiselement.values[1].data[3]*pow(10,6)
                elif(odb.steps['Step-1'].frames[frameIndex] == odb.steps['Step-1'].frames
                    ↪ [0]):
                        el = int(elem[0])
                        if (elem[1]==0):
                                sigmaXX1[el] = 0
                                sigmaYY1[el] = 0
                                tauXY1[el] = 0
                        else:
                                sigmaXX2[el] = 0
                                sigmaYY2[el] = 0
                                tauXY2[el] = 0
                else:
                        el = int(elem[0])
                        thiselement = stresses.getSubset(region = instance.elements[el],
                            ↪ position=CENTROID)
                        if (elem[1]==0):
                                sigmaXX1[el] = thiselement.values[0].data[0]*pow(10,6)
                                sigmaYY1[el] = thiselement.values[0].data[1]*pow(10,6)
                                tauXY1[el] = thiselement.values[0].data[3]*pow(10,6)
```

```
            else:
                sigmaXX2[el] = thiselement.values[1].data[0]*pow(10,6)
                sigmaYY2[el] = thiselement.values[1].data[1]*pow(10,6)
                tauXY2[el] = thiselement.values[1].data[3]*pow(10,6)
else:
    thiselement = stresses.getSubset(region = instance.elements[relevantElements[0]],
      ↪ position=CENTROID)
    if (relevantElements[1]==0):
        sigmaXX1[relevantElements[0]] = thiselement.values[0].data[0]*pow(10,6)
        sigmaYY1[relevantElements[0]] = thiselement.values[0].data[1]*pow(10,6)
        tauXY1[relevantElements[0]] = thiselement.values[0].data[3]*pow(10,6)
    else:
        sigmaXX2[relevantElements[0]] = thiselement.values[1].data[0]*pow(10,6)
        sigmaYY2[relevantElements[0]] = thiselement.values[1].data[1]*pow(10,6)
        tauXY2[relevantElements[0]] = thiselement.values[1].data[3]*pow(10,6)
#-------------------------------------------------------------------------------------
#For every element/flaw, check relevant flaws in all plates if they fail
#for the stresses
#-------------------------------------------------------------------------------------
numPlates = numPlatesNext
for plate in numPlates:
    failedflawcount = 0
    #---------------------------------------------------------------------------------
    # For the first run for a time frame that is not the first, store the flaws
    #failing for previous time step and initialize the list for retrieving failing
    #flaws at this time step.
    #---------------------------------------------------------------------------------
    if ((frameIndex!=-1 and plate==numPlates[0] and nonlinear==False) or (frameIndex!=0
      ↪ and plate==numPlates[0] and nonlinear==True)):
        if (failuresAtThisFrame.size != 0):
            failuresAtLastFrame = failuresAtThisFrame
            failuresAtThisFrame = np.array([])
    #---------------------------------------------------------------------------------
    #Checking first failing flaw candidates for failure
    #---------------------------------------------------------------------------------
    for elem in relevantElements:
        if (relevantElements.shape != (2L,)):
            if((frameIndex != -1 and nonlinear==False) or (frameIndex!=0 and
              ↪ nonlinear==True)):
                flaw = int(elem[0])
                pos = int(elem[1])
            else:
                pos = 5 #tweak
                flaw = int(elem)
        else:
            flaw = int(relevantElements[0])
            pos = int(relevantElements[1])

        thisFlaw = file[int((plate*numElements)+(flaw))].split()
        thisFlawLength = float(thisFlaw[0])
        thisFlawAngle = float(thisFlaw[1])

        thisFlaw2 = file2[int((plate*numElements)+(flaw))].split()
        thisFlawLength2 = float(thisFlaw2[0])
        thisFlawAngle2 = float(thisFlaw2[1])
        if ((frameIndex == -1 and nonlinear==False)or(frameIndex==0 and nonlinear==
            ↪ True)):
            sigma1 = sigmaXX1[flaw] + sigmaYY1[flaw] - (0.5*(sigmaXX1[flaw]+
              ↪ sigmaYY1[flaw]) + 0.5*(sigmaXX1[flaw]-sigmaYY1[flaw])*cos(2*
              ↪ thisFlawAngle)+(tauXY1[flaw]*sin(2*thisFlawAngle)))
            sigma2 = sigmaXX2[flaw] + sigmaYY2[flaw] - (0.5*(sigmaXX2[flaw]+
              ↪ sigmaYY2[flaw]) + 0.5*(sigmaXX2[flaw]-sigmaYY2[flaw])*cos(2*
              ↪ thisFlawAngle2)+(tauXY2[flaw]*sin(2*thisFlawAngle2)))
            sigmalist = np.array([sigma1, sigma2])
        else:
            if (pos == 0):
                sigmalist = np.array([sigmaXX1[flaw] + sigmaYY1[flaw] -
                ↪ (0.5*(sigmaXX1[flaw]+sigmaYY1[flaw]) + 0.5*(sigmaXX1
                ↪ [flaw]-sigmaYY1[flaw])*cos(2*thisFlawAngle)+(tauXY1[
                ↪ flaw]*sin(2*thisFlawAngle)))])
            else:
                sigmalist = np.array([sigmaXX2[flaw] + sigmaYY2[flaw] -
                ↪ (0.5*(sigmaXX2[flaw]+sigmaYY2[flaw]) + 0.5*(sigmaXX2
                ↪ [flaw]-sigmaYY2[flaw])*cos(2*thisFlawAngle2)+(tauXY2
                ↪ [flaw]*sin(2*thisFlawAngle2)))])
        sigmacount = 0
        #----------------------------------------------------------------------
        #Calculating toughness
```

```
#---------------------------------------------------------------------------
for sigma in sigmalist:
        sigmacount = sigmacount + 1
        if (sigma >= 0):
                if ((len(sigmalist)==2 and sigmacount==1) or (pos==0)):
                        K = Y*sigma*sqrt(pi*thisFlawLength)
                else:
                        K = Y*sigma*sqrt(pi*thisFlawLength2)
        else:
                K = 0
        #-----------------------------------------------------
        #Checking for failure and in that case saving
        #information about the failing flaw
        #-----------------------------------------------------
        if (Kcrit <= K):
                if (frameIndex == -1 and nonlinear==False):
                        if (int(areAllPlatesFailing[plate]) == 0):
                                areAllPlatesFailing[plate] = 1


                        if (failuresAtThisFrame.size == 0):
                                if (sigmacount==1):
                                        failuresAtThisFrame = np.array([
                                            ↪ plate, flaw, sigma,
                                            ↪ frameIndex, 0])
                                else:
                                        failuresAtThisFrame = np.array([
                                            ↪ plate, flaw, sigma,
                                            ↪ frameIndex, 1])
                        else:
                                if (sigmacount==1):
                                        failuresAtThisFrame = np.vstack((
                                            ↪ failuresAtThisFrame, [plate,
                                            ↪  flaw, sigma, frameIndex,
                                            ↪ 0]))
                                else:
                                        failuresAtThisFrame = np.vstack((
                                            ↪ failuresAtThisFrame, [plate,
                                            ↪  flaw, sigma, frameIndex,
                                            ↪ 1]))
                              else:
                                    if (failuresAtThisFrame.size == 0):
                                          failuresAtThisFrame = np.array([plate, flaw
                                              ↪ , sigma, frameIndex, pos])
                                    else:
                                          failuresAtThisFrame = np.vstack((
                                              ↪ failuresAtThisFrame, [plate, flaw,
                                              ↪ sigma, frameIndex, pos]))
                              failedflawcount = failedflawcount + 1
            #-------------------------------------------------------------------------
            # Linear: If no flaws failed in this plate at this time frame, the list of
            #failed flaws from the last time frame is registered as the final failing flaws.
            #The plate is removed in order to not be included at next time frame.
            #-------------------------------------------------------------------------
            if (int(failedflawcount) == 0 and nonlinear==False):
                  if (failuresAtLastFrame.shape != (5L,)):
                        indexes = width_search(failuresAtLastFrame, int(plate))
                        for index in indexes:
                              if (finalFailures.size == 0):
                                    finalFailures = np.array(failuresAtLastFrame[index
                                        ↪ ])
                              else:
                                    finalFailures = np.vstack((finalFailures,
                                        ↪ failuresAtLastFrame[index]))
                  else:
                        if (int(failuresAtLastFrame[0])==int(plate)):
                              if (finalFailures.size == 0):
                                    finalFailures = failuresAtLastFrame
                              else:
                                    finalFailures = np.vstack((finalFailures,
                                        ↪ failuresAtLastFrame))

                  numPlatesNext = np.delete(numPlatesNext,np.where(numPlatesNext==plate)[0])
            #-------------------------------------------------------------------------
            #Nonlinear: If failure occurs at this time frame for a plate, they are stored
            #in finalFailures. The plate is then removed from numPlatesNext not to be
            #included in a new loop.
            #-------------------------------------------------------------------------
```

```
        if (int(failedflawcount)!=0 and nonlinear == True):
            if(failuresAtThisFrame.shape != (5L,)):
                indexes = back_search(failuresAtThisFrame, int(plate))
                for index in indexes:
                    if (finalFailures.size == 0):
                        finalFailures = np.array(failuresAtThisFrame[index
                          ↪ ])
                    else:
                        finalFailures = np.vstack((finalFailures,
                          ↪ failuresAtThisFrame[index]))
            else:
                if (int(failuresAtThisFrame[0])==int(plate)):
                    if (finalFailures.size == 0):
                        finalFailures = failuresAtThisFrame
                    else:
                        finalFailures = np.vstack((finalFailures,
                          ↪ failuresAtThisFrame))

            numPlatesNext = np.delete(numPlatesNext,np.where(numPlatesNext==plate)[0])
    #-----------------------------------------------------------------------------
    # Validates that all plates are failing at first time frame for linear cases. If no
    #failure occurs at first time frame, it is unlikely failure will occur at a later
    #stage. This only suits the linear case and for nonlinear cases should be removed.
    #-----------------------------------------------------------------------------
    if(int(frameIndex)==-1 and nonlinear==False and int(np.amin(areAllPlatesFailing)==0)):
        print 'Not␣all␣glass␣plates␣failed␣for␣the␣greatest␣stresses.␣The␣analysis␣will␣be␣
          ↪ incomplete.␣Apply␣greater␣stresses␣in␣the␣Abaqus␣job.'
        quit()
    #-----------------------------------------------------------------------------
    # Updating the list of first failure candidates as to which have failed and not.
    #-----------------------------------------------------------------------------
    if(nonlinear==False):
        if (failuresAtThisFrame.size == 0):
            allPlatesHaveFirstFailingFlaws = 1
        else:
            if (failuresAtThisFrame.shape != (5L,) ):
                relevantElements = np.array([])
                for fails in failuresAtThisFrame:
                    if ((fails == failuresAtThisFrame[0]).all()):
                        relevantElements = np.array([int(fails[1]),int(
                            ↪ fails[4])])
                    else:
                        relevantElements = np.vstack((relevantElements, [
                            ↪ int(fails[1]),int(fails[4])]))
                relElemCheck = relevantElements
                relevantElements = unique_rows(relevantElements)
            else:
                relevantElements = np.array([int(failuresAtThisFrame[1]),int(
                    ↪ failuresAtThisFrame[4])])
    else:
        if(len(numPlatesNext)==0):
            allPlatesHaveFirstFailingFlaws = 1
        else:
            if(frameIndex==0):
                for elements in relevantElements:
                    if(elements==relevantElements[0]):
                        relevantElements2 = np.array([elements, 0])
                        relevantElements2 = np.vstack((relevantElements2, [
                            ↪ elements,1]))
                    else:
                        relevantElements2 = np.vstack((relevantElements2, [
                            ↪ elements,0]))
                        relevantElements2 = np.vstack((relevantElements2, [
                            ↪ elements,1]))
            relevantElements = relevantElements2

#------------------------------------------------------------------------------------------
# Converging the failed flaws of the plates by finding the critical stresses. Retrieving
#deflection from output database. Can retrieve center pressure where applicable, but
#needs uncommenting.
#------------------------------------------------------------------------------------------
finalFailuresLinearized = np.array([])

numPlates = np.arange(numLines/numElements)
#Allowing for only a few plates failing in forward search of time frames
if (len(numPlatesNext) != 0 and nonlinear==True):
    for plate in numPlatesNext:
        numPlates = np.delete(numPlates,np.where(numPlates==plate)[0])
```

```
for plate in numPlates:
        plateReached = False
        thisPlateLinearized = np.array([])
        stressList = np.array([])
        for row in finalFailures:
                #assuming more than one plate in the analysis
                if (row[0] == plate):
                        el = int(row[1])
                        pos = int(row[4])
                        frameFail = odb.steps['Step-1'].frames[int(row[3])]
                        deflections = frameFail.fieldOutputs['U']
                        node = int(ceil(((sqrt(numElements)+1)*(sqrt(numElements)+1))/2))
                        centerDeflection = deflections.getSubset().values[node].data[2]

                        #pressureFail = frameFail.fieldOutputs['P'].getSubset().values[node].data
                        pressureFail = 'Not␣included␣in␣analysis'

                        frameNoFail = odb.steps['Step-1'].frames[int(row[3])-1]
                        deflectionsNoFail = frameNoFail.fieldOutputs['U']
                        centerDeflectionNoFail = deflectionsNoFail.getSubset().values[node].data[2]

                        #pressureNoFail = frameNoFail.fieldOutputs['P'].getSubset().values[node].
                            ↪ data
                        pressureNoFail = 'Not␣included␣in␣analysis'

                        stresses = frameFail.fieldOutputs['S']
                        thiselement = stresses.getSubset(region = instance.elements[el],position=
                            ↪ CENTROID)
                        sigmaXXfail = thiselement.values[pos].data[0]*pow(10,6)
                        sigmaYYfail = thiselement.values[pos].data[1]*pow(10,6)
                        tauXYfail = thiselement.values[pos].data[3]*pow(10,6)

                        frameNoFail = odb.steps['Step-1'].frames[int(row[3])-1]
                        stresses = frameNoFail.fieldOutputs['S']
                        thiselement = stresses.getSubset(region = instance.elements[el],position=
                            ↪ CENTROID)
                        sigmaXXnofail = thiselement.values[pos].data[0]*pow(10,6)
                        sigmaYYnofail = thiselement.values[pos].data[1]*pow(10,6)
```

```
            tauXYnofail = thiselement.values[pos].data[3]*pow(10,6)

            convergence = False
            if (row[4] == 0):
                thisFlaw = file[int((plate*numElements)+(el))].split()
                thisFlawLength = float(thisFlaw[0])
                thisFlawAngle = float(thisFlaw[1])
            else:
                thisFlaw = file2[int((plate*numElements)+(el))].split()
                thisFlawLength = float(thisFlaw[0])
                thisFlawAngle = float(thisFlaw[1])
            criticalStress = (Kcrit)/(Y*sqrt(pi*thisFlawLength))

            failcontrol = sigmaXXfail + sigmaYYfail − (0.5*(sigmaXXfail+sigmaYYfail) +
              ↪ 0.5*(sigmaXXfail−sigmaYYfail)*cos(2*thisFlawAngle)+(tauXYfail*sin(2*
              ↪ thisFlawAngle)))
            nofailcontrol = sigmaXXnofail + sigmaYYnofail − (0.5*(sigmaXXnofail+
              ↪ sigmaYYnofail) + 0.5*(sigmaXXnofail−sigmaYYnofail)*cos(2*
              ↪ thisFlawAngle)+(tauXYnofail*sin(2*thisFlawAngle)))

            this = 0
            while (not convergence):
                [newXX, newYY, newXY] = findFailStress(sigmaXXfail, sigmaYYfail,
                  ↪ tauXYfail, sigmaXXnofail, sigmaYYnofail, tauXYnofail)
                sigma = newXX + newYY − (0.5*(newXX+newYY) + 0.5*(newXX−newYY)*cos
                  ↪ (2*thisFlawAngle)+(newXY*sin(2*thisFlawAngle)))
                sigmasqrtA = sigma*sqrt(thisFlawLength)
                K = Y*sigma*sqrt(pi*thisFlawLength)

                if (K >= Kcrit):
                    sigmaXXfail = newXX
                    sigmaYYfail = newYY
                    tauXYfail = newXY
                else:
                    sigmaXXnofail = newXX
                    sigmaYYnofail = newYY
                    tauXYnofail = newXY
                if (this == abs(sigma−criticalStress)/criticalStress):
                    print 'Will␣not␣converge␣properly'
```

```
                            quit()
                        this = abs(sigma−criticalStress)/criticalStress

                        if (abs(sigma−criticalStress)/criticalStress <= 0.0002):
                            convergence = True

                #—————————————————————————————————————————————————————————————————————
                #Saving information of converged flaw. Breaking the loop of all failures in
                #all plates if all failures in current plate is accounted for, in order to
                #move on to next plate.
                #—————————————————————————————————————————————————————————————————————
                if (plateReached == False):
                        stressList = np.array([plate, el, sigmaXXfail, sigmaYYfail,
                            ↪ tauXYfail, failcontrol, nofailcontrol, sigma, row[4], row
                            ↪ [3], centerDeflection, centerDeflectionNoFail, pressureFail,
                            ↪  pressureNoFail])
                else:
                        stressList = np.vstack((stressList, [plate, el, sigmaXXfail,
                            ↪ sigmaYYfail, tauXYfail, failcontrol, nofailcontrol, sigma,
                            ↪ row[4], row[3], centerDeflection, centerDeflectionNoFail,
                            ↪ pressureFail, pressureNoFail]))
                plateReached = True
            elif ((plateReached == True) and (int(row[0]) is not plate)):
                break


        #—————————————————————————————————————————————————————————————————————
        #Finding which of the converged failed flaws of the current plate is failing first
        #by comparing by percentage which of the flaws critical stresses is the closest to
        #the stresses of the no fail time frame. If pressure is to be included in the
        #analysis, this needs uncommenting.
        #—————————————————————————————————————————————————————————————————————
        lastpercentage = 1
        if (stressList.shape == (14L,)):
            firstFailingFlaw = stressList
            lastpercentage = (firstFailingFlaw[7]−firstFailingFlaw[6])/(firstFailingFlaw[5]−
                ↪ firstFailingFlaw[6])
            actualCenterDeflection = flaw[11] + (flaw[10]−flaw[11])*lastpercentage
            #actualPressureFail = flaw[13] + (flaw[12] − flaw[13])*lastpercentage
```

```
                actualPressureFail = 'Not␣included␣in␣analysis'
        else:
                for flaw in stressList:
                        percentage = (flaw[7]-flaw[6])/(flaw[5]-flaw[6])
                        if (percentage <= lastpercentage):
                                firstFailingFlaw = flaw
                                lastpercentage = percentage
                                actualCenterDeflection = flaw[11] + (flaw[10]-flaw[11])*
                                    ↪ lastpercentage
                                #actualPressureFail = flaw[13] + (flaw[12] - flaw[13])*
                                    ↪ lastpercentage
                                actualPressureFail = 'Not␣included␣in␣analysis'
        #if (plate == 0 or (nonlinear==True and plate == numPlates[0])):
        if(plate==numPlates[0]):
                finalFailuresLinearized = np.array(['Plate␣no.', 'Flaw␣no.', 'sigma_xx', 'sigma_yy'
                    ↪ , 'tau_xy', 'controlparameter_fail', 'controlparameter_nofail', 'tensile␣
                    ↪ stress', 'side␣of␣plate', 'time␣frame␣index', 'deflection_frame_fail', '
                    ↪ deflection_frame_nofail', 'pressure_frame_fail', 'pressure_frame_nofail', '
                    ↪ closenessToNoFailFramePercentage','criticalCenterDeflection', '
                    ↪ criticalCenterPressure'])
                finalFailuresLinearized  = np.vstack((finalFailuresLinearized, np.append(
                    ↪ firstFailingFlaw, [lastpercentage, actualCenterDeflection,
                    ↪ actualPressureFail])))
        else:
                finalFailuresLinearized  = np.vstack((finalFailuresLinearized, np.append(
                    ↪ firstFailingFlaw, [lastpercentage, actualCenterDeflection,
                    ↪ actualPressureFail])))



#-------------------------------------------------------------------------------
#Write results to a text file for post-processing.
#-------------------------------------------------------------------------------
writefile = open('monteCarloResults.txt','w')
for row in finalFailuresLinearized:
        writefile.write('␣'.join(map(str, row)))
        writefile.write('\n')
writefile.close()
#-------------------------------------------------------------------------------
```

```
#Close .odb-file.
#----------------------------------------------------------------------------------------------
odb.close()
```

```
import matplotlib.pyplot as plt
import matplotlib.mlab as mlab
import numpy as np
from math import *
from scipy import stats
import scipy.stats as ss

#EXAMPLE OF MONTE CARLO POST PROCESSING
#This is the Monte Carlo script for the small sample in the four point bending tests in Section
    ↪ 6.1.2

load = 4000
length =(10) #cm
width = 2
file = open('MonteCarloResults.txt', 'r').readlines()
sigma = np.zeros(len(file))
critload = np.zeros(len(file))
d = np.zeros(len(file))
x = np.zeros(len(file))
y = np.zeros(len(file))
num = np.zeros(len(file))
index = 0
p = 0
v = np.zeros(len(file))

for row in file:
        values = row.split()
        d[index] = float(values[-1])
        sigma[index] = float(values[7])*pow(10,-6)
        percentage = float(values[-2])
        p = p + percentage
        critload[index] = (load)*percentage
        v[index] = values[1]
        xindicator = (float(values[1])+1)%(length*5)
        if (xindicator == 0):
                xindicator = length*5
        xindicator = xindicator/5

        num[index] = values[1]
        yindicator = ceil((float(values[1])+1)/(length*5*5))
        x[index] = xindicator - 0.3
        y[index] = yindicator - 0.5 #tweak

        index = index + 1
plt.show()
nbins = 30
ss = np.unique(v)
open('monteCarloResults.txt', 'r').close()


actual = open('smallTestResults.txt','r').readlines()
defl = np.zeros(len(actual))
F = np.zeros(len(actual))
index = 0
for line in actual:
        values = line.split()
        F[index] = values[0]
        defl[index] = values[1]
        index = index + 1

open('smallTestResults.txt','r').close()
critload = np.sort(critload)
F = np.sort(F)
defl = np.sort(defl)
d = np.sort(d)

meanF = sum(F)/len(F)
meanDefl = sum(defl)/len(defl)
sigmF = np.std(F)
sigmDefl = np.std(defl)



XX = np.linspace(0,3000,200)
```

```
mean1 = sum(critload)/len(critload)
meand = sum(d)/len(d)
sigmd = np.std(d)
sigm1 = np.std(critload)
bins = 30
plt.axis([0,10,0,2])
plt.xlabel('Distance␣along␣the␣plate␣length␣[cm]', fontsize = 20)
plt.ylabel('Distance␣along␣the␣plate␣width␣[cm]',fontsize = 20)
plt.scatter(x,y, label = 'Analysis␣results')
plt.show()



#CUMULATIVE NORMAL
plt.plot(XX, stats.norm.cdf(XX,loc=meanF, scale=sigmF), 'k--')
plt.plot(XX, stats.norm.cdf(XX,loc=mean1, scale=sigm1), 'k--')

Xa = np.sort(F)
Na = len(F)
Fa = np.array(range(Na))/float(Na)
plt.plot(Xa,Fa,color = 'red',label = 'Test␣results')
X1 = np.sort(critload)
N = len(critload)
F2 = np.array(range(N))/float(N)
plt.plot(X1,F2, color = 'blue', label = 'Analysis␣results')
plt.xlabel('Critical␣load␣[N]', fontsize = 20)
plt.ylabel('Cumulative␣probability',fontsize = 20)
plt.axis([0,3000,0,1])
plt.show()

#CUMULATIVE WEIBULL
plt.plot(XX, stats.exponweib.cdf(XX, *stats.exponweib.fit(critload,1,1,scale=0.2, loc=0)), 'k--')
plt.plot(XX, stats.exponweib.cdf(XX, *stats.exponweib.fit(F,1,1,scale=0.2, loc=0)), 'k--')

Xa = np.sort(F)
Na = len(F)
Fa = np.array(range(Na))/float(Na)
plt.plot(Xa,Fa,color = 'red',label = 'Test␣results')
X1 = np.sort(critload)
N = len(critload)
```

```
F2 = np.array(range(N))/float(N)
plt.plot(X1,F2, color = 'blue', label = 'Analysis␣results')
plt.xlabel('Critical␣load␣[N]', fontsize = 20)
plt.ylabel('Cumulative␣probability',fontsize = 20)
plt.axis([0,3000,0,1])
plt.show()



#crLoad = np.array([critload]).T
crLoad = critload
plt.plot(np.linspace(0,3000,200),mlab.normpdf(np.linspace(0,3000,200),meanF,sigmF), linewidth = 2,
    ↪ color = 'r')
plt.hist(F, bins, fill = False, hatch='+', edgecolor='r', label = 'Test␣results',normed=1)
plt.plot(np.linspace(0,3000,200),mlab.normpdf(np.linspace(0,3000,200),mean1,sigm1), linewidth = 2,
    ↪ color = 'b')
plt.hist(critload, bins, fill = False, hatch='x', edgecolor='b', label = 'Analysis␣results', normed
    ↪ =1)
plt.xlabel('Critical␣load␣[N]',fontsize=20)
plt.ylabel('Density',fontsize=20)
plt.axis([0,3000,0,0.006])
plt.show()

#weibull
crLoad = critload
plt.hist(F, bins, fill = False, hatch='+', edgecolor='r', label = 'Test␣results',normed=1)
plt.hist(critload, bins, fill = False, hatch='x', edgecolor='b', label = 'Analysis␣results', normed
    ↪ =1)
plt.plot(XX,stats.exponweib.pdf(XX, *stats.exponweib.fit(critload,1,1,scale=0.2, loc=0)), color='b'
    ↪ , linewidth=2)
plt.plot(XX,stats.exponweib.pdf(XX, *stats.exponweib.fit(F,1,1,scale=0.2, loc=0)), color='r',
    ↪ linewidth=2)
plt.xlabel('Critical␣load␣[N]',fontsize=20)
plt.ylabel('Density',fontsize=20)
plt.axis([0,3000,0,0.006])
plt.show()

#DEFLECTION PLOTS
F = defl
critload = d
```

```
meanF = sum(defl)/len(defl)
mean1 = sum(critload)/len(critload)
sigmF = np.std(F)
sigm1 = np.std(critload)
XX = np.linspace(0,2,200)


#CUMULATIVE NORMAL DEFLECTION
plt.plot(XX, stats.norm.cdf(XX,loc=meanF, scale=sigmF), 'k--')
plt.plot(XX, stats.norm.cdf(XX,loc=mean1, scale=sigm1), 'k--')

Xa = np.sort(F)
Na = len(F)
Fa = np.array(range(Na))/float(Na)
plt.plot(Xa,Fa,color = 'red',label = 'Test␣results')
X1 = np.sort(critload)
N = len(critload)
F2 = np.array(range(N))/float(N)
plt.plot(X1,F2, color = 'blue', label = 'Analysis␣results')
plt.xlabel('Deflection␣at␣failure␣[mm]', fontsize = 20)
plt.ylabel('Cumulative␣probability',fontsize = 20)

plt.show()y

#CUMULATIVE WEIBULL DEFLECTION
plt.plot(XX, stats.exponweib.cdf(XX, *stats.exponweib.fit(critload,1,1,scale=0.2, loc=0)), 'k--')
plt.plot(XX, stats.exponweib.cdf(XX, *stats.exponweib.fit(F,1,1,scale=0.2, loc=0)), 'k--')

Xa = np.sort(F)
Na = len(F)
Fa = np.array(range(Na))/float(Na)
plt.plot(Xa,Fa,color = 'red',label = 'Test␣results')
X1 = np.sort(critload)
N = len(critload)
F2 = np.array(range(N))/float(N)
plt.plot(X1,F2, color = 'blue', label = 'Analysis␣results')
plt.xlabel('Deflection␣at␣failure␣[mm]', fontsize = 20)
plt.ylabel('Cumulative␣probability',fontsize = 20)
plt.show()


#Deflection normal histogram
crLoad = critload
plt.plot(np.linspace(0,2,200),mlab.normpdf(np.linspace(0,2,200),meanF,sigmF), linewidth = 2, color
    ↪ = 'r')
plt.hist(F, bins, fill = False, hatch='+', edgecolor='r', label = 'Test␣results',normed=1)
plt.plot(np.linspace(0,2,200),mlab.normpdf(np.linspace(0,2,200),mean1,sigm1), linewidth = 2, color
    ↪ = 'b')
plt.hist(critload, bins, fill = False, hatch='x', edgecolor='b', label = 'Analysis␣results', normed
    ↪ =1)
plt.xlabel('Deflection␣at␣failure␣[mm]',fontsize=20)
plt.ylabel('Density',fontsize=20)
plt.show()
#Deflection weibull histogram


plt.hist(F, bins, fill = False, hatch='+', edgecolor='r', label = 'Test␣results',normed=1)
plt.hist(critload, bins, fill = False, hatch='x', edgecolor='b', label = 'Analysis␣results', normed
    ↪ =1)
plt.plot(XX,stats.exponweib.pdf(XX, *stats.exponweib.fit(critload,1,1,scale=0.2, loc=0)), color='b'
    ↪ , linewidth=2)
plt.plot(XX,stats.exponweib.pdf(XX, *stats.exponweib.fit(F,1,1,scale=0.2, loc=0)), color='r',
    ↪ linewidth=2)
plt.xlabel('Deflection␣at␣failure␣[mm]',fontsize=20)
plt.ylabel('Density',fontsize=20)
plt.show()
```

Appendix C

FORTRAN source code

```
      subroutine vusdfld(
! Read only variables
     . nblock, nstatev, nfieldv, nprops, ndir, nshr,
     . jElem, kIntPt, kLayer, kSecPt,
     . stepTime, totalTime, dt, cmname,
     . coordMp, direct, T, charLength, props,
     . stateOld,
! Write only variables
     . stateNew,field)
      include 'vaba_param.inc'
!-----------------------------------------------------------------------
!     Declare general variables
!-----------------------------------------------------------------------
      dimension jElem(nblock),coordMp(nblock,*),direct(nblock,3,3),
     .          T(nblock,3,3),charLength(nblock),props(nprops),
     .          stateOld(nblock,nstatev),stateNew(nblock,nstatev),
     .          field(nblock,nfieldv)
      character*80 cmname
      real*8 stressdata(maxblk*(ndir+nshr))
      integer jSData(maxblk*(ndir+nshr))
      character*3 cSData(maxblk*(ndir+nshr))
      integer jStatus
      integer i
      real*8 s(nblock,6)
      integer flag
      data flag/0/
!-------------------------------------------------------------------------
!     Declare Glass model variables
!-------------------------------------------------------------------------
      real*8 K(nblock),sigma(nblock)
      integer nfail(nblock),nfailcrit
      real*8 theta(nblock),a(nblock)
      real*8 Y,Kcrit
      real*8 pi
!-------------------------------------------------------------------------
!     Read material properties
!-------------------------------------------------------------------------
      Kcrit     = props(1)
      Y         = props(2)
      nfailcrit = int(props(3))
      pi        = 4.0d0*atan(1.0d0)
c
      if(flag.eq.0)then
         print*,'Kcrit␣␣␣␣␣=',Kcrit
         print*,'Y␣␣␣␣␣␣␣␣␣=',Y
         print*,'nfailcrit␣=',nfailcrit
         flag = 1
      endif
!-------------------------------------------------------------------------
!     Call data from ABAQUS memory
!-------------------------------------------------------------------------
      call vgetvrm(  'S' , stressdata,jSData,cSData,jStatus)
!-------------------------------------------------------------------------
!     Extract data from ABAQUS memory
!-------------------------------------------------------------------------
      do i=1,nblock
         s(i,1) = stressdata(i)
         s(i,2) = stressdata(i+nblock)
         s(i,3) = stressdata(i+nblock*2)
         s(i,4) = stressdata(i+nblock*3)
      enddo
!-------------------------------------------------------------------------
!     Get flaw information from history variables
!-------------------------------------------------------------------------
```

```
      do i=1,nblock
         a(i)     = stateOld(i,1)
         theta(i) = stateOld(i,2)*pi/180.0
         nfail(i) = int(stateOld(i,3))
      enddo
!-----------------------------------------------------------------------
!     Compute stress intensity factor
!-----------------------------------------------------------------------
      do i=1,nblock
         sigma(i) = s(i,1)+s(i,2)-(0.5*(s(i,1)+s(i,2))
     .                         +0.5*(s(i,1)-s(i,2))*cosd(2*theta(i))
     .                         +s(i,4)*sind(2*theta(i)))
         K(i)     = Y*max(sigma(i),0.0)*sqrt(pi*a(i))
      enddo
!-----------------------------------------------------------------------
!     Check for failure
!-----------------------------------------------------------------------
      do i=1,nblock
         if(K(i).ge.Kcrit)then
            nfail(i) = nfail(i)+1
         else
            nfail(i) = 0
         endif
      enddo
!-----------------------------------------------------------------------
!     Delete elements if required
!-----------------------------------------------------------------------
      do i=1,nblock
         if(nfail(i).eq.nfailcrit)then
            statenew(i,nstatev) = 0
            nfail(i) = nfail(i)+1
         elseif(nfail(i).gt.nfailcrit)then
            statenew(i,nstatev) = 0
         else
            statenew(i,nstatev) = 1
         endif
      enddo
!-----------------------------------------------------------------------
!     Store informations
```

```
!-------------------------------------------------------------------------
      do i=1,nblock
         stateNew(i,1) = stateOld(i,1)
         stateNew(i,2) = stateOld(i,2)
         stateNew(i,3) = nfail(i)
         stateNew(i,4) = K(i)/Kcrit
      enddo
!-------------------------------------------------------------------------
!     End of subroutine
!-------------------------------------------------------------------------
      return
      end
```