Master's thesis

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

Katja Hansen

# Python/DIANA framework for robust nonlinear analysis of reinforced concrete beams

Master's thesis in Civil and Environmental Engineering
Supervisor: Daniel Cantero
June 2022

NTNU
Norwegian University of
Science and Technology

Katja Hansen

# Python/DIANA framework for robust nonlinear analysis of reinforced concrete beams

Master's thesis in Civil and Environmental Engineering
Supervisor: Daniel Cantero
June 2022

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

NTNU
Kunnskap for en bedre verden

**Department of Structural Engineering**
Faculty of Engineering
**NTNU – Norwegian University of Science and Technology**

| ACCESSIBILITY |
|---|
| Open |

# MASTER THESIS 2022

| SUBJECT AREA: | DATE: | NO. OF PAGES: |
|---|---|---|
| Nonlinear analysis of reinforced concrete | 10.06.2022 | 8 + 95 + 37 |

TITLE:

**Python/DIANA framework for robust nonlinear analysis of reinforced concrete beams**

Python/DIANA rammeverk for robust ikkelineær analyse av armerte betongbjelker

BY:

Katja Hansen

SUMMARY:
A Python/DIANA framework for robust nonlinear finite element analysis (NLFEA) of reinforced concrete (RC) beams in 2D has been created. The framework consists of four scripts which feature a combination of Python commands and specific DIANA script commands. The aim has been to create a flexible, user-friendly and robust framework for generating the DIANA workflow.

The framework provides a reliable way of approaching NLFEA of RC beams. It relies on recommended properties, which will be applied by default. These recommendations are taken from the guidelines by Rijkwaterstaat Centre of Infrastructure. Hence, the framework provides a more efficient way of modelling than through the DIANA graphical interface. While all properties have to be defined when working in the DIANA graphical interface, the user only have to define selected input utilizing the framework. In general, performing NLFEA requires experience to ensure quality, robustness and speed of the analysis. However, the provided default properties make it possible for users with limited expertise to perform complex analysis.

The flexibility of the framework is ensured through the use of object-oriented programming. Templates, examples and explanations have been added to further improve the user-friendliness of the framework. Furthermore, parametric studies can be performed using the framework. The user can select one of the input parameters to be varied, while the rest stay the same. This is a very useful feature, which can be used to perform sensitivity studies. To demonstrate this feature, a parametric study has been executed for a RC beam with varying length. The results from this parametric study demonstrated how a higher load-bearing capacity, than for sectional analysis, can be obtained using well-defined NLFEA.

The framework has been validated by recreating two well-known published experimental tests. The results of the NLFEAs have been compared to the experimental results, to validate the accuracy and reliability of the provided solution strategies. In general, the results of the performed NLFEAs agreed well with the experimental results. However, an even better agreement between the numerical and experimental results could have been obtained. The significant input parameters could have been studied in greater detail using the framework and tweaked in accordance with the experimental results. In general, the benchmark studies highlight how the default parameters provide a great starting point for robust NLFEA of RC beams.

RESPONSIBLE TEACHER: Daniel Cantero

SUPERVISOR: Daniel Cantero

CARRIED OUT AT: Department of Structural Engineering

# Preface

This master's thesis is the final work of a five-year long master's degree in Civil and Environmental Engineering, at the Norwegian University of Science and Technology (NTNU). It has been carried out over a period of 20 weeks during the spring semester of 2022, at the Department of Structural Engineering. This thesis work provides 30 credits. Daniel Cantero has been the supervisor for this thesis.

My motivation for working with this master's thesis has been to achieve a better understanding of the actual material behaviour of reinforced concrete, as well as nonlinear analysis. With no prior experience with nonlinear finite element analysis (NLFEA), the learning curve has been steep. Even so, it has been a rewarding process and I am certain that the knowledge I have gained from working with this thesis will be useful in my professional life. Hopefully, this master's thesis can also prove to be useful for other students and serve as a starting point for getting familiar with NLFEA of concrete structures.

I would like to thank my supervisor Daniel Cantero, for the support and guidance he has given throughout the work with this master's thesis. His availability for frequent meetings has been greatly appreciated.

Trondheim, Juni 2022

Katja Hansen

Katja Hansen

# Abstract

A Python/DIANA framework for robust nonlinear finite element analysis (NLFEA) of reinforced concrete (RC) beams in 2D has been created. The framework consists of four scripts which feature a combination of Python commands and specific DIANA script commands. The aim has been to create a flexible, user-friendly, and robust framework for generating the DIANA workflow. This includes modelling the geometry of beam, generating the mesh, defining and running the selected analyses and generating output.

The framework provides a reliable way of approaching NLFEA of RC beams. It relies on recommended properties, which will be applied by default. These recommendations are taken from the guidelines by Rijkwaterstaat Centre of Infrastructure [1], which are based on long-term experience and scientific research. Hence, the framework provides a more efficient way of modelling than through the DIANA graphical interface. While all properties have to be defined when working in the DIANA graphical interface, the user only has to define selected input utilizing the framework. In general, performing NLFEA requires experience to ensure quality, robustness and speed of the analysis. However, the provided default properties make it possible for users with limited expertise to perform complex analysis.

The flexibility of the framework is ensured through the use of object-oriented programming. Templates, examples and explanations have been added to further improve the user-friendliness of the framework. Furthermore, parametric studies can be performed using the framework. The user can select one of the input parameters to be varied, while the rest stay the same. This is a useful feature, which can be applied to evaluate the significance of different input parameters on the results of the NLFEA. To demonstrate this feature, a parametric study has been executed for a RC beam with varying length. The results from this parametric study demonstrated how a higher load-bearing capacity, than for sectional analysis, can be obtained using well-defined NLFEA.

The framework has been validated by recreating two well-known published experimental tests. The results of the NLFEAs have been compared to the experimental results, to validate the accuracy and reliability of the provided solution strategies. In general, the results of the performed NLFEAs agreed well with the experimental results. However, an even better agreement between the numerical and experimental results could have been obtained. The significant input parameters could have been studied in greater detail using the framework and tweaked in accordance with the experimental results. In general, the benchmark studies highlight how the implemented default parameters provide a great starting point for robust NLFEA of RC beams.

# Sammendrag

Et Python/Diana-rammeverk for robust ikkelineær elementanalyse (NLFEA) av armerte betongbjelker i 2D har blitt laget. Rammeverket består av fire skript som inneholder en rekke instruksjoner bestående av Python-kommandoer og spesifikke DIANA kommandoer. Målet har vært å skape et fleksibelt, brukervennlig og robust rammeverk som kan brukes til å generere DIANA sin arbeidsflyt. Dette inkluderer å modellere geometrien til bjelken, generere elementinndelingen (*mesh*), definere og kjøre de valgte analysene og å generere utdata.

Rammeverket gir en pålitelig måte å tilnærme seg NLFEA av armerte betongbjelker. Det baserer seg på anbefalte egenskaper, som vil bli brukt som standard. Disse anbefalingene er hentet fra retningslinjene til Rijkswaterstaat [1], som baserer seg på lang erfaring og vitenskapelig forskning. Følgelig gir rammeverket en mer effektiv måte å modellere på, enn via DIANA sitt grafiske brukergrensesnitt. Mens alle egenskaper må defineres når en arbeider i det grafiske DIANA-grensesnittet, trenger brukeren kun å definere utvalgte inndata ved bruk av rammeverket. Generelt krever utførelse av NLFEA erfaring for å sikre kvaliteten, robustheten og hastigheten til analysen. De angitte standardegenskapene gjør det imidlertid mulig for brukere med begrenset ekspertise å utføre komplekse analyser.

Fleksibiliteten til rammeverket er ivaretatt gjennom bruk av objekt-orientert programmering. Maler, eksempler og forklaringer har blitt lagt til for å ytterligere forbedre brukervennligheten. Videre kan parametriske studier utføres ved hjelp av rammeverket. Brukeren kan velge å variere en av inndataparameterne, mens resten forblir de samme. Dette er en nyttig funksjon som kan brukes til å evaluere betydningen forskjellige inndataparametere har på resultatene fra NLFEA-en. For å demonstrere denne funksjonen er det blitt utført en parametrisk studie av en armert betongbjelke med varierende lengde. Resultatene fra denne parametriske studien illustrerte hvordan høyere bæreevne, enn for analytisk beregnet tverrsnittskapasitet, kan bli oppnådd ved bruk av en veldefinert NLFEA.

Rammeverket har blitt validert ved å gjenskape to velkjente publiserte eksperimentelle tester av armerte betongbjelker. Resultatene fra NLFEA-ene har blitt sammenlignet med de eksperimentelle resultatene for å validere nøyaktigheten og påliteligheten til de angitte løsningsstrategiene. Generelt sett stemte resultatene fra de utførte NLFEA-ene godt overens med de eksperimentelle resultatene. Bedre samsvar mellom de numeriske og eksperimentelle resultatene kunne imidlertid blitt oppnådd. De signifikante inndataparameterne kunne blitt studiert nøyere ved hjelp av rammeverket, og justert i samsvar med de eksperimentelle resultatene. Generelt fremhever referansestudiene hvordan de implementerte standardparametrene gir et godt utgangspunkt for robust NLFEA av armerte betongbjelker.

# Abbreviations

The following abbreviations have been used throughout this thesis:

NS-EN 1992-1-1:2004+A1:2014+NA:2021 [2] is referred to as EC2.

fib Model Code for Concrete Structures 2010 [3] is referred to as MC2010.

The DIANA FEA software, version 10.5, is referred to as DIANA.

Rijkwaterstaat Centre of Infrastructure is referred to as Rijkwaterstaat.

The four scripts that make up the framework are referred to as the Beam Script.

# Table of Contents

# 1 Introduction

## 1.1 Background

The interest in nonlinear finite element analysis (NLFEA) of concrete structures has increased steadily in the recent years, due to the wide use of concrete as a structural material and the rapid development of faster and more powerful digital computers and computational programs. The desire for more efficient and sustainable design has powered this interest. A more accurate analysis of the damage development and capacity of a reinforced concrete (RC) structure can be performed using NLFEA. In general, a higher load-bearing capacity than for linear analysis will be obtained using NLFEA, as the nonlinear material properties of the concrete, such as yielding and cracking, will be taken into consideration. The redistribution of stresses in the concrete after cracking due to the bond between the concrete and reinforcement, usually results in an increased strength and stiffness of the structure. Hence, a well-defined nonlinear analysis with a high level-of-approximation allows for optimal design of RC structures, including increased material efficiency, increased service life and reduced costs.

DIANA FEA (further called DIANA) is a finite element analysis solver with strong focus on reinforced concrete analysis. It is possible to generate numerical models in DIANA by running Python scripts. By combining Python and DIANA, a framework can be created to generate the DIANA workflow. This workflow includes modelling the geometry, generating the mesh, running analyses and generating output. Python scripting also allows for adjustable user input, such that the same script can be used to model a large number of different RC beams. Hence, a more efficient and flexible way of modelling in DIANA than through the graphical interactive interface, can be achieved by combining DIANA and Python.

## 1.2 Method

For this master's thesis, a Python/DIANA framework for robust NLFEA of RC beams in 2D has been created. The framework consists of four scripts which features a combination of Python commands and specific DIANA script commands. These scripts will be referred to as the Beam Script. The aim has been to create a flexible, user-friendly and robust framework for generating the DIANA workflow. Compared to linear analysis, NLFEA requires costly load incrementation and iterative schemes. Informed choices therefore need to be taken in order to ensure the quality, robustness and speed of the analysis. For this reason, default properties and values have been provided in the script, based on recommendations for NLFEA of concrete structures [1]. This enables the user to perform complicated analysis, even if they have limited expertise when it comes to NLFEA of concrete structures. Furthermore, the framework features flexible user input, which allows the user to easily model RC beams with different geometry and material properties. The option to perform a parametric study, where one of the input parameters can be varied while the rest stay the same, has also been implemented. To further improve the user-friendliness of the Beam Script, explanations, examples and templates for creating the different components and properties of the RC beam have been added. Object-oriented programming in Python has been used to create a flexible structure for the Beam Script and to provide the default properties.

## 1.3 Previous work

Previous work that is worth mentioning when it comes to the approach for robust NLFEA of RC structures, includes the guidelines by Rijkwaterstaat Centre of Infrastructure [1] and the master's thesis by Arjen de Putter [4]. On the initiative of the Dutch Ministry of Infrastructure and Water Management, guidelines for NLFEA of concrete structures have been developed. These guidelines are based on scientific research, consensus among peers, and long-term experience with nonlinear analysis of concrete structures by the contributors to the guidelines [1]. The default properties of the framework are based on the recommendations from the guidelines by Rijkswaterstaat. These recommendations will also be referred to frequently in Part I, where theory and recommendations for NLFEA of RC structures will be explained. The master's thesis by de Putter includes the development of 119 solution strategies for application of NLFEA for concrete structures [4]. A beamscript [5] was created in this regard, and has been used as an inspiration for the created Beam Script. The DIANA Documentation [6] has also been an important document for understanding how to perform NLFEA of RC structures in DIANA.

## 1.4 Thesis outline

This master's thesis is divided into five parts and consists of 14 chapters in total. This introduction is followed by an account of the theory and recommendations for NLFEA of RC structures in Part I. Part II will explain certain aspects of modelling in DIANA and Python. The Beam Script itself will be presented in Part III. In Part IV, two well-known published experiments will be recreated to benchmark the Beam Script, and in Part V a parametric study will be performed using the Beam Script. The thesis is structured with a discussion in Chapter 10, Chapter 11 and Chapter 13, relating to the obtained results of the performed NLFEAs using the created framework. Chapter 14 will recap the previous results and discussions, include some general observations from creating and using the framework and provide recommendations for future work.

**Part I - Theory and recommendations**
This part addresses some of the key aspects of performing a NLFEA. Furthermore, the relevant aspects and recommendations for finite element modelling of reinforced concrete that have been applied in the created framework will be addressed. Part I consists of Chapter 2 and Chapter 3.

**Part II - Modelling in DIANA and Python**
This part will provide some general information about object-oriented programming in Python and how this has been put to practical use in the Beam Script. Certain aspects of modelling in DIANA will be explained as well. Part II consists of Chapter 4, Chapter 5 and Chapter 6.

**Part III - The Beam Script**
The Beam Script will be presented in this part. First, some general info about the Beam Script will be explained in Chapter 7, then the input script will be explained in more detail in Chapter 8. The latter will be structured similarly to the created input script. Part III consists of Chapter 7 and Chapter 8.

**Part IV - Experiments to benchmark the script**
The Beam Script has been validated in this part by recreating two well-known published experi-

mental tests. The results of the NLFEAs will be compared to the experimental results, to validate the accuracy and reliability of the selected solution strategy. Part IV consists of Chapter 9, Chapter 10 and Chapter 11.

**Part V - Parametric study**

The framework allows for the execution of a parametric study, where one of the input parameters can be varied. To demonstrate this feature, a very simple parametric study has been executed in this part, where the length of a beam has been varied from 3m to 6m. Part V consists of Chapter 12 and Chapter 13.

**Conclusions and recommendations**

Chapter 14 summarizes the main features of the created DIANA/Python framework, as well as the results and discussions from Part IV and Part V. Recommendations for future work are also included in this final chapter.

# Part I

# Theory and recommendations

# 2 Nonlinear finite element analysis

This chapter addresses some of the key aspects of performing a NLFEA. The mentioned recommendations are based on the guidelines by Rijkswaterstaat [1]. Special attention has been given to the methods and procedures for NLFEA that has been used in the Beam Script.

## 2.1 Finite element analysis

The practical application of the finite element method (FEM) to solve engineering problems is called finite element analysis (FEA). The finite element method gives an approximate numerical solution to a boundary value problem, also called a field problem [7]. Mathematically, field problems are expressed by partial differential equations (PDEs) or an integral expression. Compared to calculus, FEM uses finite elements instead of infinitesimal elements, where the structure to be analysed is divided into smaller pieces/elements. Each of these elements have a much simpler behaviour compared to the whole structure. The behaviour of each element is defined by degrees of freedom at the nodal points. The nodal points (nodes) are the connections between the different elements of the structure. In reality, the behaviour of the structure is more complicated than what will be represented by the elements, hence FEM gives an approximate solution. This solution can become more accurate by increasing the number of elements and by carefully choosing the most suitable type of elements [8].

## 2.2 Nonlinear finite element analysis

Nonlinear finite element analysis (NLFEA) will give a better level-of-approximation of a concrete structure than linear analysis. NLFEA takes into account the nonlinear material properties of the concrete and the reinforcement, like cracking and yielding, as well as the influence of changing geometry on the structural response [9]. Hence, a more accurate analysis of damage development and capacity of the reinforced concrete structure can be done. However, NLFEA requires, in contrast to the linear method, costly load incrementation and iterative schemes to find the structural stiffness [8].

In nonlinear analysis one often consider three types of nonlinearity [7]:

- Material nonlinearity (E.g. nonlinear elasticity, plasticity, creep)
- Contact nonlinearity (E.g. changing contact forces)
- Geometric nonlinearity (E.g. large deformations changing the structural geometry)

All these cases are nonlinear as the stiffness, and sometimes load, become functions of the displacement or deformation (the degree of freedom) [7]. For linear finite element method the structural equation can be written as:

$$[K]\{D\} = [R] \tag{2.1}$$

While, for nonlinear finite element method the stiffness and load become functions of {D}:

$$[K(D)]\{D\} = [R(D)] \tag{2.2}$$

Since Equation 2.2 is nonlinear, the principle of superposition is no longer valid. A nonlinear system of equations therefore have to be solved iteratively until equilibrium is reached. As with any iterative method there is no guarantee that the iterations will reach convergence. However, if the load steps are smaller (i.e. applying load incrementally), the method will have improved stability [10]. To do a successful NLFEA, a better understanding of the equation-solving procedures is required than for linear analysis [7]. The solving strategy might have to be changed, and several attempts might have to be done in order to achieve a good enough analysis.

### 2.2.1 Solution procedure

To determine the state of equilibrium, the problem will be discretized in space (using finite elements) as well as in time (using increments). An iterative method is used to reach equilibrium at the end of each increment. This combination is called an incremental-iterative solution procedure [6]. To maintain the solution on the path of equilibrium, informed choices about the incrementation procedure, the iterative solution method and the convergence criteria are needed [11].

For equilibrium to be reached, the external forces must equal to the internal forces, or the difference between them have to be within an accepted tolerance.

$$\mathbf{f}_{ext} = \mathbf{f}_{int} \tag{2.3}$$

If only one increment is considered, the internal and external forces will simply depend on the displacement increment, $\Delta\mathbf{u}$ [6]:

$$\mathbf{g}(\Delta\mathbf{u}) = \mathbf{f}_{ext}(\Delta\mathbf{u}) - f_{int}(\Delta\mathbf{u}) = \mathbf{0}, \tag{2.4}$$

$$^{t+\Delta t}\mathbf{u} = {}^{t}\mathbf{u} + \Delta\mathbf{u}, \tag{2.5}$$

where $\mathbf{g}$ is the residual forces.

#### 2.2.1.1 Iterative solution method

A solely incremental solution procedure would require the displacement increment, $\Delta\mathbf{u}$, to be very small in order to achieve an accurate solution. Combining the incremental process with an iterative procedure, creating an implicit procedure, allows for a higher increment size [6]. The iteration process for each increment is shown in Figure 2.1.

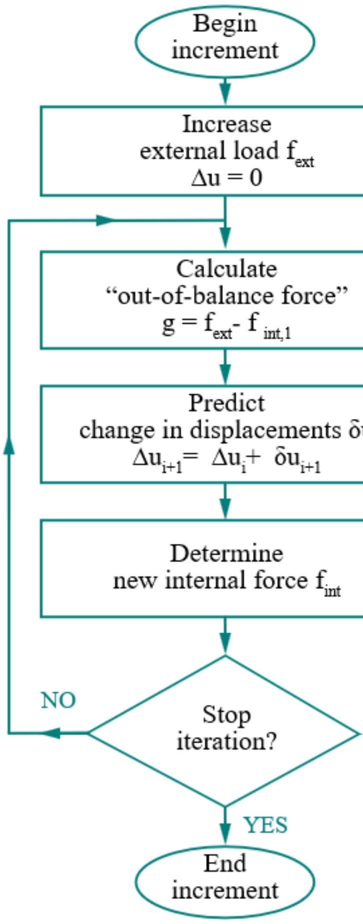



**Figure 2.1:** Iteration process [6]

$\Delta\mathbf{u}$ for each increment is decided iteratively by adding $\delta\mathbf{u}$ increments until equilibrium or an accepted tolerance is reached. For each iteration, the new $\Delta\mathbf{u}_{i+1}$ can be calculated as follows:

$$\Delta\mathbf{u}_{i+1} = \Delta\mathbf{u}_i + \delta\mathbf{u}_{i+1} \tag{2.6}$$

A direct approach can be used to find $\delta\mathbf{u}$, based on the linear relationship presented in Equation 2.1:

$$\delta\mathbf{u}_i = \mathbf{K}_i^{-1}\mathbf{g}_i \tag{2.7}$$

#### 2.2.1.1.1 Newton-Raphson

Several iterative procedures are available when using DIANA. Newton-Raphson (NR) is one of the most common methods and is the recommended method by Rijkswaterstaat [1]. The method is a pure iterative procedure, and can be combined with for example a line search algorithm which is explained in Section 2.2.1.1.2. For the Newton-Raphson iteration, the stiffness matrix $\mathbf{K}_i$ is calculated based on the tangential stiffness of the structure [6]:

$$\mathbf{K}_i = \frac{\partial \mathbf{g}}{\partial \Delta \mathbf{u}} \tag{2.8}$$

The two most common types of the Newton-Raphson methods are Regular and Modified Newton-Raphson. These methods differ due to the point at which $\mathbf{K}$ is evaluated [6]. Both methods are shown in Figure 2.2.

For the Regular Newton-Raphson method, $\mathbf{K}$ is evaluated at every iteration. In other words, each evaluation of $\mathbf{K}_i$ is based upon the last predicted situation, even though this might not be an equilibrium state. The Regular Newton-Raphson method is effective as it might exhibit a quadratic rate of convergence, which means that the method converges to the final solution in few iterations. On the other hand, each of these iterations might be relatively time-consuming. One therefore has to be careful when choosing the increment size, and some degree of trial and error will most likely have to be done in order to decide appropriate load increments.

The Modified Newton-Raphson method only evaluates $\mathbf{K}$ at the beginning of each increment. This means that the stiffness matrix is always predicted based on an equilibrium state [6]. The method usually has a slower convergence than the Regular Newton-Raphson method, and therefore needs more iterations. However, every iteration is faster than Regular Newton-Raphson. In situations where the Regular Newton-Raphson method does not converge, the Modified Newton-Raphson method still might [6].

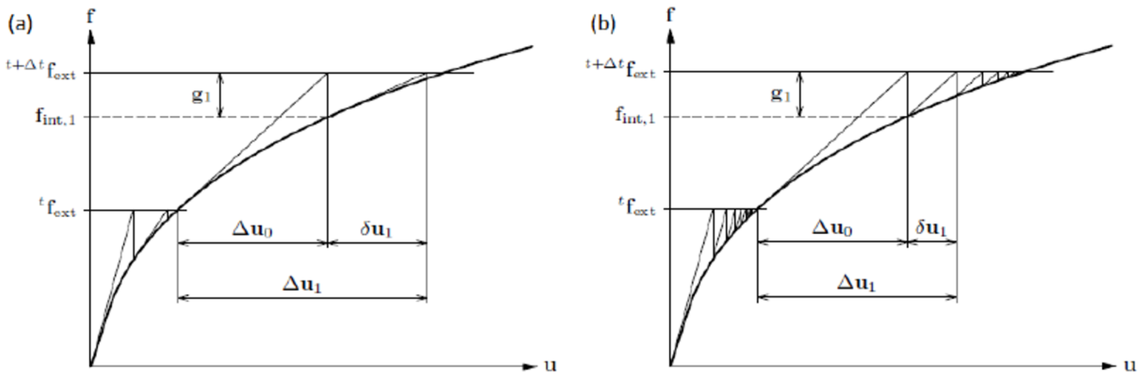



**Figure 2.2:** Newton-Raphson iterations a) Regular NR b) Modified NR [6]

#### 2.2.1.1.2 Line Search

To help convergence, a line search algorithm can be applied. This method is particularly useful for highly nonlinear structures, like reinforced concrete which experience cracking and steel yielding, as this leads to rapid changes in the structural stiffness [8]. Line search may be used in all type of NR methods. Instead of updating the previous solution by the entire increment, $\delta\mathbf{u}$, as done in Equation 2.6, this increment can be scaled by a factor, $\eta$, as shown in Equation 2.9. The procedure for obtaining the optimal choice of $\eta$ is called a line search algorithm. The line search obtains an optimal incremental step length by minimising the residual force in the predicted direction [8].

$$\Delta\mathbf{u}_{i+1} = \Delta\mathbf{u}_i + \eta \cdot \delta\mathbf{u}_{i+1} \tag{2.9}$$

#### 2.2.1.2 Incremental procedure

The two most common incremental procedures are load control and displacement control, as shown in Figure 2.3. For each increment, the applied load on the system is increased. When using load control, this is done directly by increasing the external load vector $\mathbf{f}_{ext}$. For displacement control, the external load is put on the structure by prescribed displacements [6].

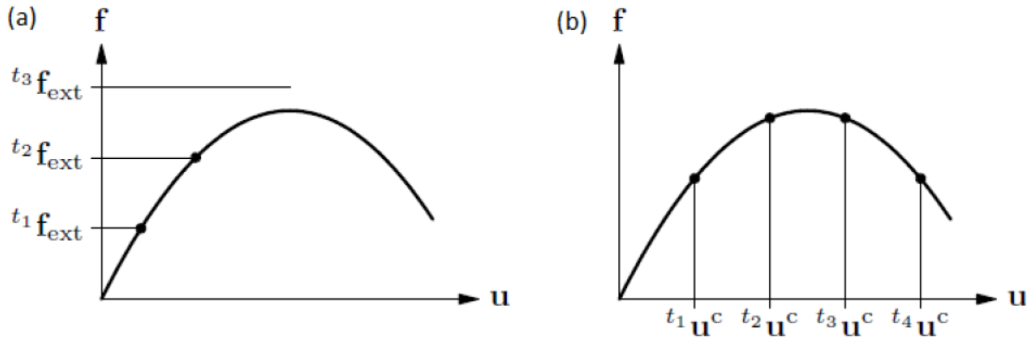
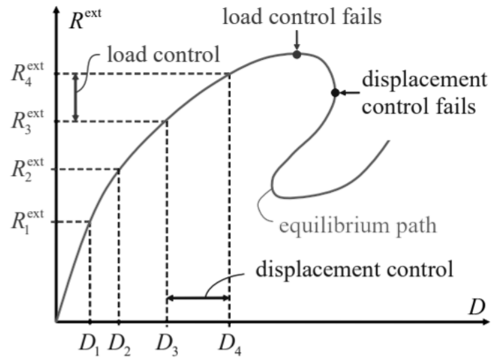


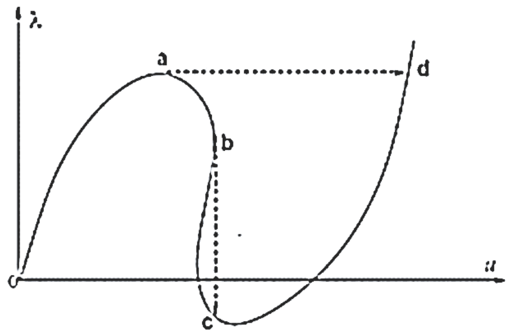

**Figure 2.3:** Incremental procedures a) Load control b) Displacement control [6]

Looking at the load-displacement graph after a nonlinear analysis, simply applying load control will not provide relevant results after the maximum point (failure load). After this point less force would be needed to displace the structure further, as can still be shown in the results for displacement control. With load control the only possibility is to further increase the external force with each load step, while displacement control will show that it takes less load to displace the structure after this point.

Both load control and displacement control might fail at critical points, as shown in Figure 2.4a. At the critical points an instability might cause the analysis to fail, due to either snap-through (load control) or snap-back (deformation control) behaviour [8]. As shown in Figure 2.4b, the analysis might fail to converge after the critical point, or end up following a misleading path. For the load control method, the analysis may fail to converge after point $a$ or trace the path $o-a-d$. For the displacement control method, the analysis may fail to converge after point $b$ and describe the path $o-a-b-c$ [12].




**(a)** Critical points on equilibrium path [8]




**(b)** Snap-back (o-a-b-c) and snap-through (o-a-d) behaviour [12]

**Figure 2.4:** Critical points where load and displacement control might fail

Even though displacement control is a more robust method, the guidelines does in general recommend load control [1]. Displacement control can be relevant for research-oriented analyses, where a concentrated load can be replaced by an equivalent displacement. Generally, displacement control is not recommended as it will restrict the displacement of a point to a decided value, which is not suited for multiple loads and/or distributed loads. Load control is therefore recommended, to take distributed loads, like dead weight, into consideration. A force controlled analysis normally results in a requirement for arc-length control, which is further discussed in Section 2.2.1.2.1. If arc-length control is not applied together with load control, the analysis will not be able to capture softening behaviour without diverging [4]. This is essential for a correct model of the concrete material, as further discussed in Chapter 3.

#### 2.2.1.2.1 Arc-length method

The arc-length method is a variation of the iteration scheme, which adapts the increment size [6]. It is recommended for stability reasons, as it allows the simulation to continue beyond a local or global maximum in the load-deflection response [1]. The arc-length method is able to pass the points where snap-back and snap-through will happen for the other control methods, as shown in Figure 2.4b. With the arc-length method both displacement and load increments are managed at the same time, by controlling the "arc-length" of the combined displacement-load increment during the equilibrium iterations [8]. The idea behind the method is that instead of using fixed increments, as for load and displacement control, both the load and displacement increments are modified during iterations. The arc-length method is especially useful when combined with adaptive load increments [6].

#### 2.2.1.2.2 Adaptive load incrementation

The load incrementation can be done manually, however this often proves to be problematic when the solution path is nonlinear. Larger increments may be suitable when the equilibrium path is almost linear, and smaller ones are needed where the path is highly nonlinear [8], as exemplified in Figure 2.5. Rijkswaterstaat therefore recommends using an automatic procedure [1]. The load increment, $\Delta\lambda$, leading to the first crack can easily be decided with a linear-static analysis. The subsequent increments should be determined using an automatic procedure. Instead of treating the increment size as a fixed value, the automatic procedures uses adaptive loading to allow for result dependent increment sizes [6]. Usually the amount of nonlinearity of an increment is not known beforehand, and the optimal increment size can therefore not be fixed before the analysis starts. Two of the most common automatic procedures for adaptive loading are the iteration based method and the energy based method.

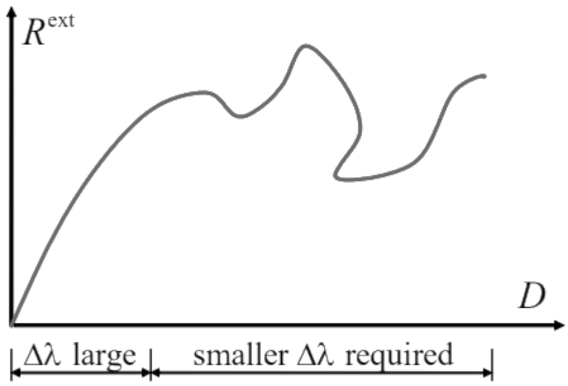



**Figure 2.5:** A nonlinear solution path [8]

The **iteration based** method can be used for all types of loading. The method is based upon a user-decided 'desired number of iterations', $N^d$, and the increment size can be made larger or smaller depending on the number of iterations of the previous increment, ${}^tN$. The method is suitable for passing snap-through behaviour and more stable in case of softening behaviour [6]. The size of the the new load increment, ${}^{t+\Delta t}\Delta\lambda_0$, is calculated as follows [6]:

$$ {}^{t+\Delta t}\Delta\lambda_0 = \frac{{}^t\Delta l}{\sqrt{\delta\mathbf{u}_0^T\delta\mathbf{u}_0}}(\frac{N^d}{{}^tN})^{\gamma} \tag{2.10}$$

${}^t\Delta l$ is the length of the predictor displacement of the previous step and $\gamma$ is usually set to 0.5.

The **energy based** method can only be used in combination with arc-length control. This method calculates the vector product of the load increment and the displacement increment (the work done), such that the first prediction equals the final prediction of the previous step [6]. The method is shown in Figure 2.6.

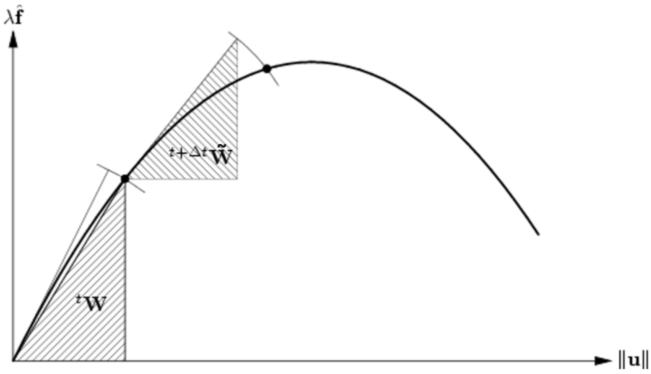



**Figure 2.6:** Work increment used in energy based adaptive loading [6]

${}^{t}\mathbf{W}$ equals the final vector product of the previous step, while ${}^{t+\Delta t}\tilde{\mathbf{W}}$ is the first prediction of the vector product for the current step.

#### 2.2.1.3 Convergence criteria

For defining when equilibrium is satisfied, DIANA offers several convergence norms. The iteration process will be stopped in case of convergence. Besides convergence, the iteration process will also be stopped if the maximum number of iterations (decided by the user) has been reached, or in case of divergence. The convergence criteria must be carefully chosen. Too loose criteria will give inaccurate results, while too strict criteria will be time-consuming and not very economical [8]. The most common convergence criteria in NLFEA are based on displacements, residual forces or energy. In general, a displacement based criteria is not recommended as it can be misleadingly satisfied by a slow convergence rate [8]. Therefore, Rijkswaterstaat recommends using a force norm combined with an energy norm, and to avoid using a displacement norm [1]. When prescribing multiple convergence norms, DIANA will terminate the iteration process if one of the criteria is satisfied, unless otherwise specified [6].

The force norm is the Euclidian norm of the out-of-balance force vector $\boldsymbol{g}$, while the energy norm uses the internal force. The norms are defined as follows [6]:

$$\text{Force norm ratio} = \frac{\sqrt{\mathbf{g}_i^T \mathbf{g}_i}}{\sqrt{\mathbf{g}_0^T \mathbf{g}_0}} \tag{2.11}$$

$$\text{Energy norm ratio} = \left| \frac{\delta \mathbf{u}_i^T (\mathbf{f}_{int,i+1} + \mathbf{f}_{int,i})}{\Delta \mathbf{u}_0^T (\mathbf{f}_{int,1} + \mathbf{f}_{int,0})} \right| \tag{2.12}$$

##### 2.2.1.3.1 Nonconvergence

In practical application of NLFEA, steps with nonconvergence are almost unavoidable [13]. Successive cracking and local effect may lead to convergence issues, but do not necessary mean failure. Rijkswaterstaat advises that load increments which do not completely fulfil the convergence criteria still can be included, if they are followed by a converged load increment and a plausible explanation for the nonconvergence is provided [1]. In case of nonconvergence, one could check the norms and see how far from convergence the results are. To improve convergence the number of iterations can be increased, the size of the increments reduced or one could change the iteration method [6].

### 2.2.2 Finite element discretization

The model of the structure to be analysed is discretized in space using finite elements. It generally requires skills and experience to be able to define a finite element (FE) mesh that is cost-effective and provides an accurate enough solution [8]. The quality of the analysis is influenced by the shape of the elements, the degree of interpolation of the displacement field, and the numerical integration scheme [1].

#### 2.2.2.1 Finite elements for concrete

Rijkswaterstaat recommends using elements with a quadratic interpolation of the displacement field. These elements have three nodes at each edge. The preferred element for analysis of a RC beam in 2D is an 8-node quadrilateral element [1], as shown in Figure 2.7. This element is called CQ16M and is an eight-node quadrilateral isoparametric plane stress element. The isoparametric formulation allows the quadrilateral element to have nonrectangular shape and curved sides [14]. By default DIANA applies $3 \times 3$ Gaussian integration for CQ16M, which is equivalent to a higher integration scheme [6]. Rijkswaterstaat recommends full integration, as a reduced integration scheme can lead to spurious modes when the RC beam starts to experience extensive cracking.

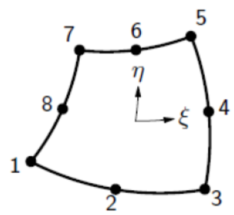



**Figure 2.7:** CQ16M [6]

The size of the mesh is important, as it amongst others influences how the stress-strain relationship, the geometry and the expected damage distribution are captured [1]. The minimum element size should be 1.5 times the maximum aggregate size [1]. However, most often the minimum element size is governed by practical considerations, as smaller elements will highly increase the computational time. The maximum element size should be chosen such that a relatively smooth stress fields can be calculated. For 2D modelling of beams, at least 6 elements over the height of the beam should be used [1]. To achieve mesh independence, the size of the elements should be such that the results with elements of size $l$ are equal to the results when using elements of size $l/2$ [15]. In other words, further refinement of the mesh should not influence the results of the analysis.

#### 2.2.2.2 Finite elements for reinforcement

The reinforcement elements adds stiffness to the finite element model. Embedded reinforcement elements are recommended [1]. These elements are embedded in the structural elements, therefore having no degrees of freedom of their own [6]. This technique allows the lines of the reinforcement to deviate from the lines in the mesh. The reinforcement elements should be compatible with the elements in which the reinforcement is embedded. Hence, a quadratic interpolation of the displacement field should be used, as for the finite elements for concrete. As the reinforcement elements are embedded, both regular and reduced integration can be used without the risk of spurious modes [6].

# 3 Modelling of reinforced concrete

This chapter explains the relevant aspects and recommendations for finite element modelling of reinforced concrete that have been applied in the Beam Script. Special attention has been given to the nonlinear properties of the concrete, as the framework aims to model the behaviour of RC beams with a high level-of-approximation. With this regard, crack modelling has been given special attention in this chapter.

## 3.1 Constitutive model for concrete

A constitutive model, also called a material model, describes mathematically the material responses to mechanical (and/or thermal) loading, giving the stress-strain relationship of the material [16]. Constitutive models are a simplification of the actual material behaviour. Depending on the purpose of the analysis and the desired accuracy, a suitable material model can be chosen.

Reinforced concrete is a quasi-brittle material [6]. The behaviour of the material is largely influenced by tensile cracking and compressive crushing of the concrete, as well as yielding of the reinforcement. In addition, the long-term effects of shrinkage and creep also characterize the constitutive behaviour of the material. For the scenarios presented in this thesis, the long-term effects need not be considered. The material response of simple reinforced concrete quickly becomes difficult to model correctly, as the behaviour is not straightforward. During loading new cracks may form, already existing cracks may propagate or close, and the stresses in the reinforcement bars will vary with the crack development [17]. In a cracked concrete, crack surfaces will be able to transfer shear and compression at contact points, while tension will not be transmitted. Tensile stresses will however still exist in the concrete laying between the cracks. An accurate constitutive model needs to take all this into consideration.

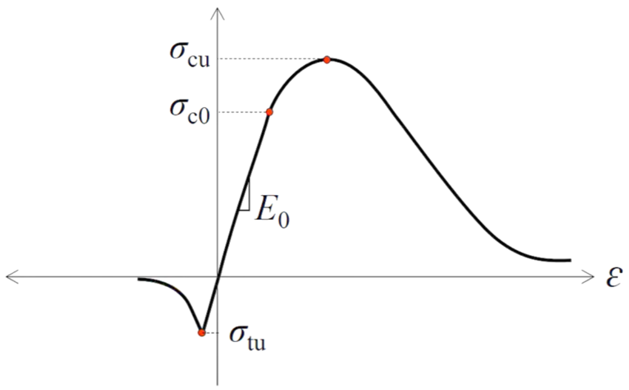



**Figure 3.1:** Typical stress-strain curve of concrete [18]

A typical stress-strain curve of concrete [18] is shown in Figure 3.1, with compression denoted as positive on the right-hand side and tension on the left-hand side. $\sigma_{cu}$ is the ultimate compressive strength of concrete, often called the crushing strength. $\sigma_{c0}$ is the compressive strength of concrete that marks the onset of plastic deformation, thus the theoretical end of the elastic branch in the stress-strain curve. $\sigma_{tu}$ is the ultimate tensile capacity, beyond which cracking will occur.

## 3.2 Crack modelling

The two main approaches for modelling plain concrete are discrete and smeared cracking. Discrete cracking is directly modelled as a discontinuity between two elements, using an interface [15]. The material model then defines how the crack behaves. In contrast, using smeared cracking, the cracked material is modelled as a continuous, anisotropic medium. A discrete model is recommended when the places the cracks will occur is known, as for example in a laboratory experiment. The behaviour, like opening, sliding and closing, of the specified cracks can then be monitored. Smeared cracking, on the other hand, is used when one does not know exactly where the cracks will occur. This method is the most commonly used approach [15]. Cracking is described by means of stress - (crack) strain relations for smeared cracking [15]. The smeared cracking assumption can be used for both compression failure and shear failure, and is the recommended method for crack modelling [1].

### 3.2.1 Crack bandwidth for smeared cracking

One of the challenges of smeared cracking is mesh sensitivity at material level [15]. The element size determines the amount of energy that is dissipated with the crack. A solution to this problem is to introduce the crack bandwidth, $h_{cr}$. This is a commonly used parameter in constitutive models with a smeared crack approach, and can be described as a length scale used to normalise the effect of the element size in energy redistribution [15]. The crack bandwidth, $h_{cr}$, also known as the equivalent length, $h_{eq}$, should be determined using an automatic procedure [1]. DIANA offers two automatic methods: Rots' element based method and Govindjee's projection method. While Rot's method takes the shape and other properties of the element into consideration, Govindjee's method also accounts for the direction of the crack [4]. Therefore, Godvindjee's method is preferred [1]. Figure 3.2 shows examples of crack bandwidths when the crack direction is taken into consideration. For a square-shaped quadratic quadrilateral element, the estimated crack bandwidth is $h_{cr} = \sqrt{2}h$.

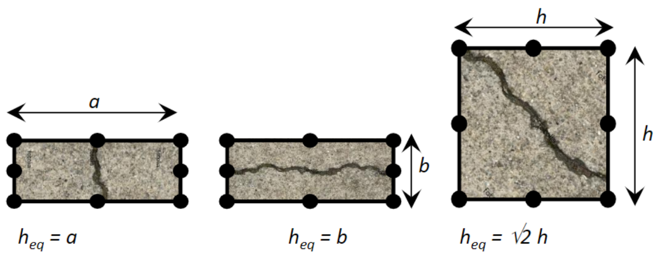



**Figure 3.2:** Crack bandwidths when crack orientation is considered [1]

## 3.3 Total Strain Based Cracking

DIANA offers several models for smeared cracking. Rijkswaterstaat recommends using a total strain based crack model (TSCM) [1]. This model describes the stress as a function of the strain. The method is based upon the modified compression-field theory presented by Vecchio & Collins [17] and its 3D extension by Selby & Vecchio [19]. The total strain based crack model is commonly used due to its robustness [15]. The input for the TSCM consist of two parts: i) the basic material properties, like the Poisson's ratio and the Young's modulus, and ii) definitions of the material behaviour in tension, compression and shear. The material properties should be obtained from EC2, and material properties that are not described in EC2 should be taken from MC2010 [1]. Lateral influence models may also be applied to describe the effect of lateral cracking and confinement on the material behaviour.

### 3.3.1 Fixed and rotating crack models

There are three variants of the TSCM: fixed, rotating and rotating to fixed. For the last variant, a threshold strain decides when to switch from a rotating model to a fixed model. For all variants, the stress is evaluated in the directions which are given by the crack directions [6]. The crack directions are either fixed or continuously rotating with the principal directions of the strain vector.

The **rotating crack model** is a computational procedure with a coaxial stress-strain concept [6]. The crack plane rotates to follow the principal directions, as shown in Figure 3.3. The principal plane will always coincide with the crack plain, so no shear component will be present.

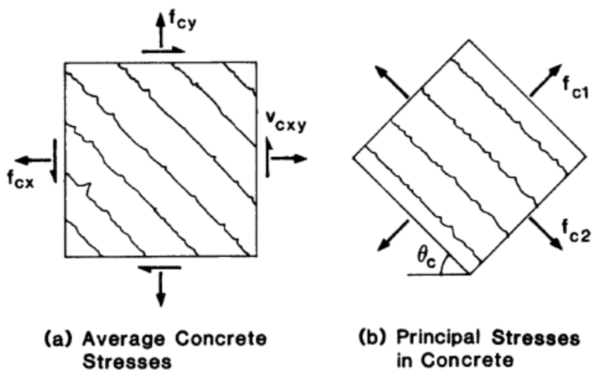



**Figure 3.3:** Stresses in cracked concrete [17]

The **fixed crack model** has a fixed stress-strain concept. The stress-strain relations are assessed in a fixed coordinate system, which is fixed upon cracking [6]. In the event of using a fixed crack model, this should be combined with an appropriate shear retention model [1], which is further explained in Section 3.3.2.

According to Rijkswaterstaat, a rotating crack model should be used. The argument is that a rotating model will result in a lower-limit failure load and suffer less from spurious stress locking [1]. Stress locking is an error that can occur in a finite element analysis where a smeared crack model is used. Since the cracks are modelled as a distributed effect and not as an actual geometric discontinuity, reduction of stress in a cracked integration point will not cause a relaxation of the neighbouring elements [4]. Hence, deformation of the cracked element results in spurious stresses being 'locked' in around the localised cracks, making the elements appear stiffer than they actually are. Figure 3.4 shows an example of this behaviour.

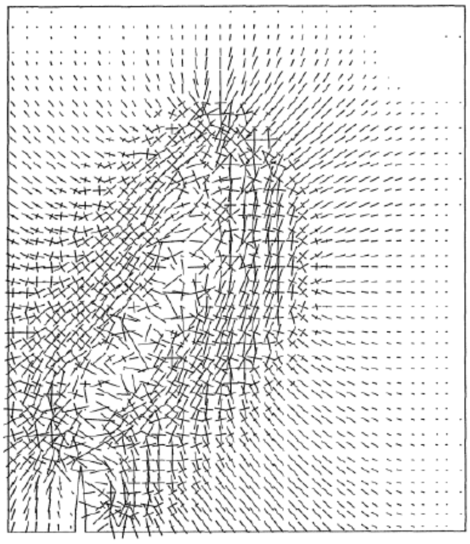


**Figure 3.4:** Severe stress locking around a fixed crack [4]

According to de Putter et. al. [13], the most suitable variant of the TSCM depends upon the failure mode. A rotating crack model performs well for the relatively ductile failures in beams with stirrups, while a fixed crack model performs better for the more brittle failures in beams without stirrups [13].

### 3.3.2 Poisson effect and shear behaviour

Uncracked concrete can be modelled as a linear-elastic isotropic material. The stress-strain relation is then given by the Young's modulus, $E$, and the Poisson's ratio, $\nu$. While $E$ is derived from the characteristic cylinder compressive strength, $\nu = 0.2$ can be used for the initial Poisson's ratio regardless of the concrete strength [1]. However, as the concrete starts cracking, these parameters should be reduced [1]. The Poisson effect will cease to exist in cracked concrete, that is to say the stretching of a cracked direction will no longer lead to contractions in the perpendicular directions [6]. In DIANA, an orthotropic formulation is used to model the reduction of the Poisson's ratio. Furthermore, the stiffness of the concrete will decrease as the cracking increases. The cracked concrete should be modelled using an orthotropic material model, which the rotating and fixed crack model are examples of.

For a fixed crack model, shear behaviour has to be modelled as well [1]. As the concrete cracks, the shear stiffness will normally decrease. Therefore, a suitable shear retention model has to be chosen. A shear retention model based on the damage due to cracking has proven to be the most appropriate according to de Putter [4], when using a fixed crack model to model beams without stirrups. The damage based recalculation of the shear modulus, $G$, is based on the reduced Young's modulus and reduced Poisson's ratio [4]:

$$G^{cr} = \frac{E^{cr}}{2(1+\nu^{cr})} \tag{3.1}$$

### 3.3.3 Tensile behaviour

Concrete in tension experiences softening due to cracking, which is a nonlinear behaviour. Rijkswaterstaat recommends using an exponential-type diagram for the softening [1]. This is a better representation of the nonlinear behaviour, than for example a bilinear diagram. The preferred curves are the exponential softening curve, shown in Figure 3.5a, or the nonlinear softening curve according to Hordijk, shown in Figure 3.5b. Both tension softening curves are based on the fracture energy, $G_f$. The functions are related to the crack bandwidth, $h_{cr}$, as is common in a smeared crack model. Note that the ratio $G_f/h_{cr}$ determines the actual softening.

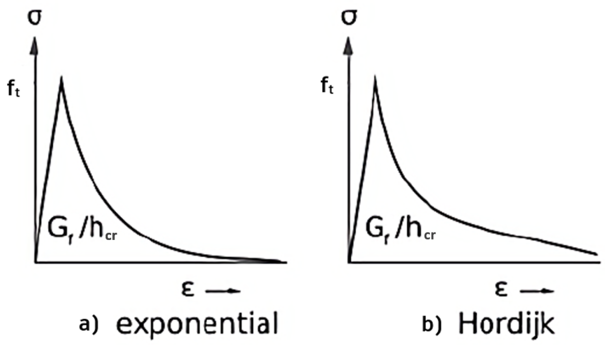



**Figure 3.5:** Tension softening curves of concrete [6]

#### 3.3.3.1 Fracture energy

In fracture mechanics one usually distinguishes between three different modes of fracture [20]: Mode I: opening (tension), Mode II: sliding (in-plane shear) and Mode III: tearing (out-of-plane shear). These modes are shown in Figure 3.6. Mode I is the main fracture form of concrete, it is the fracture mode of concrete subjected to tension. The fracture energy used for the tension softening curves is the mode-I tensile fracture energy, which can also be denoted $G_f^I$. The fracture energy is defined as the energy required to propagate a tensile crack of unit area [3].

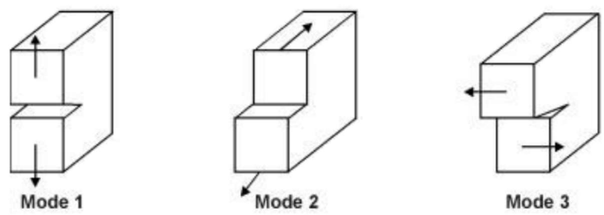



**Figure 3.6:** Fracture modes in fracture mechanics [20]

### 3.3.4 Compressive behaviour

The compressive behaviour of concrete is complex to model. For example does the boundary conditions to some extent influence the compressive behaviour [1]. Rijkswaterstaat recommends using a parabolic stress-strain diagram with a softening branch based on the compressive fracture energy, $G_c$, as shown in Figure 3.7 [1]. Similar to the tensile behaviour, the ratio $G_c/h$ is used to model the softening. The crushing bandwidth, $h$, is determined similarly to the crack bandwidth, $h_{cr}$. However, it should be based on the principal compression strain direction [1].

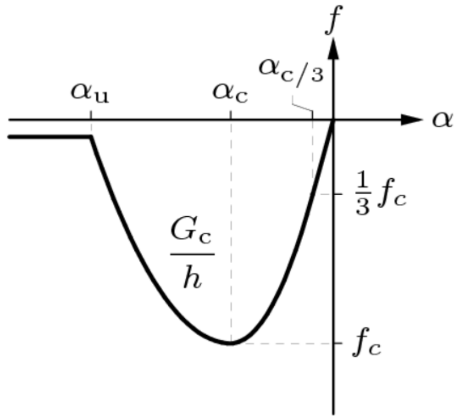



**Figure 3.7:** Parabolic compression curve [6]

The recommended parabolic compression curve uses three characteristic strain values, $\alpha$ [6]. The strain $\alpha_{c/3}$, which marks the point at which one-third of the maximum compressive strength, $f_c$, is reached, is defined as:

$$\alpha_{c/3} = -\frac{1}{3}\frac{f_c}{E} \tag{3.2}$$

$\alpha_c$ marks the point of the maximum compressive strength, and is defined as:

$$\alpha_c = -\frac{5}{3}\frac{f_c}{E} \tag{3.3}$$

$\alpha_u$ is the ultimate strain, at which point the curve is completely softened:

$$\alpha_u = \min(\alpha_c - \frac{3}{2}\frac{G_c}{hf_c}, 2.5\alpha_c) \tag{3.4}$$

Note that the fracture energy and the crushing bandwidth determine the softening part of the curve only [6], while $\alpha_{c/3}$ and $\alpha_c$ can be calculated independently of these parameters.

#### 3.3.4.1 Lateral influence on compression

Lateral confinement increases the compressive strength and ductility of the concrete, while lateral cracking reduces the compressive strength and ductility [4]. While the effects of lateral confinement can be ignored as a conservative assumption, the effects of lateral cracking have to be included. Even so, to model the nonlinear behaviour of concrete as accurately as possible, both effects should be modelled. For lateral cracking a minimum reduction factor of 0.4 is recommended [1], such that a minimum of 40 % of the strength remains. This is exemplified in Figure 3.8, which describes the relation for reduction due to lateral cracking described by Vecchio & Collins in 1993 [6].

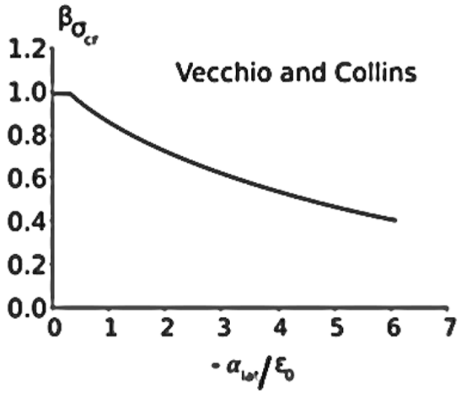



**Figure 3.8:** Vecchio & Collins reduction factor due to lateral cracking (1993) [6]

## 3.4 Constitutive model for reinforcement

For the nonlinear modelling of the reinforcement, an appropriate description of the yielding of the reinforcement is needed. Rijkswaterstaat recommends using an elasto-plastic material model with hardening for the reinforcement steel [1]. In DIANA, the standard Von Mises elasto-plastic model can be used to model the embedded reinforcement. The Von Mises yield function is given by the following formulation [6]:

$$f(\boldsymbol{\sigma}, \boldsymbol{\eta}, \kappa) = \sigma_v - \bar{\sigma}(\kappa), \tag{3.5}$$

where $\sigma\bar{(\kappa)}$ is the uniaxial yield strength, $\kappa$ is the internal state variable and $\eta$ is the back stress. For further information about these parameters, please refer to the Diana Documentation (section 54.3) [6]. The equivalent Von Mises stress, $\sigma_v$, is defined as follows for plane stress [21]:

$$\sigma_{\mathrm{v}} = \sqrt{\sigma_x^2 - \sigma_x \sigma_y + \sigma_y^2 + 3\tau_{xy}^2} \tag{3.6}$$

$\sigma_x$ is the x-component of the normal stress, $\sigma_y$ is the y-component of the normal stress and $\tau_{xy}$ is the shear stress. The relation between the internal state variable $\kappa$, seen in Equation 3.5, and the plastic process, is governed by the hardening hypothesis. DIANA offers two hardening hypotheses: strain hardening and work hardening. For more information about these hypotheses, please refer to the DIANA Documentation [6].

### 3.4.1 Hardening

The hardening can be approximated by a bilinear diagram. Optionally, rupture could be modelled by defining steep softening branches in the diagram. If rupture is not modelled, a post-analysis check is needed [1]. Figure 3.9 shows the suggested stress-strain curves in EC2 for normal design of the reinforcement, for both tension and compression. Curve A displays an idealised bilinear stress-strain diagram for the reinforcement steel, where the characteristic yield strength, $f_{yk}$, marks the onset of plastic deformation. The slope after this point is dependent on the hardening behaviour.

Mean or "measured" values for the tensile strengths can also be used to define the bilinear diagram.

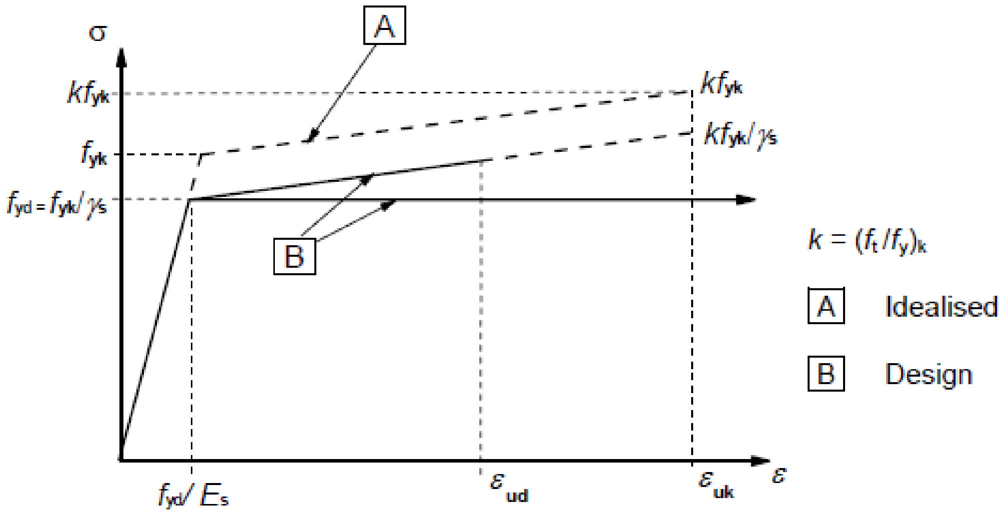



**Figure 3.9:** Idealised and design stress-strain diagram for reinforcement steel from EC2 [2]

If no hardening specifications are available, the minimum values for $k$ and the characteristic ultimate strain, $\varepsilon_{uk}$, can be taken from Annex C in EC2 [1]. The properties presented in Table 3.1 are retrieved from EC2, based on the reinforcement class.

**Table 3.1:** Excerpt from Table C.1 in EC2: Properties of reinforcement [2]

| Class | A | B | C |
|---|---|---|---|
| Characteristic yield strength $f_{yk}$ | 400 to 600 | | |
| Minimum value of $k = (f_t/f_y)_k$ | $\geqslant 1.05$ | $\geqslant 1.08$ | $\geqslant 1.15$ $< 1.35$ |
| Characteristic strain at maximum force, $\varepsilon_{uk}$ (%) | $\geqslant 2.5$ | $\geqslant 5.0$ | $\geqslant 7.5$ |

## 3.5 Concrete-reinforcement interaction

The concrete-reinforcement interaction is an important characteristic of reinforced concrete. While plain concrete has a low tensile strength and a more brittle behaviour, reinforced concrete has a higher tensile capacity and increased ductility thanks to the reinforcement. The bond between the concrete and the reinforcement allows for a redistribution of stresses between the concrete and the reinforcement after cracking occurs [22]. The redistribution continues gradually during the formation of secondary cracks, until a stabilised crack pattern has been developed [23]. In fact, the stiffness of the reinforced tensile member, when the crack pattern has stabilised, is higher than the stiffness of the reinforcement bar alone [1]. This effect is called tension stiffening, and is demonstrated in Figure 3.10. The stiffness of the uncracked concrete between the cracks contributes to the stiffness of the tensile member. Tension stiffening not only has a significant influence on the nonlinear behaviour of reinforced concrete under stress, but also contributes to an increased flexural strength, increased shear strength and increased stiffness of the RC structure [22].

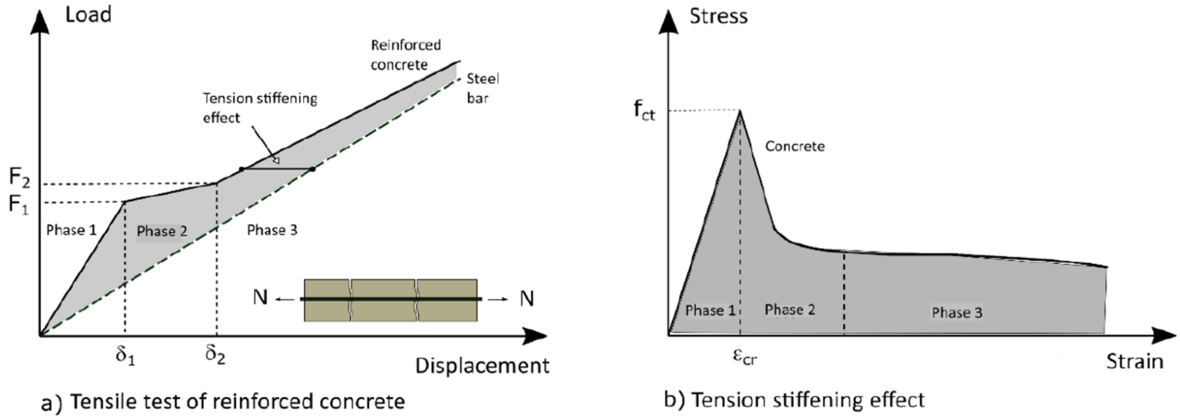



**Figure 3.10:** Tension stiffening [22] Phase 1: uncracked concrete, Phase 2: cracking of concrete (unstable), Phase 3: stabilised crack pattern

As can be seen from Figure 3.10b, the tension stiffening effect is at its maximum at the beginning of the cracking (Phase 2), and is then gradually reduced with the increasing displacement. Rijkswaterstaat declares that tension stiffening has to to be taken into account when modelling reinforced concrete, however with a small enough element size this is already accomplished by the tension softening curve [1], which is described in Section 3.3.3. The element size should be lower than the estimated average crack spacing, or else the fracture energy should be related to the crack spacing and element size [1]. EC2 provides guidelines for calculation of the crack spacing [2].

# Part II

# Modelling in DIANA and Python

# 4 General information

The created framework features a combination of Python commands and specific DIANA script commands. All operations in DIANA are logged as Python commands, and it is possible to run saved Python scripts to create and analyse models in DIANA. All the script commands specific for DIANA can be found in the DIANA Documentation - "Appendix B" [6]. By combining Python and DIANA, a framework can be created to generate the DIANA workflow. This workflow includes modelling the geometry, generating the mesh, running analyses and generating output. Hence, a more efficient, but still user-friendly way of modelling in DIANA than through the graphical interactive interface can be achieved. To improve the robustness of the NLFEA and make the modelling less dependent on the experience of the user, the framework contains default properties based on recommendations for NLFEA of RC structures. Object-oriented programming has been used to provide the default input.

This part will provide some general information about object-oriented programming in Python and how this has been used to structure the Beam Script. Moreover, certain aspects of modelling in DIANA will be explained. Part II is by no means intended to serve as a complete guide on how to use DIANA and Python. For more in detail information about modelling in DIANA and Python, please refer to the DIANA Documentation [6] and "Python.org". On the contrary, Part II is meant as a help for new users that wish to use the created framework to model in DIANA, as a selection of the obstacles faced as a new user when modelling in DIANA will be addressed.

Part II will be divided into two chapters: first certain aspects of modelling in DIANA will be explained, then modelling in Python using object-oriented programming will be discussed.

# 5 Modelling in DIANA

## 5.1 DIANA workflow

DIANA is a finite element analysis solver with strong focus on reinforced concrete analysis. The program has numerous elements, materials and solution procedure libraries, as well as a graphical interactive environment for pre- and post-processing. All operations in the Diana Interactive Environment (DianaIE) are logged as Python commands, and the DianaIE can be used by typing Python commands directly or by running saved Python scripts , as well as by using the graphical interactive interface [6].

The DIANA Documentation describes the preferable workflow in DianaIE as follows [6]:

1. Start a new project
2. Model the geometry
3. Generate the mesh
4. Define and run the analysis
5. Inspect the results
6. Generate a report

The Beam Script has been structured with a similar workflow, to ensure a smooth transition between working directly in the graphical interface and using the framework.

## 5.2 Importing Python modules in DIANA

Before using the framework for the first time, some Python modules have to be installed in DIANA. The needed modules are NumPy and Matplotlib. These modules enables amongst others working with arrays and graphical plotting. The modules can be installed in the following manner [23]:

1. Open the DIANA Command Box (via the start menu on your computer)
2. Choose "Run as administrator" (*Kjør som administrator*)
3. Paste the following lines into the command box:

   ```
   %DIAPATH_W%\python\python -m pip install -t %DIAPATH_W%\modules numpy

   %DIAPATH_W%\python\python -m pip install -t %DIAPATH_W%\modules matplotlib
   ```

In general, the following line can be used to install third party Python modules in DIANA:

```
%DIAPATH_W%\python\python -m pip install -t %DIAPATH_W%\modules <name of module>
```

Figure 5.1 and Figure 5.2 show how this would look like using the Windows 11 operating system.

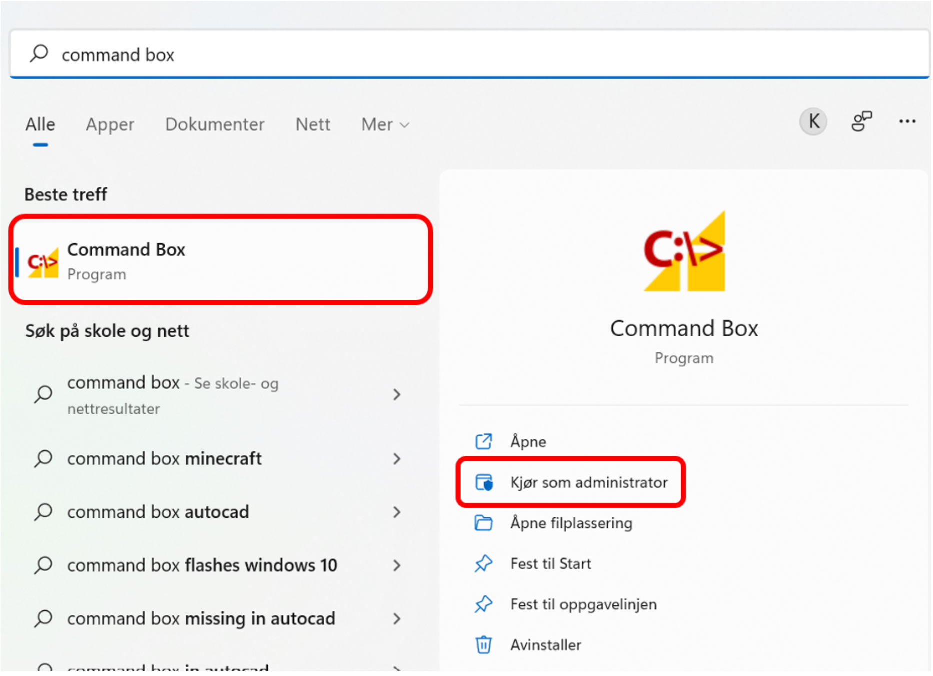



**Figure 5.1:** Running DIANA Command Box as administrator

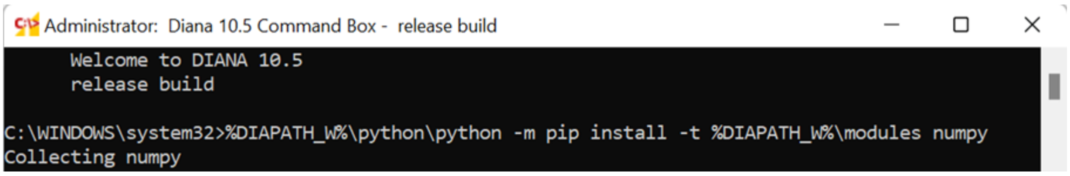



**Figure 5.2:** DIANA 10.5 Command Box

## 5.3 Structural interfaces

Connections in DIANA are used to define interaction between the boundaries of different shapes. A structural interface is a type of connection. The material model for the structural interface defines the normal and shear relation between the tractions (i.e. stresses) and relative displacements across the boundary [6]. This can be a linear or a nonlinear relation. In the Beam Script, 2D-line interfaces have been used to define the interactions between the concrete beam and the steel plates. The linear constitutive relation between the tractions and the relative displacements for a 2D line interface is shown in Equation 5.1 [6]. The nonlinear relation of the interface can be specified using a diagram or reduction functions.

$$\begin{Bmatrix} t_n \\ t_t \end{Bmatrix} = \begin{bmatrix} k_n & 0 \\ 0 & k_t \end{bmatrix} \begin{Bmatrix} \Delta u_n \\ \Delta u_t \end{Bmatrix} \tag{5.1}$$

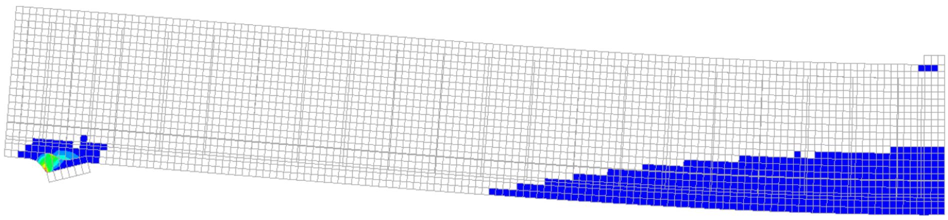
The traction vector, $\mathbf{t}$, consists of the normal traction, $t_n$, and the shear traction, $t_t$. The relative displacement vector, $\mathbf{\Delta u}$, consists of the normal relative displacement, $\Delta u_n$, and the shear relative displacement, $\Delta u_t$. The normal stiffness is denoted $k_n$ and the shear stiffness is denoted $k_t$.

For the finite element model, loads and supports are applied using load and support plates. These plates have to be defined in such a way that spurious local stress concentrations, which can cause local failure in the plates, are avoided [1]. This can be done using a no-tension/no-friction interface between the beam and the plates. According to the guidelines by Rijkswaterstaat:

> The compressive interface stiffness [i.e. the normal stiffness in compression] should be set relatively high, e.g. 1000 times more stiff than a neighbouring concrete element: 1000 $E_c/h$, in which $h$ is the size of the neighbouring concrete element. The interface shear stiffness should be set relatively low. [1]

Figure 5.3 demonstrates the importance of using interfaces between the steel plates and concrete beam. Figure 5.3 displays the crack pattern at failure of Case B1, which is one of the benchmark studies presented in Chapter 10, with and without the use of structural interfaces. The only difference between the two models is whether or not structural interfaces have been applied. As can be seen in Figure 5.3a, the stresses localise around the support plate when no interface has been created, and the beam fails due to local failure near the plate. Hence, the expected crack pattern which can be seen in Figure 5.3b is not achieved, and the experimental setup of the beam and its boundary conditions are not modelled correctly.

a)



b)

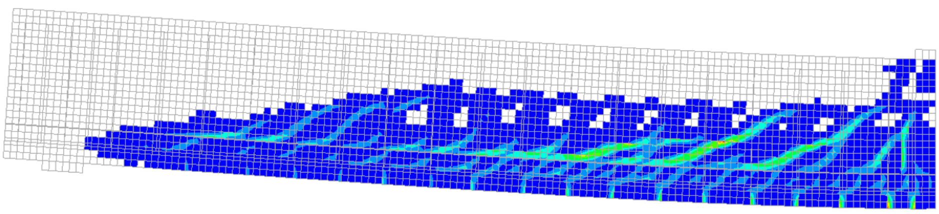


**Figure 5.3:** Failure crack pattern of Case B1 a) without structural interfaces b) with structural interfaces

## 5.4 Load step execution

Loads on the structure should be applied according to the specifications in the Eurocode. Dead weight and permanent loads should be applied first, as a separate initial load case [1]. Then live loads can be applied, in sequential load steps. The command for initiating a structural nonlinear analysis in DIANA is denoted `NONLIN`, and can contain several `EXECUT`-blocks. The `EXECUT`-command in DIANA specifies in which order to execute the load steps. Each of these `EXECUT`-blocks could again be divided into load increments and substeps. In the Beam Script, the first `EXECUT`-block corresponds to the self-weight of the concrete beam, while the second `EXECUT`-block corresponds to the applied point loads. The `EXECUT`-blocks used in the command sequence for application `NONLIN` in the Beam Script are presented in Script 5.1. The dead-weight loading is applied in one load step, with the step size explicitly set to 1 for this `EXECUT`-block. The load steps for the second `EXECUT`-block, containing the point loads, are automatically determined by DIANA using adaptive load increments. The default maximum number of steps for the second `EXECUT`-block is set to 150 load steps. For further information about the properties of the NLFEA, see Section 8.7.2.

```
addAnalysis( analysis.name )
addAnalysisCommand( analysis.name, "NONLIN", analysis.command )

#Dead weight
#Applied in one increment
renameAnalysisCommandDetail(analysis.name, analysis.command,"EXECUT(1)",selfweight.name)
addAnalysisCommandDetail(analysis.name, analysis.command,"EXECUT(1)/LOAD/LOADNR" )
setAnalysisCommandDetail(analysis.name, analysis.command,"EXECUT(1)/LOAD/LOADNR", 1 )

#Point loads
setAnalysisCommandDetail(analysis.name, analysis.command, "EXECUT/EXETYP", "LOAD" )
renameAnalysisCommandDetail(analysis.name, analysis.command, "EXECUT(2)", "Loads" )
addAnalysisCommandDetail(analysis.name, analysis.command, "EXECUT(2)/LOAD/LOADNR" )
setAnalysisCommandDetail(analysis.name, analysis.command, "EXECUT(2)/LOAD/LOADNR", 2)
```

**Script 5.1:** Creating execute blocks for NLFEA

# 6 Modelling in Python

## 6.1 Object-oriented programming in Python

One of the main motivations behind creating the Beam Script, was to create a framework with easily changeable input parameters and default properties based on recommendations. Object-oriented programming has been utilised to provide these features and to structure the Beam Script.

Object-oriented programming is a way of structuring a script. Sets of properties and characteristics be can created and related to individual objects. For example, all parameters relating to the reinforcement, like the yielding strength, cross section area and the Youngs's modulus of the reinforcement steel, have been bundled together with the use of object-oriented programming. The first step when it comes to this way of structuring a script is to define classes, from which individual objects with the same properties can be created. The Beam Script consists of four scripts, where Script C has been used to define the classes. A class works as a blueprint for creating actual objects [24]. In Script C, classes have been created for amongst others the beam, the reinforcement, the loads and the different material models. Default properties have been defined for the different classes. This makes sure that for example all steel plates will have the same material properties and geometry, without having to define this for each individual plate. Script 6.1, which is an excerpt from Script C, shows the class definition of the material model for the steel plates. By default, all created plate objects will have the material properties listed in line 3-5 of Script 6.1.

```
1 class Steel_Plates(): #Material model for steel plates
2     #A linear-elastic behaviour is assumed
3     material = "MCSTEL"
4     material_model = "ISOTRO"
5     density = 0 #Do not wish to take the weight of the plates into concideration
```

**Script 6.1:** Class of material model for steel plates

Script 6.2 shows a simplified example of the class Plates and how to create objects from this class. The actual class Plates in Script C has a more complex setup than the class definition shown in Script 6.2. By default, the translations of the support plates are fixed in both x- and y-direction, as can be seen from line 4 - 5 of Script 6.2. Line 7 - 8 of Script 6.2 demonstrate how to create objects from the class Plates. By default, the created objects, named `sup_plate1` and `sup_plate2`, will have the same attributes, as defined in the class Plates. The different attributes can be accessed and changed using the dot (.) notation. For example can translation in x-direction be permitted for `sup_plate1`, by setting the attribute `fixedTranslation_x` to `False` using the dot notation. This is done in line 10 of Script 6.2. Even though the translation in x-direction will no longer be fixed for `sup_plate1`, the translation in x-direction will still be fixed for `sup_plate2`, as the change only applies to the specified object before the dot notation.

```
1 class Plates():
2     allObjects = []
3     y_coord = 0 #default value
4     fixedTranslation_x = True
5     fixedTranslation_y = True
6
```

```
sup_plate1= Plates()
sup_plate2= Plates()

sup_plate1fixedTranslation_x = False
```

**Script 6.2:** Simplified example of class Plates

Script 6.3 displays the class Concrete from Script C, which is used to create the concrete material model. A number of default recommended properties for the material model are listed in line 2 - 15 of Script 6.3. The user has the option to change these properties in Script A of the Beam Script. However, the listed properties do not have to be defined or input by the user, as they are given by default when creating an object for the concrete material. Line 17 of Script 6.3 shows the method `.__init__()`, which is a method of defining the properties that must be given by the user when creating the concrete material. As can be seen from line 17 in Script 6.3, the only required input parameter for creating the concrete material model is the characteristic cylinder compressive strength, $f_{ck}$.

```
1  class Concrete():
2      #Material Properties:
3      Name = "Concrete" #Name of the material
4      material = "CONCR"
5      aggregate_type = "QUARTZ"
6      #Cement type is set to N
7      material_model = "TSCR" #Total strain based cracking.
8      crack_orientation = "ROTATE" #will be changed to FIXED for beams without stirrups.
9      tensile_curve = "EXPONE"
10     crackBandwidt_specification = "GOVIND"
11     poissonsRatio_reductionModel = "DAMAGE" #Poisson's ratio reduction, damage based
12     compression_curve = "PARABO"
13     lateralCracking_reductionModel = "VC1993" #Reduction curve due to lateral cracking
14     lateralCracking_reductionCurve_lowerBound = 0.4 #lower bound of reduction curve
15     confinementModel = "VECCHI" #Stress confinement model, "NONE" is conservative
16
17     def __init__(self, _f_ck):
18         self.f_ck = _f_ck
19         T31_f_ck = np.array([12,16,20,25,30,35,40,45,50,55,60,70,80,90])
20         T31_f_ck_cube = np.array([15,20,25,30,37,45,50,55,60,67,75,85,95,105])
21         self.f_ck_cube = np.interp(_f_ck,T31_f_ck,T31_f_ck_cube)
22         self.EC2 = "C" + str(int(self.f_ck)) + "/" + str(int(self.f_ck_cube))
23         self.fib = "C" + str(int(self.f_ck))
```

**Script 6.3:** Class for material model of concrete

## 6.2 List of objects

The object-oriented structure allows the user to create an unlimited and unspecified number of objects with user defined names. Through the class definitions in Script C, each object that is created from a class, is attached to a list of all the objects created from that class. In Script D of the Beam Script, this list is iterated through to create the different objects in DIANA. An example of how a list for all objects of a class is initalized, is shown in Script 6.4. The class Plates is defined in the first line, and the list `allObjects` is defined in line 2. The list `allObjects` is initially empty, however each object that is created of that class is appended to the list (line 7). After the plate object named `VeryNiceLoadingPlate` is created in line 14, the list `allObjects` contains one element.

```python
class Plates():
    allObjects = []
    y_coord = 0 #default value
    fixedTranslation_x = True
    fixedTranslation_y = True
    def __init__(self, _typeOfPlate, _name, _x_coord):
        self.allObjects.append(self)
        self.typeOfPlate = _typeOfPlate
        self.name = _name
        self.x_coord = _x_coord
    def add_geometry(self, _geometry):
        self.geometry = _geometry

VeryNiceLoadingPlate = Plates("L", "Loading plate 1", 70)
```

**Script 6.4:** Definition of the class Plates

# Part III

# The Beam Script

# 7 General information

A framework for creating and analysing 2D finite element models of reinforced concrete beams has been created. The framework consists of four scripts, which will be referred to as the Beam Script. All four scripts can all be found in the Appendix:

- A: User input
- B: Parametric study
- C: Classes and Functions
- D: Main script

A DIANA project can be created by running a Python script directly, and is the way in which the Beam Script has to be executed. To perform a single analysis of a user-defined RC beam, Script D has to be run in DIANA. To perform a parametric study of one of the input parameters, Script B has to be run in DIANA. All scripts have to be saved in the same working folder for DIANA to be able to access them when running the Beam Script. A saved Python script can be run in the DianaIE in the following manner:

| **Main menu** → File → Run saved script |
|---|

While the Beam Script consists of four parts, only Script A, which contains the user input, needs to be changed significantly by the user. Here, the geometry of the beam has to be defined, as well as the material properties and arguments for the NLFEA. Default parameters and choices are already in place, such that suitable material models and solution strategies will be applied by default. To perform a parametric study, Script B also has to be changed. The parametric values have to be input in Script B, this is further explained in Section 7.7. Script C and D should not be changed by the user.

To make the input script intuitive, the structure and workflow are very similar to the that of the DIANA graphical interface. The user starts by defining the geometry and ends with defining the output of the analysis, before running the Beam Script in DIANA. Script A is structured in the following manner:

1. Info
2. Creating the project
3. Creating the geometry
4. Loads
5. Material models
6. Structural interfaces
7. Mesh
8. Analyses
9. Output from NLFEA

## 7.1 Units

The finite element analysis itself has no concept of units, and it is therefore important to be mindful of using a consistent set of units to get the correct output. The framework uses millimetres, and the corresponding unit system is found in Table 7.1. For a correct analysis, it is important that the user is aware of the chosen units, and provides an input in accordance with the selected unit system.

**Table 7.1:** Unit system

| **Entity** | **Unit** |
|---|---|
| Length | Millimetres [mm] |
| Mass | Ton [t] |
| Time | Seconds [s] |
| Temperature | Celsius [°C] |

## 7.2 Coordinate system

The coordinate system is defined with its origin located in the lower left corner of the RC beam, as shown in Figure 7.1. In other words, the lower left corner of the beam has the coordinates [x,y] = [0,0]. Positive x-axis is defined to the right, and positive y-axis is defined upwards. Consequently, positive loads will act in the positive directions of the coordinate system. When creating the loading plates and support plates, the x-coordinate might be required. The x-coordinate of a plate refers to the midpoint of the plate, where the support or load is attached, as shown in Figure 7.1. In general, coordinates might be asked for in Script A, in order to specify the position of different objects. These must be given in accordance with the specific coordinate system.

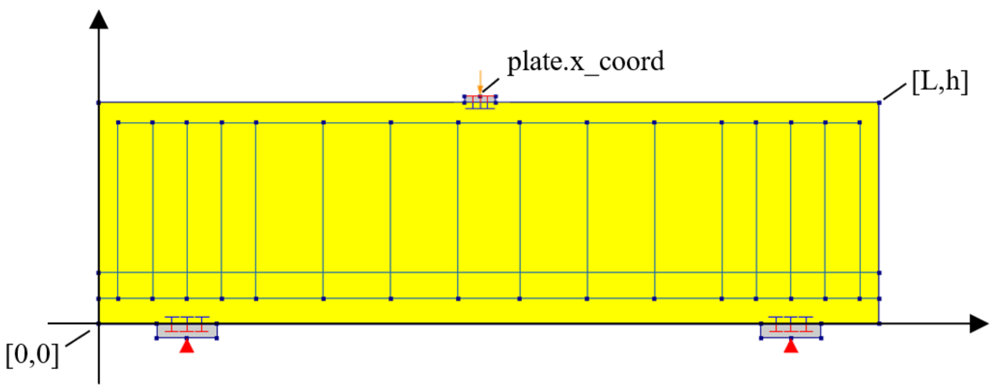



**Figure 7.1:** Coordinate system for Beam Script

## 7.3 Input parameters

Script A is structured in such a way that if the user chooses to run the Beam Script without making any changes, this can be done. A very simple beam without reinforcement is then created, which is shown in Figure 7.2. This beam uses the template for a three-point bending test, which is further explained in Section 7.6.

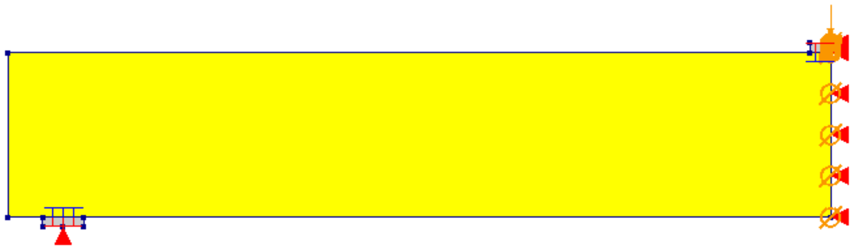


**Figure 7.2:** Default beam: View of model

The input parameters are divided into optional and required parameters. Even so, as default values and properties are provided, no user input are in actuality needed for the Beam Script to run. The required parameters are amongst others related to the geometry of the concrete beam, the loading, the concrete class and the incremental procedure. The default values for creating the beam geometry have been chosen at random, and will have to be changed by the user to create the desired beam. The other default properties, however, are based on experience or guidelines. The optional parameters have recommended values as default, but can be changed by the used if desired. However, the user is in general recommended to be careful when changing the input script. A useful principle is to not change more parameters than necessary, when running the Beam Script and the nonlinear analysis for the first time.

Script 7.1 shows an example of how different parameters are defined in Script A. The optional parameters, line 5 - 7, are commented and marked with **Optional**. The required parameters, line 1 - 3, have default values which can be changed by the user. If the user wishes to include an optional parameter, this line has to be uncommented and given an input value. In general, lines that are not relevant for the desired beam should stay commented (ctrl + k), while relevant lines should be uncommented (ctrl + shift + k).

```
1 beam.geometry.length = 6000 #[mm]
2 beam.geometry.height = 600 #[mm]
3 beam.geometry.width = 400 #[mm] #thickness
4
5 # concrete.poissons_ratio = #**Optional**
6 # concrete.mass_density = #**Optional**
7 # concrete.tensile_strength = #**Optional**
```

**Script 7.1:** Required and optional parameters

Some sections of the input script provide different options, where one of the options has to be chosen. Other sections are optional in their entirety. This is clearly specified in the script. If an option is chosen, this section should be uncommented, while the not chosen options should be commented. For example does section `5.5 Plates` in Script A provide three options for creating the plates, where one option has to be chosen. Section `5.2.1 Cover` in Script A on the other hand is an optional section, where neither of the given options has to be chosen. An excerpt of section `5.2.1 Cover` is shown in Script 7.2. Note how the whole section is marked as optional in line 1. As can be seen by the uncommented lines, Option 1 has been selected.

```
#'#5.2.1 Cover: #**Optional input**
##Cover is an optional object, which can be used for your convinience:
cover = Cover() #creating cover object
##Choose one of the options.
##The section with the chosen option should be uncommented,
##while the other options should be commented.

####**Option 1**: definition of each cover
cover.side = 48 #[mm]
cover.top = 50 #[mm]
cover.bot = 64 #[mm]

# ####**Option 2**: all sides have same cover:
# cover.length = 100 #[mm]
# cover.side = cover.top = cover.bot = cover.length
```

**Script 7.2:** Optional section for defining cover

## 7.4 Symmetry

A symmetry option is added to the input script, as shown in Script 7.3. If the symmetry option is chosen, only half the beam will be modelled. Modelling only half of the beam is a useful time-saving strategy, and should be employed when possible. However, the user has to be certain that not only the geometry, but also the expected failure mode is symmetric when choosing this option.

```
## If the following option is chosen, only half the beam will be modelled:
Options.symmetric_Beam =  True #**Optional**, default is False
```

**Script 7.3:** Symmetry option

Modelling a symmetric beam is done by constraining the symmetry plane in x-direction as well as constraining the rotations. If a load is applied at midspan, as can be seen in Figure 7.2, the value of this load would equal half of the applied load on the whole beam. Furthermore, if the coordinates of a transverse reinforcement bar corresponds to the coordinates of the symmetry plane, the area of this bar will automatically be set to half the area of the bar. Hence, with reference to the whole beam, the bar will have the correct cross section area and not double the area.

## 7.5 Templates

Several empty templates have been added to the input script, to simplify the creation of different objects. The code inside the empty template can be copied and filled with the desired input. Example blocks have also been added to demonstrate how the templates can be used to create objects. Script 7.4 shows the example block and template for creating longitudinal reinforcement in Script A.

```
##Examples of the creation of longitudinal reinforcement bars:
'''Example block (can be copied and changed to create your own objects)
#
upper_reinfo = Longitudinal_Reinfo("upper", 300, 315, 460, 2.3425E-2)
upper_reinfo.y_coord = beam.geometry.height - cover.top
```

```
upper_reinfo.x_coord_start = cover.side
# upper_reinfo.x_coord_end = beam.geometry.length - cover.side #**If not symmetric**

#
ReinfoLow = Longitudinal_Reinfo("lower", 1400, 436, 700, 4.78E-2, 220000)
ReinfoLow.y_coord = 20
ReinfoLow.x_coord_start = 0
# ReinfoLow.x_coord_end = beam.geometry.length #**If not symmetric**
'''

##Copy this block to create long.reinfo objects:
''' Empty template
templateName = Longitudinal_Reinfo( )
templateName.y_coord =
templateName.x_coord_start =
# templateName.x_coord_end = #**Input needed if beam is not symmetric**
'''
```

**Script 7.4:** Example block and empty template for longitudinal reinforcement

As can be seen from the example block of Script 7.4, the content of the template can be copied to create an object. In case of Script 7.4, the object is a longitudinal reinforcement bar. Unless otherwise stated, the user can create as many objects as desired, including zero. When an empty template is used to create an objects of a class, the user is free to choose the name of the object. This allows the user to create object names that makes it easier to remember for example which reinforcement bar one is editing, as can be seen in the example block of Script 7.4. To easily substitute the template names with a user-defined name, a replace box (crtl + h) can be used.

## 7.6 Templates for bending tests

Two templates have been created, to simplify the workload when creating a beam subjected to a symmetric three-point or four-point bending test. These tests are commonly used to determine the flexural strength (bending strength) of a specimen. Script 7.5 shows how these templates can be selected in Script A. As can be seen from the uncommented lines of Script 7.5, using the template for a three-point bending test has been chosen.

```
#'#5.5.1 Option 1: using template for 3-point bending
Options.template_3PointBending = True
beam.geometry.lengthOfSpan = beam.geometry.length - 400 #[mm]

#'#5.5.2 Option 2: using template for 4-point bending
# Options.template_4PointBending = True
# beam.geometry.lengthOfSpan = 5000 #[mm]
# beam.geometry.lengthOfInnerSpan = 3000 #[mm]
```

**Script 7.5:** Bending test options

The beam setups of the bending test templates are presented in Figure 7.3. The beams are simply supported, and the span lengths determine the placement of the supports and point loads. For the three-point bending test, the point load, F, is placed at the midpoint of the beam.

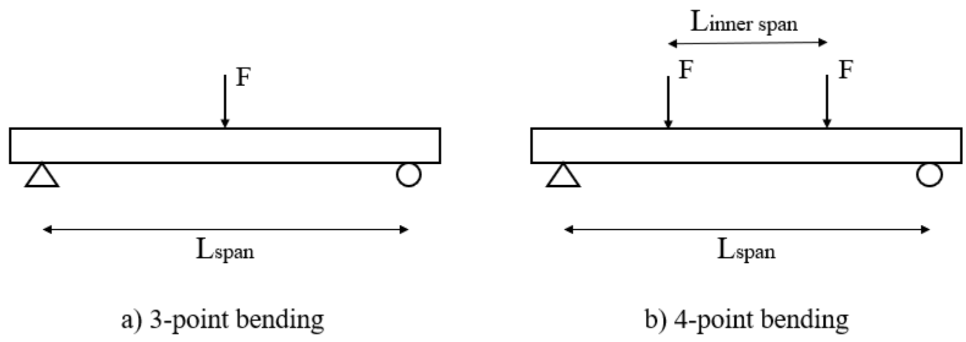



**Figure 7.3:** Setups of bending test templates

As can be seen in Figure 7.3b, both loads are assumed to have the same magnitude and direction for the four-point bending test. Therefore, only one load has to be defined in the input script. Script 7.6 shows how the load, F, is defined in Script A, when using a bending test template. As can be seen in line 3, the default initial value of the point load(s) is 1000 N = 1 kN.

```
1 #-#6.1 Option 1: using template for 3- or 4-point bending
2 ##Both loads are assumed to have same value for 4-point bending.
3 ##Only one has to be defined.
4 load = Loads("Point", "Load", -1000, "Y")
```

**Script 7.6:** Load input for bending test templates

Since the setups of the bending test templates are symmetric, only half of the beam will be modelled in DIANA when using one of these templates. This will happen even though the option of symmetry is not selected (`Options.symmetricBeam = False`), as the template option overrules the symmetry option. Examples of the models created in DIANA when using the bending test templates are shown in Figure 7.4 and Figure 7.5. As can be seen in the figures, symmetry supports which restrict the translation in x-direction as well as rotations, are attached at the symmetry edge. Furthermore, the translation in y-direction is fixed for the support plate. The x-direction is not constrained, to avoid unjustified modelling of arching effects [4].

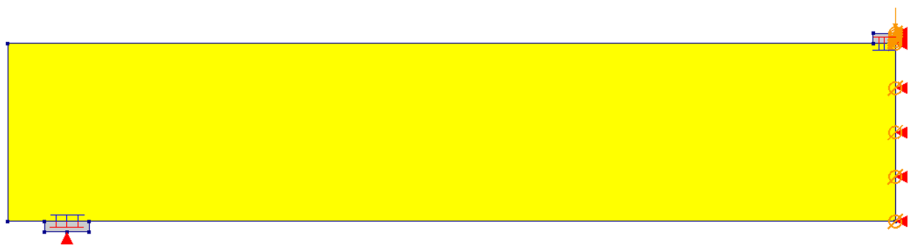


**Figure 7.4:** Model of three-point bending test

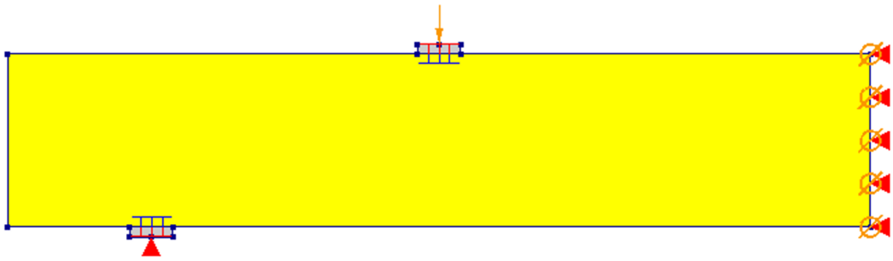


**Figure 7.5:** Model of four-point bending test

## 7.7 Parametric study

The framework also allows for a parametric study to be performed. If a parametric study is chosen, multiple analyses will be executed. Per usual, the user has to use Script A to define the user input. While one of the parameters will be varied for the parametric study, the rest of the user defined parameters will stay the same for all analyses. The parametric study is performed by editing Script B: Parametric study, as well as Script A: User input. In Script B, the input for the varying parameter is defined. In Script A, the varying parameter has to be marked with the input `parametricValue` and the rest of the parameters should be defines as per usual. Figure 7.6 shows an excerpt of Script B. The only line that should be changed in Script B is line 18. Here, a list of values to be used for the parameter to be varied in the parametric study is defined. In Figure 7.6, the chosen values are 3000 and 4000. There is no upper limit for the number of values that can be added to the list. However, as each value will generate a new project in DIANA, a larger number of values will result in a more time-consuming parametric study.

```
1   def parametricStudy(valuesList):
16
17  ##INPUT LIST OF PARAMETRIC VALUES HERE:
18  parametricStudy([3000,4000])
```

**Figure 7.6:** Excerpt of Script B: Parametric study

Script 7.7 shows how the parameter to be varied for the parametric study should be marked in Script A. Only one parameter should be marked as `parametricValue`.

```
beam.geometry.length = parametricValue #[mm]
beam.geometry.height = 600 #[mm]
beam.geometry.width = 400 #[mm] #thickness
```

**Script 7.7:** Defining the parameter to be varied in the input script

The user has to make sure that other parameters that depend on the chosen varying parameter are defined in a way that reflects this dependence. For example does the length of the span of the beam most likely depend on the length of the beam. Hence, if the beam length is to be varied, the span length has to be defined accordingly. An example of this is shown in Script 7.8. Line 5 demonstrates a good way of defining the length of the span, which reflects the dependence. Line 6, on the other hand, would result in a span length independent of the varying length of the beam.

```
beam.geometry.length = parametricValue #[mm]

#'#5.5.1 Option 1: using template for 3-point bending
Options.template_3PointBending = True
beam.geometry.lengthOfSpan = beam.geometry.length - 400 #[mm]
#beam.geometry.lengthOfSpan = 3000 #[mm]
```

**Script 7.8:** Defining parameters depending on the varying parameter

Due to the possible dependencies between parameters, it is recommended to first generate the parametric models, or some of the models if a large number is to be modelled, without running any analyses. This is a quick process, compared to running the nonlinear analyses, and the user can easily verify that the geometry is modelled as expected. A linear analysis should also be performed for at least one of the models, to check that the finite element model has the expected behaviour. Finally, a parametric study that includes the nonlinear analysis can be performed. After defining the user input and the values for the parametric study, the user can simply run Script B: Parametric study in DIANA. From this point, the user no longer have to interact with the program. The creation of the projects, the analyses and the generation of the outputs will be carried out automatically.

# 8 Script A: User input

## 8.1 Creating the project

Before running the Beam script for the first time, some Python modules have to be installed in DIANA. How to install these are explained in Section 5.2.

By default, the generated DIANA project is saved in the same folder as the Beam Script. As stated earlier, all four scripts have to be saved in the same working folder for DIANA to be able to run the Beam Script. Optionally, if the user wishes to save the project someplace else, the directory path can be specified by the user. When the project is saved, a new folder is created in the specified directory with the name of the model. In this folder, the DIANA files as well as the user selected output will be saved. The required input for creating the project is the name of the model. By default the name of the DIANA project is set to:
`Project.name = Project.modelName + "_" + TodaysDate + "_" + Project.modelExtraInfo`
If the main script is run without changing the user input, the project will be saved as:
`TestBeam_yyyymmdd_default`
For a parametric study, the value of the varying parameter denoted `parametricValue` in Script A, will be added to the end of the project name.

## 8.2 Creating the geometry

### 8.2.1 Beam

The geometry of the beam is created by defining the length, height and width of the beam, as shown in Script 8.1. Only line 8 - 10 require input. Line 4 - 6 should not be changed by the user, and is used to create the beam object and attaching the geometry object to the beam object. For further details on the object oriented structure of the Beam Script, see Chapter 6.

```
1  ###5. CREATING THE GEOMETRY
2  #-#5.1 The beam
3  #'#5.1.1 Geometry:
4  beam = Beam() #creating beam object
5  beamGeometry = Geometry() #creating geometry object
6  beam.add_geometry(beamGeometry) #adding the geometry to the beam
7
8  beam.geometry.length = 6000 #[mm]
9  beam.geometry.height = 600 #[mm]
10 beam.geometry.width = 400 #[mm] #thickness
```

**Script 8.1:** Creating the beam geometry

### 8.2.2 Reinforcement

In Script A, the creation of the reinforcement is divided into two parts: creation of the longitudinal reinforcement and creation of the transverse (shear) reinforcement. Furthermore, an optional section for defining the cover of the RC beam is provided, as well as a section where the user can define new parameters to simplify the creation of the reinforcement. Defining a parameter for the diameter of the reinforcement is suggested.

#### 8.2.2.1 Cover

The optional section of Script A for defining the concrete cover is shown in Script 8.2.

```
#'#5.2.1 Cover: #**Optional input**
##Cover is an optional object, which can be used for your convinience:
cover = Cover() #creating cover object
##Choose one of the options.
##The section with the chosen option should be uncommented
##while the other options should be commented.

# ##**Option 1**: definition of each cover
# cover.side = 48 #[mm]
# cover.top = 50 #[mm]
# cover.bot = 64 #[mm]

##**Option 2**: all sides have same cover:
cover.length = 100 #[mm]
cover.side = cover.top = cover.bot = cover.length
```

**Script 8.2:** Creating the cover

Three different concrete covers can be defined: concrete cover for the top of the beam, concrete cover for the bottom of the beam and concrete cover for the side of the beam. Option 1 of Script 8.2 allows for these covers to have different values, while Option 2 can be chosen if all covers have the same size. The different covers are shown in Figure 8.1. Defining the cover is useful when the coordinates of the reinforcement are to be decided.

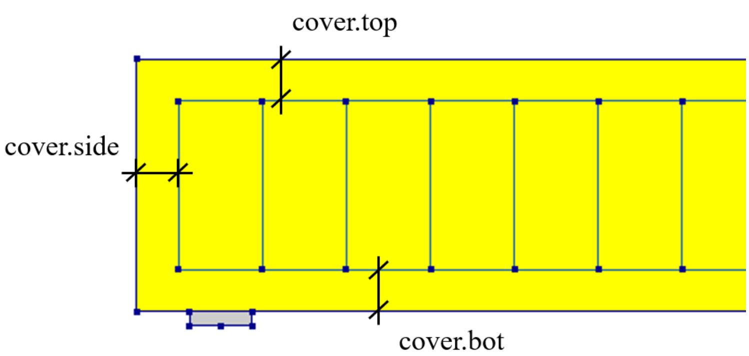



**Figure 8.1:** Definition of cover

#### 8.2.2.2 Longitudinal reinforcement

The longitudinal reinforcement can be created by defining objects of the class `Longitudinal_Reinfo`. Each object corresponds to a reinforcement bar with specified x- and y-coordinates. Since the dimension of the model is 2D (XY plane), the area of the reinforcement should be defined as equivalent to the cross section area of the total number of bars given at the in-plane coordinates over the whole out-of-plane width of the beam. With reference to Figure 8.2, the lower longitudinal reinforcement should for example be defined with a cross section area equivalent to the area of all three bars.

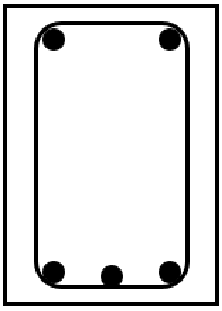


**Figure 8.2:** Cross section of RC beam

The longitudinal reinforcement is created as shown in Script 8.3. Line 1 explains the input that is needed when creating a longitudinal reinforcement bar. `nameOfBar` can be chosen by the user. `Name` is a user defined name of the bar, and is the name the geometry object will be given in DIANA. `As` is the cross section area. `F_y` is the yield strength, `F_u` is the ultimate strength and `Epsilon_u` is the ultimate strain of the reinforcement steel. A default value of 200 000 MPa will be used for the Young's modulus, if no value is given when defining the reinforcement. Furthermore, the x- and y-coordinates of the reinforcement bar have to be defined. If the beam is not symmetric, the x-coordinates of both the starting point and the end point of the longitudinal reinforcement bar have to be entered. Line 5 - 8 in Script 8.3 show an example of how to create a longitudinal reinforcement object.

```
1 #nameOfBar = Longitudinal_Reinfo(Name, As, F_y, F_u, Epsilon_u, (E_mod))
2
3 #'#5.3.1 Creation of longitudinal reinforcement
4 ##CREATE YOUR LONG.REINFO OBJECTS HERE
5 upper_reinfo = Longitudinal_Reinfo("upper", 300, 315, 460, 0.02)
6 upper_reinfo.y_coord = beam.geometry.height - cover.top
7 upper_reinfo.x_coord_start = cover.side
8 upper_reinfo.x_coord_end = beam.geometry.length - cover.side #**If not symmetric**
```

**Script 8.3:** Creating the longitudinal reinforcement

An example block and an empty template are provided in Script A to simplify the process of creating the longitudinal reinforcement. The content of the template can be copied and pasted as many times as desired by the user. Script 8.4 displays an excerpt of section `5.3 Longitudinal reinforcement` in Script A. The reinforcement objects should be created below the line `##CREATE YOUR LONG.REINFO OBJECTS HERE`, as shown in Script 8.3.

```
##Copy this block to create long.reinfo objects:
''' Empty template
templateName = Longitudinal_Reinfo( )
templateName.y_coord =
templateName.x_coord_start =
# templateName.x_coord_end = #**Input needed if beam is not symmetric**
'''

```

```
#'#5.3.1 Creation of longitudinal reinforcement
##CREATE YOUR LONG.REINFO OBJECTS HERE
```

**Script 8.4:** Template for creating the longitudinal reinforcement

#### 8.2.2.3 Transverse reinforcement

The Beam Script is designed such that all the transverse reinforcement, also known as the shear reinforcement, will have the same material properties. Therefore, only one object should be created for the transverse reinforcement, if this type of reinforcement is desired. The class `Transverse_Reinfo` is used to create the transverse reinforcement object. The object is created as shown in Script 8.5. The input parameters for creating the reinforcement object (line 1), are the same as the input parameters for creating the longitudinal reinforcement, which is explained in Section 8.2.2.2. A default value of 200 000 MPa will be used for the Young's modulus, if no value is given when defining the transverse reinforcement. The y-coordinates of the transverse reinforcement bars also have to be defined. All bars will have the same y-coordinates. Similarly to the creation of the longitudinal reinforcement, an example block and an empty template are provided in Script A for the creation of the transverse reinforcement.

```
1 ##nameOfBar = Transverse_Reinfo(Name, As, F_ym, F_um, Epsilon_u, (E_mod))
2
3 #'#5.4.1 Creation of transverse reinforcement
4 ##CREATE YOUR TRANS. REINFO OBJECTS HERE
5 shear_reinfo = Transverse_Reinfo("Shear_reinfo", 25.7, 600, 651, 0.05, 220000)
6 shear_reinfo.y_coord_bot = cover.bot
7 shear_reinfo.y_coord_top = beam.geometry.height - cover.top
```

**Script 8.5:** Creating the transverse reinforcement

In section `5.4.2 Spacing` of Script A, the user can define the spacing/positions of the transverse reinforcement bars. The user can define one or multiple sections with different spacing. If transverse reinforcement is to be included, at least one section with a given spacing has to be defined. For a symmetric beam, this includes the bending test templates, sections only have to be defined for half of the beam. Script 8.6 shows how sections for the transverse reinforcement are created, as well as the example block from section `5.4.2 Spacing`. Line 1 in Script 8.6 demonstrates how to create a section object. `spacing` is the distance between the transverse reinforcement bars, `x_coord_start` is the x-coordinate of where the section with the given spacing starts and `x_coord_end` is the x-coordinate of where the section with the given spacing ends. As can be seen from the example block, additional parameters can be defined by the user to simplify using the same spacing for section objects at different coordinates.

```
1 ##section_i = Section(spacing, x_coord_start, x_coord_end) #All three inputs are int
2
3 ##Examples of the creation of sections:
4 '''Example block (can be copied and changed to create your own objects)
5 #Ex. 1:
6 section = Section(10, cover.side, beam.geometry.length - cover.side)
7
8 #Ex. 2:
9 spacing_large = 168
10 spacing_small = 86
11 section_1 = Section(spacing_small, cover.side, cover.side + 4*spacing_small)
12 section_2 = Section(spacing_large, cover.side + 4*spacing_small,
```

```
(beam.geometry.length/2) − 2*spacing_small)
section_3 = Section(spacing_small, (beam.geometry.length/2) − 2*spacing_small,
beam.geometry.length)
'''
```

**Script 8.6:** Creating the sections for the transverse reinforcement

A model of a beam with transverse reinforcement with different spacing is shown in Figure 8.3. The x-coordinates of the different sections, as well as the y-coordinates of the transverse reinforcement are marked on Figure 8.3.

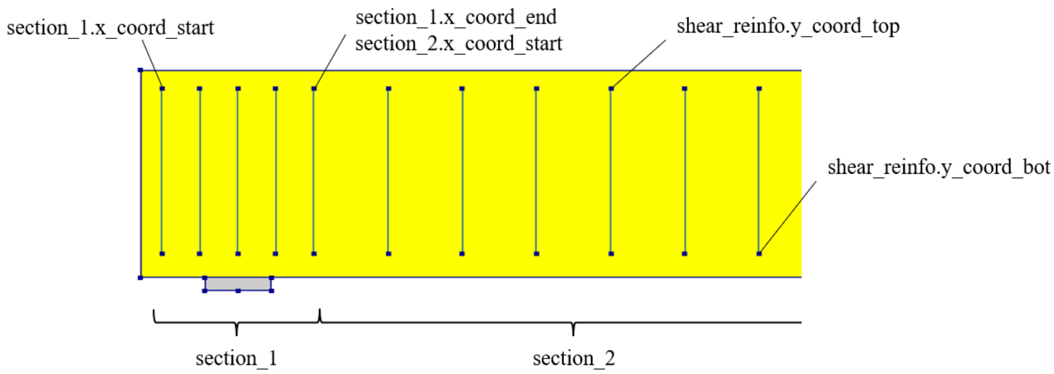



**Figure 8.3:** Sections and coordinates of transverse reinforcement

### 8.2.3 Plates

Steel plates are used to apply the point loads and supports. The geometry of the plates is created by defining the length and the height of the plates, as shown in Script 8.7. The width of the plates will equal the width of the beam. The Beam Script is designed in such a way that all plates will have the same geometry. Only line 7 - 8 of Script 8.7 require input. Line 5 should not be changed by the user, and is used to create the geometry object for the plates. As the number of plates are unknown at this stage, the plate geometry will first be attached to the plate objects in the main script.

```
1 #-#5.5 Plates
2 ##Available plates are loading plates (L) and support plates (S)
3 ##All plates will have the same geometry.
4 ##The width/thickness of the plates will be equal to the beam thickness
5 plateGeometry = Geometry() #creating geometry object for plates
6
7 plateGeometry.length = 150  #[mm]
8 plateGeometry.height = 35 #[mm]
```

**Script 8.7:** Creating the plate geometry

Three options are provided in Script A for creating the plates objects. Option 1 is to use the template for a three-point bending test, Option 2 is to use the template for a four-point bending test and Option 3 is to create the plates individually. The bending test templates are explained in Section 7.6. For the template options, the midpoint of the plates will be related to the span length of the beam. Option 3, creation of individual plates, allows the user to create plates by specifying the x-coordinate of the midpoint of the desired plates. As with the reinforcement objects, the user is free to choose the name of each plate object. Script 8.8 demonstrates how the plate objects can be created for Option 3. The type of plate has to be specified, which can either be a loading plate or a support plate.

```
1 ##nameOfPlate = Plates(typeOfPlate, name, x_coord)
2
3 ##CREATE YOUR PLATE OBJECTS HERE (option 3)
4 sup_plate = Plates("S", "Support plate 1", 50)
5 sup_plate.fixedTranslation_y = False #**Optional**
6 # sup_plate.fixedTranslation_x = False #**Optional**
7 load_plate = Plates("L", "Loading plate 1", 70)
```

**Script 8.8:** Creating the plates

`nameOfPlate` can be chosen by the user. `typeOfPlate` has to be set to either `"S"` for a support plate or `"L"` for a loading plate. `name` is a user defined name for the plate, and is the name the geometry object will be given in DIANA. `x_coord` is the x-coordinate of the midpoint of the plate object.

By default translations are fixed in both x- and y-direction for the support plates. These boundary conditions can be changed. The template for creating plate objects is shown in Script 8.9. By uncommenting line 5 and/or 6, the translations in x- and y-direction can be changed from fixed to free translations.

```
1 ##Copy this block to create plate objects:
2 '''Empty template
3 templateName = Plates(   )
4 # nameOfPlate.fixedTranslation_y = False #**Optional**
5 # nameOfPlate.fixedTranslation_x = False #**Optional**
6 '''
```

**Script 8.9:** Template for creating the plates

## 8.3 Loads

The Beam Script is designed in such a way that the dead weight of the beam always is applied in the first load step. The dead weight loading is evaluated directly by DIANA, based on the mass density derived from the material properties of specific materials [6]. Embedded reinforcement in DIANA does not contribute to the weight of the element [6]. The density of the steel plates is set to zero. In other words, the weight (mass) of the concrete determines the dead weight loading.

Point loads are applied in the second load step. While several load types can be chosen in DIANA, only point loads have been implemented for the Beam Script. Similarly to the creation of the plates, two options are provided in Script A for the creating of the point loads. Option 1 is to use one of the bending test templates, which is further explained in Section 7.6. For the template option, only one load has to be defined. Option 2, creation of individual loads, corresponds to the third option for creation of the plates. Each individual loading plate, which have been created in option 3 of section `5.5 Plates` in Script A, should now be associated with an individual point load. This is done by setting a target, in the form of a loading plate, for each of the created point loads.

Script 8.10 displays option 2 for creation of the point loads. Line 2 explains how a load object can be created. `typeOfLoad` should be set to "Point", which indicates the creation of a point load.

`name` is a user defined name for the load, and is the name the load object will be given in DIANA. `value` is the magnitude and direction of the point load. It is important to note that negative vertical loads act downwards, as the positive y-axis is defined in the upwards direction. `direction` should be denoted as either `"X"` or `"Y"`. The plate objects should be created below line 21 in Script 8.10. Line 5 explains how the target plate of the point load should be defined. The example block demonstrates how point loads can be created and attached to the related loading plates.

```
1  #-#6.2 Option 2: Creation of individual loads
2  ##load = Loads(typeOfLoad, name, value, direction)
3  ##Each load should have a plate object as its target.
4  ##A target is defined as follows:
5  #load.target = plateobject
6
7  ##Examples of the creation of point loads:
8  '''Example block (can be copied and changed to create your own objects)
9  load1 = Loads("Point", "Load 1", -1000, "Y")
10 load1.target = plate1
11 load2 = Loads("Point", "Load 2", -200, "Y")
12 load2.target = plate2
13 '''
14
15 ##Copy this block to create point load objects:
16 '''Empty template
17 templateName_Load = Loads()
18 templateName_Load.target = templateName_Plate
19 '''
20
21 ##CREATE YOUR LOAD OBJECTS HERE (option 2)
```

**Script 8.10:** Option 2: creation of individual point loads

## 8.4 Material models

### 8.4.1 Concrete

The concrete material model has one required input. This is the characteristic cylinder compressive strength, $f_{ck}$, of the concrete, as shown in Script 8.11. Based on this value, the rest of the concrete material properties will be retrieved from EC2, as recommended by Rijkswaterstaat [1]. The material properties which is not given in EC2, like the tensile fracture energy, will be estimated according to MC2010. In practice, this is done by creating three material models in DIANA: one based on EC2, one based on MC2010, and one with all the desired properties. The last material model is assigned to the concrete beam.

```
#-#7.1 Concrete material model
#The concrete material is defined as follows: concrete = Concrete(f_ck)
concrete = Concrete(30)
```

**Script 8.11:** Creating the material model for the concrete

The compressive fracture energy, $G_C$, can not be retrieved from the DIANA material model based on EC2 nor MC2010. The default value of this parameter is therefore estimated in accordance with the guidelines [1], as seen in Equation 8.1:

$$G_C = 250 \cdot \frac{f_{ck}}{f_{cm}} \cdot 0.073 f_{cm}^{0.18} \tag{8.1}$$

$f_{cm}$ is the mean compressive strength of the concrete, calculated as shown in Equation 8.2 [2]:

$$f_{cm} = f_{ck} + 8\ \text{MPa} \tag{8.2}$$

A total strain based crack model will be used for the concrete material model, as recommended. A rotating crack model will be assigned if the RC beam has shear reinforcement, and a fixed crack model will be used if the RC beam does not have shear reinforcement. As mentioned in Section 3.3, the input for the total strain crack model consist of two parts: the basic material properties and the definitions of the material behaviour in tension, compression and shear. An overview of the curves and models used for the concrete material behaviour is presented in Table 8.1. For further information about the different models, see Chapter 3. These are all based on the recommendations by Rijkswaterstaat. A stress confinement model has been included, even though not using one is a conservative measure, as the aim is to model the actual behaviour of the RC beam as accurately as possible. The stress confinement model by Selby & Vecchio has been chosen.

**Table 8.1:** Default material behaviour of concrete

| Crack model | TSCM |
|---|---|
| Crack orientation | Rotating<br>Fixed* |
| Tensile curve | Exponential |
| Crack bandwidth specification | Govindjee |
| Possion's ratio reduction model | Damage based |
| Compression curve | Parabolic |
| Compressive strength reduction | Vecchio and Collins 1993 |
| Lower bound of reduction curve | 0.4 |
| Stress confinement model | Selby and Vecchio |
| Shear retention model* | Damage based* |

* Only used for RC beams without shear reinforcement

If desired, the default material properties can be changed by the user. Script 8.12 presents the optional input for the concrete material model. The properties can be changed by uncommenting the desired lines. The properties that remains commented, will use recommended values. If the recommended curves are changed, the Beam Script is not guaranteed to run since additional parameters might be needed. To avoid this issue, the curves might be changed by the user in the DIANA graphical interface after generating the model, instead of directly in the input script. Suggested input are presented for some of the optional parameters. To use the nonlinear tension softening curve according to Hordijk, instead of the default exponential softening curve, is suggested. This curve is also recommended to model the tensile behaviour [1]. Moreover, to ignore the effects of the lateral confinement instead of using the confinement model by Selby & Vecchio is suggested, due to the fact that this is a conservative assumption.

```
## The following properties can be changed:
# concrete.Youngs_modulus = #**Optional**
```

```
# concrete.poissons_ratio = #**Optional**
# concrete.mass_density = #**Optional**
# concrete.tensile_strength = #**Optional**
# concrete.compressive_strength =  #**Optional**
# concrete.tensile_FractureEnergy = #**Optional**
# concrete.compressive_FractureEnergy = #**Optional**
# concrete.aggregate_type = #**Optional**
# concrete.tensile_curve = "HORDYK" #**Optional**
# concrete.crackBandwidt_specification = #**Optional**
# concrete.poissonsRatio_reductionModel = #**Optional**
# concrete.compression_curve = #**Optional**
# concret.lateralCracking_reductionModel = #**Optional**
# concrete.lateralCracking_reductionCurve_lowerBound =  #**Optional**
# concrete.confinementModel = "NONE" #**Optional**
```

**Script 8.12:** Optional user defined concrete material properties

### 8.4.2 Reinforcement steel

The Von Mises plasticity model will be used to model the reinforcement. The linear elasticity is based on the Young's modulus, which is decided when creating the reinforcement in part `5.2 Reinforcement` of Script A, see Section 8.2.2.2 and Section 8.2.2.3. A bilinear stress-strain diagram is used to model the plastic hardening, as shown in Figure 8.4. The needed user input for this diagram is `f_y, f_u` and `epsilon_u`, which are already specified when creating the reinforcement bars in section `5.2 Reinforcement` of Script A.

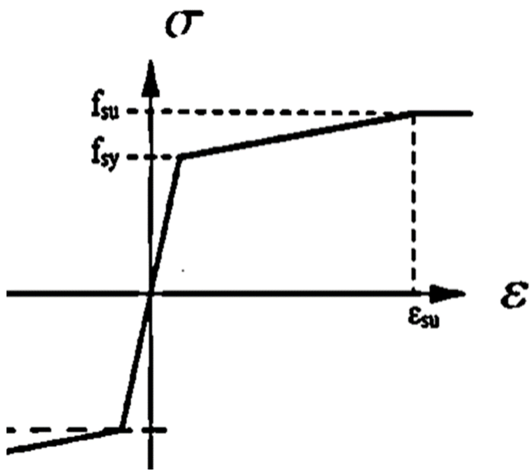



**Figure 8.4:** Stress-strain diagram for reinforcement steel

The default properties of the Von Mises plasticity model for the reinforcement are summarised in Table 8.2. A strain hardening hypothesis, with isothropic hardening, has been chosen. This is the DIANA default. Except for the parameters used to define the bilinear stress-strain diagram, the properties of the material model for the reinforcement steel are not changeable.

**Table 8.2:** Default properties of the Von Mises plasticity model

| | |
|---|---|
| Plastic hardening | Plastic strain - yield stress |
| Strain-Stress diagram | Bilinear |
| Hardening hypothesis | Strain hardening |
| Hardening type | Isotropic |

### 8.4.3 Steel for plates

A linear-elastic isotropic behaviour is assumed for the steel plates. The Young's modulus and the Poisson's ratio are the required input of the material model for the steel plates. The default values for these parameters are shown in Script 8.13. As mentioned in Section 8.3, the density of the steel plates is set to zero.

```
#-#7.2 Steel plates material model
#A linear-elastic behaviour is assumed for the plates

Steel_Plates.e_mod = 2000000
Steel_Plates.poissons_ratio = 0.3
```

**Script 8.13:** Required parameters of the material model for the steel plates

## 8.5 Structural interfaces

No-tension/no-friction 2D line interfaces are added between the steel plates and the concrete beam. For more information about structural interfaces, see Section 5.3. By default, the values presented in Table 8.3 will be used for the for the stiffness properties of the structural interfaces. The normal stiffness is denoted $K_{nn}$ and the shear stiffness is denoted $K_t$. The default values are taken from the beamscript by de Putter [13], and can optionally be changed by the user. As can be seen from Table 8.3, a high value is used for the normal compressive stiffness, while low values are used for the shear stiffness and normal tensile stiffness as recommended. A diagram based on the normal stiffness is used to define the nonlinear relation. The default traction-displacement diagram in normal direction is shown in Figure 8.5, where $K_{nn}$ determines the slope. Compression is defined as negative for Figure 8.5.

**Table 8.3:** Default interface properties [13]

| $K_{nn}$ in tension | $K_{nn}$ in compression | $K_t$ |
|---|---|---|
| 1.0 e-09 N/mm$^3$ | 1.0 e+03 N/mm$^3$ | 1.0 N/mm$^3$ |

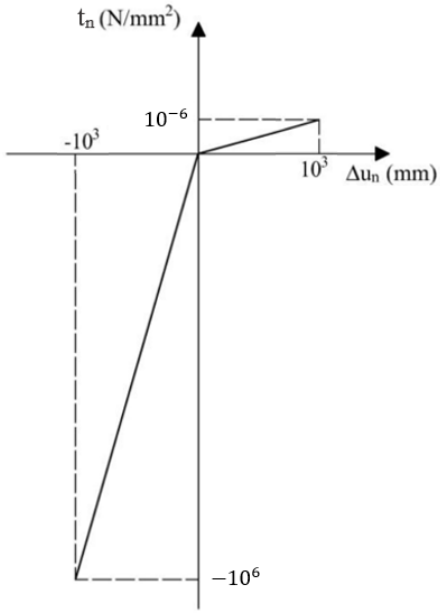



**Figure 8.5:** Default traction-displacement diagram in normal direction for structural interfaces (not to scale)

## 8.6 Mesh

The required input for the finite element mesh is the element size. For more information on how to decide on a suitable element size, see Section 2.2.2.1. As can be seen in line 4 of Script 8.14, an element size of 25 mm is set as default. For meshing the concrete and the steel plates, 8-node quadrilateral elements (CQ16M) with a full Gauss integration scheme (3×3) are used by default. The default mesh properties are summarised in Table 8.4. Optionally, the mesh order and mesher type could be changed by the user, as shown in line 7 - 8 of Script 8.14.

```
1 ###9. MESH
2 ##Recommended mesh is used as default.
3 ##Maximum elementsize = min(beam.Length/50, beam.Height/6)
4 Mesh.elementsize = 25 #[mm]
5
6 ## The following properties can be changed:
7 # Mesh.meshorder =  #**Optional**
8 # Mesh.meshertype =  #**Optional**
```

**Script 8.14:** Creating the mesh

**Table 8.4:** Default mesh properties

| | |
|---|---|
| Mesher type | Quadrilateral |
| Mesh order | Quadratic |
| Element type | Plane Stress |
| Element name | CQ16M |
| Integration scheme | 3×3 Gaussian |

Embedded bar reinforcement elements are used for the longitudinal and the transverse reinforcement. Embedded reinforcement is further explained in Section 2.2.2.2. The integration scheme for the bar reinforcement is derived from the one for the embedding structural elements (i.e. the concrete elements) [6]. A perfect bond is assumed.

## 8.7 Analyses

Two types of analyses can be performed using the Beam Script: a linear analysis and a nonlinear analysis. While the focus of this master's thesis is NLFEA of RC beams, it is recommended to run a linear analysis before executing the nonlinear analysis. A nonlinear analysis is much more time-consuming than a linear analysis. Hence, the latter is recommended to use first, to check that the RC beam behaves as expected [23]. By default, the options to run the analyses, when running the Beam Script in DIANA, are set to false. In other words, both the linear and nonlinear analysis, with all the needed properties, will be added in the DIANA graphical interface, but neither of these will be run. To run an analysis, the option `nameOfAnalysis.runSolver` has to be set to `True`. Script 8.15 shows how this option can be chosen in Script A, in this case for the nonlinear analysis. As can be seen from Script 8.15, the line `NLFEA.runSolver = True` (line 3) has to be uncommented to run the nonlinear analysis when the Beam Script is executed in DIANA. `NLFEA` is the given object name of the nonlinear analysis. The section for running the linear analysis in Script A is structured the same way.

```
1 #'#10.2.7 RUN NONLINEAR ANALYSIS
2 ##To run the analysis, uncomment the following line:
3 NLFEA.runSolver = True #**optional**
```

**Script 8.15:** Option to run the nonlinear analysis when the Beam Script is executed in DIANA

### 8.7.1 Linear analysis

No user input is required for the linear analysis. By default, an iterative method has been chosen to solve the system of equations. Furthermore, the DIANA primary results output for a structural linear static analysis are selected by default, which include displacements, strains, stresses, forces and fracture mechanics [6]. Optionally, the solution method and output can be changed by the user, as shown in line 6 - 7 of Script 8.16. Line 3 of Script 8.16 creates the linear analysis object, and should not be changed by the user. `LFEA` is the given object name of the linear analysis.

```
1 #-#10.1 Linear Analysis
2 ##DIANA Primary output is chosen for the linear analysis.
3 LFEA = Analysis("Linear analysis", "Linear") #creating linear static analysis
4 
5 ## The following properties can be changed:
6 # LFEA.method = **Optional input**
7 # LFEA.output = **Optional input**
```

**Script 8.16:** Creation of the linear analysis (LFEA)

### 8.7.2 Nonlinear analysis

For the nonlinear analysis, several input are required. However, suggested input are provided for all parameters, and the nonlinear analysis can be run without the user having to change any of the properties. Even so, suitable input for the step size, number of steps and number of iterations greatly depends on the RC beam to be modelled, and will most likely have to be changed by the user. The properties for the nonlinear analysis are presented in Table 8.5. The optional properties are marked with *, the rest of the properties presented in Table 8.5 will be used by default.

**Table 8.5:** Properties for the nonlinear finite element analysis

| | |
|---|---|
| Incrementation method | Energy based adaptive loading |
| Iteration method | Regular Newton-Raphson<br>Modified Newton-Raphson* |
| Analysis control | Arc-length<br>Line search |
| Arc-length control nodes | Whole beam |
| Convergence criteria | Force norm<br>Energy norm<br>Displacement norm* |
| All convergence norms have to be satisfied | False** |
| Continue if no convergence | True** |

*Optional, not selected by default
**Can be changed to True/False.

The selected incremental procedure for the nonlinear analysis is load control. An adaptive load incrementation method will be used, as recommended. The implemented automatic procedure for the adaptive loading is based on energy. This method is further explained in Section 2.2.1.2.2. The energy method has to be combined with arc-length control. By default, arc-length control will be applied, as shown in Script 8.17. However, Script A allows the user to turn off the arc-length control by changing the bool value in line 4 to `False`. Seeing that the use of arc-length control is highly recommended, simply commenting line 4 will result in the use of the default value, which is `True`. Since energy based adaptive loading, which has to be combined with arc-length control, is the only implemented incrementation method, this control should not be turned off. Even so, it has been chosen to include the option to turn of the arc-length control in Script A, as more options for the incrementation method might be added in the future. For example does the iteration based method, also explained in Section 2.2.1.2.2, not have to be combined with arc-length control, even though this is highly recommended. When running the Beam Script, Script D checks that the energy based method is combined with arc-length control. If this is not the case, the incrementation method will not be added, and the nonlinear analysis will terminate after one single load step.

```
1 #'#10.2.1 Arc length control:
2 ##Arc length control is strongly recommended.
3 ##The arc length control will be applied over the whole beam.
4 NLFEA.arcLengthControl = True #**Default and strongly recommended as true**
```

**Script 8.17:** Arc-length control

The required parameters for the energy based incrementation method are shown in Script 8.18. Line 3 of Script 8.18 should not be changed by the user. The user should be careful when choosing the limits for the step size, and some degree of trial and error will likely have to be done to in order to decide appropriate load increments. The suggested parameter values have been chosen based on personal experience from working with the Beam Script, as well as the validation studies by Rijkswaterstaat [25]. Trying to run the nonlinear analysis with the suggested parameters before adjusting them is therefore recommended. Changing to a smaller step size might improve the stability of the analysis, but can also result in a much more time-consuming analysis.

```
##Option 1: Energy based adaptive loading
##The energy based method MUST be combined with arc length control.
incrementation.method = "ENERGY"
incrementation.initial_step_size = 5    #initial size for the first step
incrementation.max_step_size = 10       #upper limit of the step size
incrementation.min_step_size = 3        #lower limit of the step size
incrementation.nrOfSteps = 150          #maximum number of steps
```

**Script 8.18:** Energy based adaptive loading

Newton-Raphson is the recommended method for the iterative procedure [1], and Script A provides the option to choose either the Regular Newton-Raphson method or the Modified Newton-Raphson method. The desired option should be uncommented, while the other option should be commented. As seen in Script 8.19, which is an excerpt of Script A, the Regular Newton-Raphson method is chosen by default. For more information about both methods, see Section 2.2.1.1.1. `nrOfIterations` (line 11 or 16) is the only parameter that require input. The value of this parameter should be set high enough such that a large number of nonconverged steps are avoided, but not too high, as this can result in a very time-consuming analysis. By default, the Newton-Raphson methods set up the tangential stiffness before each iteration. The parameter `firstStiffnessMatrix` is optional (line 19), and allows the user to specify an alternative for the first iteration: new tangential stiffness, linear stiffness or the stiffness of the last iteration of the previous step [6]. By default, a new tangential stiffness is set up also for the first iteration.

```
1  #'#10.2.3 Iteration method:
2  ##Only Newton-Raphson methods have been implemented.
3  ##Choose one of the options. The section with the chosen option should be uncommented,
4  ##while the other options should be commented.
5  iteration = Iteration() #creating iteration object
6  NLFEA.setIterationMethod(iteration) #adding iterative procedure to the analysis
7
8  ## Option 1: Regular Newton-Raphson
9  iteration.method = "NEWTON-RAPHSON"
10 iteration.typeOfMethod = "REGULA"
11 iteration.nrOfIterations = 100
12
13 # ## Option 2: Modified Newton-Raphson
14 # iteration.method = "NEWTON-RAPHSON"
15 # iteration.typeOfMethod = "MODIFI"
16 # iteration.nrOfIterations = 100
17
18 ## The following properties can be changed:
19 # iteration.firstStiffnessMatrix = **Optional input**
```

**Script 8.19:** Iteration methods

Using a line search algorithm is recommended, and will be applied by default. Script 8.20 presents how this option is presented in Script A. Similarly to the arc-length method, the bool value has to be changed to `False` for the line search algorithm not to be applied.

```
#'#10.2.4 Line search:
##Using a line search algorithm is recommended.
NLFEA.lineSearch = True #**Default and recommended as true**
```

**Script 8.20:** Line Search

At least one convergence criterion has to be chosen, but the user is allowed to choose multiple criteria. Line 8 - 10 show the required parameters, and can be changed by the user. By default, a force norm and an energy norm are chosen for the convergence criteria.

```
1  #'#10.2.5 Convergence criteria:
2  ##One or multiple convergence criteria MUST be chosen.
3  ##Recommended to use a force norm and an energy norm.
4  ##Choose one of the options. Optional to choose one or more convergence criteria.
5  convergence = Convergence()          #creating convergence object
6  NLFEA.setConvergence(convergence)    #adding convergence criteria to the analysis
7
8  convergence.useForceNorm = True      #**Default and recommended as true**
9  convergence.useEnergyNorm = True     #**Default and recommended as true**
10 convergence.useDispNorm = False      #**Default and recommended as false**
```

**Script 8.21:** Choosing the convergence criteria

Two options are provided for defining the convergence criteria. The first option is to use suggested tolerances for the force norm and/or the energy norm. These tolerances are based on suggestions by Rijkswaterstaat and presented in Table 8.6.

**Table 8.6:** Suggested tolerances [1]

| | Load case 1 (Dead load) | Load case 2 (Point loads) |
|---|---|---|
| Force Norm | 0.05 | 0.01 |
| Energy Norm | 0.01 | 0.001 |

The second option is for the user to choose the tolerances, as shown in Script 8.22. One or multiple convergence criteria should be uncommented and assigned values, if this option is chosen. The convergence criteria must be carefully chosen, as further explained in Section 2.2.1.3.

```
## Option 2: Choose your own tolerances:
##One or multiple convergence criteria MUST be chosen.

##Force norm:
# convergence.forceNorm =
# convergence.forceNorm_deadLoad =

##Energy norm:
# convergence.energyNorm =
# convergence.energyNorm_deadLoad  =

##Displacement norm:
# convergence.dispNorm =
# convergence.energyNorm_deadLoad  =
```

**Script 8.22:** Option 2: choose own tolerances for the convergence criteria

Some additional choices for the nonlinear analysis are given in Script A, these are presented in Script 8.23. By default, the iteration process will be terminated if one of the specified convergence criteria is satisfied, unless explicitly specified that all convergence criteria should be satisfied simultaneously. This can be specified by setting the bool value of `allConvergenceNormsHaveToBeSatisfied` to `True`. If no convergence occurs within the maximum number of iterations, the analysis run will be continued by default. This can be changed by setting the bool value of `continueIfNoConvergence` to `False`.

```
#'#10.2.6 Additional choices for analysis
NLFEA.allConvergenceNormsHaveToBeSatisfied = False  #**Default and recommended as false**
NLFEA.continueIfNoConvergence = True                 #**Default and recommended as true**
```

**Script 8.23:** Additional choices for the NLFEA

## 8.8 Output from nonlinear analysis

### 8.8.1 DIANA output

Script 8.24 displays the DIANA output for the nonlinear analysis that can be selected in Script A. By default, all output are set to true. These will be listed in the DIANA Results window if the analysis process has been carried out successfully. Each of these output is optional, and can be changed to `False` by the user. The left side of the python dictionary `NLFEA.output` in Script 8.24, has the output names. These names are the same as can be selected by the user in the DIANA graphical interactive environment, as is shown in Figure 8.6

```
#-#11.1 Analysis output:
NLFEA.output = {
        "DISPLA TOTAL TRANSL GLOBAL" : True,     #**Optional**
        "FORCE REACTI TRANSL GLOBAL" : True,     #**Optional**
        "STRAIN TOTAL GREEN GLOBAL" : True,      #**Optional**
        "STRAIN TOTAL GREEN PRINCI" : True,      #**Optional**
        "STRAIN CRKWDT GREEN GLOBAL" : True,     #**Optional**
        "STRAIN CRACK GREEN" : True,             #**Optional**
        "STRAIN CRKWDT GREEN PRINCI" : True,     #**Optional**
        "STRESS TOTAL CAUCHY GLOBAL" : True,     #**Optional**
        "STRESS TOTAL CAUCHY PRINCI" : True      #**Optional**
}
```

**Script 8.24:** DIANA output of NLFEA

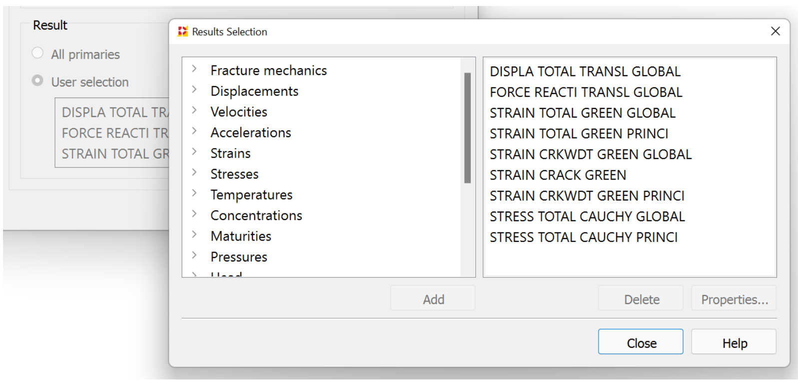



**Figure 8.6:** DIANA Results Selection window for user selected results

The window shown in Figure 8.6 can be opened with the following click order, after having run the Beam Script in DIANA:

| |
|---|
| **Analysis** → Nonlinear analysis ↳ Structural nonlinear ↳ Output → Edit properties ⦿ User selection → Modify |

### 8.8.2 Additional output

The option to create and save a load-displacement curve for the y-direction displacement at a specified node is provided. Furthermore, the option to generate a CSV-file of the load factor and y-direction displacement at a specified node is added. The output from these options will be saved to the working folder of the Beam Script, or to the user specified directory if this option has been selected. The output `"DISPLA TOTAL TRANSL GLOBAL"`, shown in Script 8.24, as well as the command `NLFEA.runsolver`, shown in Script 8.15, has to be set to `True` for the additional output to be produced when running the Beam Script. Downward displacement is defined as positive for these output. Both the load factor and the displacement can be scaled by a factor, as seen in line 12 - 13 of Script 8.25. The load factor itself is given by DIANA, and tells how much the initial load has been scaled for each load step of the nonlinear analysis.

Section `11.2 Load-displacement graph` of Script A, when uncommented, is presented in Script 8.25. By default, this output is not selected and the whole section is commented. As can be seen from line 6 - 7 of Script 8.25, the coordinates of the point where the displacement should be retrieved are the required parameters. The suggested values for these parameters are the midpoint at the bottom of the RC beam. Optional parameters are also provided. `Scalefactor` can be used to scale the load factor and/or displacement. By default, the y-values of the load-displacement graph will be the load factor for each load step and the x-values will be the corresponding displacements. Unless a negative scale factor is used, the downward displacements will be represented as positive. `x_label` and `y_label` allow the user to change the name of the axis labels. `xLim` and `yLim` can be used to specify the axis limits. The load-displacement curve is saved to a new folder, named "Plots", within the working folder. In the case of a parametric study, the load-displacement curves of all the performed analyses will be saved within this same folder. Figure 8.7 shows an arbitrary example of a generated load-displacement graph.

```
1  #-#11.2 Load-displacement graph (displacement in y-dir) **Optional**
2  ##Edit and uncomment this section to generate a load-displacement graph.
3  ##Downward displacement is defined as positive by default.
4
5  #Specify the coordinates of the point where the displacement will be retrived.
6  LoadDispY_Graph.x_coord = beam.geometry.length/2 #[mm]
7  LoadDispY_Graph.y_coord = 0 #[mm]
8
9  ##The following options can be changed:
10 ##x_label, y_label = str
11 ##xLim, y_Lim = [lowerBound, upperBound]; lowerBound, upperBound = int
12 # Scalefactor.loadFactor_plot = #**optional**
13 # Scalefactor.displacement_plot = #**optional**
14 # LoadDispY_Graph.xlabel = #**optional**
15 # LoadDispY_Graph.ylabel = #**optional**
16 # LoadDispY_Graph.xLim = #**optional**
17 # LoadDispY_Graph.yLim =  #**optional**
```

**Script 8.25:** Optional output: Load-displacement graph

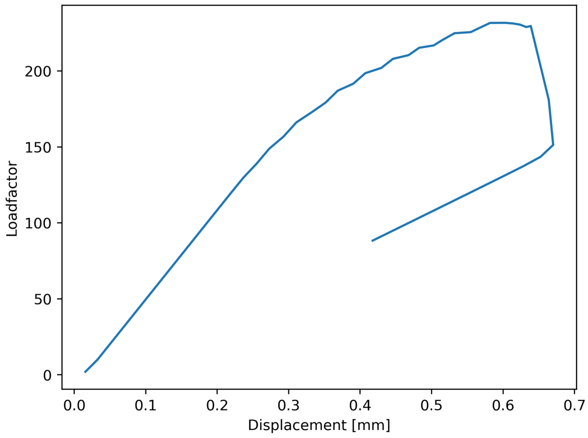



**Figure 8.7:** Example of generated load-displacement graph

Section `11.3 Load-displacement CSV` of Script A, when uncommented, is presented in Script 8.26. By default, this output is not selected and the whole section is commented. Similarly to the load-displacement graph, the coordinates of the point where the displacement should be retrieved are the required parameters. Optionally, scale factors can be defined. Line 17 of Script 8.26 lets the user specify the decimal places for the output. By default, two decimal places has been chosen. A random example of a generated CSV-file, with a scaled load factor, is shown i Figure 8.8. The header of the CSV-file is automatically changed to indicate whether the displacement or load factor have been scaled. Although CSV stands for Comma Separated Values, the delimiter could be anything. As can be seen from Figure 8.8, the chosen delimiter is a semicolon (;). This is to simplify the process of importing the CSV-file to Excel, where comma (,) might be used for decimals.

```
1  #-#11.3 Load-displacement CSV (displacement in y-dir)**Optional**
2  ##Edit and uncomment this section to generate a CSV-file for
3  ##load-displacement at specified coordinate.
4  ##Downward displacement is defined as positive by default.
5  ##The CSV-file will have the following format:
6  ##  (Scaled) Loadfactor;(Scaled) Displacement
7  ##  1.00;0.50
8  ##  2.30;0.65
9  
10 ##Specify the coordinates of the point for which the CSV-file will be generated:
11 LoadDispY_CSV.x_coord = beam.geometry.length/2 #[mm]
12 LoadDispY_CSV.y_coord = 0 #[mm]
13 
14 #The following options can be changed:
15 # Scalefactor.loadFactor_CSV =  #**optional**
16 # Scalefactor.displacement_CSV = #**optional**
17 # LoadDispY_CSV.decimalPlaces = #**optional**
```

**Script 8.26:** Optional output: Load-displacement graph

```
Displacement [mm];Scaled Loadfactor
0.02;2.00
0.03;9.99
0.05;19.98
0.07;29.97
0.08;39.96
0.10;49.94
0.12;59.93
0.13;69.92
```

**Figure 8.8:** Example of the CSV-file format (scaled load factor)

# Part IV

# Experiments to benchmark the script

# 9 General information

Benchmarks can be described as well-defined problems that have already been solved. By recreating the problem, the accuracy and reliability of the selected solution strategy can be validated. For this thesis, the framework has been validated by simulating two benchmark studies of RC beams, which are both well-known published experiments. The experimental results have been compared to the results of the 2D nonlinear finite element solution from running the Beam Script. The goal of the analyses is to replicate the experimental test as much as possible, using nonlinear procedures.

The subsequent sections will deal with two case studies of RC beams, called Case B1 and Case B2. To demonstrate the use of both the fixed and the rotating crack model, one beam with shear reinforcement and one without shear reinforcement have been selected for the benchmark studies. Additionally, the benchmark beams are subject to different failure mechanisms. Case B1 has shear reinforcement and fails in bending, and is the beam VS-C3 from the experimental program by Vecchio & Shim (2004) [26]. Case B2 has no shear reinforcement and fails in shear, and is the beam SE-50A-45 from the experimental program by Collins & Kuchma (1999). The information about the two benchmarks are acquired from the "Validation of the Guidelines for Nonlinear Finite Element Analysis of Concrete Structures. Part: Reinforced beams" by Rijkswaterstaat [25] and the DIANA Verification Report [27]. The information from the first document will be favoured, if different information is given by the two documents.

The following sections will be structured similarly, describing respectively: the experimental setup and results, the finite element model, the properties of the nonlinear analysis, the results from the nonlinear analysis and a discussion of the results.

# 10 Case B1: Vecchio & Shim (2004)

Case B1 is based upon Beam VS-C3 [26] from the experimental program of Vecchio & Shim from 2004, which is an re-examination of the classical experiments by Bresler & Scordelis from 1963. The experimental setup is shown in Figure 10.1. VS-C3 is an experimental test of a RC beam under increasing static load until failure [27]. The beam fails in bending, and exhibits a flexural-compressive failure mode [28].

## 10.1 Experimental setup and results

### 10.1.1 Geometry and loading

The geometry of the beam and the reinforcement, as well as the loading, is shown in Figure 10.1 and Figure 10.2. The length of the beam is 6.840 m, the height is 0.552 m and the width is 0.152 m. The beam is subjected to three-point bending, as shown in Figure 10.1.

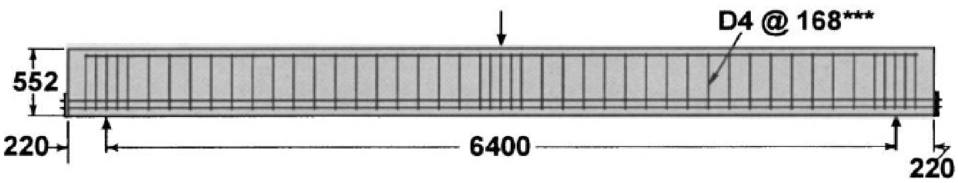



**Figure 10.1:** Case B1: Experimental setup (dimensions in mm) [25]

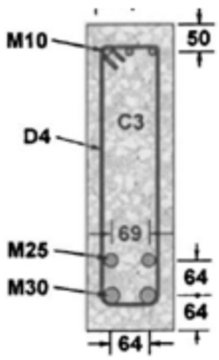



**Figure 10.2:** Case B1: Cross section [28]

### 10.1.2 Material properties

Table 10.1 lists the most important material properties of the concrete and the steel. As shown in Figure 10.2, four different types of reinforcement steel have been used in the beam. The given parameter values for the longitudinal reinforcement in Table 10.1 are for one single bar.

**Table 10.1:** Case B1: Material properties [25][28]

| Material | Parameter | Value |
|---|---|---|
| Concrete | Compressive strength, $f_{cm}$ | 43.5 MPa |
| Reinforcement steel M10 | Diameter, $\phi$ | 11.3 mm |
| | Cross section area, $A_s$ | 100 mm$^2$ |
| | Young's modulus, $E$ | 200000 MPa |
| | Yielding strength, $f_{ym}$ | 315 MPa |
| | Ultimate strength, $f_{um}$ | 460 MPa |
| | Ultimate strain, $\varepsilon_{su}$ | 0.023 |
| Reinforcement steel M25 | Diameter, $\phi$ | 25.2 mm |
| | Cross section area, $A_s$ | 500 mm$^2$ |
| | Young's modulus, $E$ | 220000 MPa |
| | Yielding strength, $f_{ym}$ | 445 MPa |
| | Ultimate strength, $f_{um}$ | 680 MPa |
| | Ultimate strain, $\varepsilon_{su}$ | 0.048 |
| Reinforcement steel M30 | Diameter, $\phi$ | 29.9 mm |
| | Cross section area, $A_s$ | 700 mm$^2$ |
| | Young's modulus, $E$ | 200000 MPa |
| | Yielding strength, $f_{ym}$ | 436 MPa |
| | Ultimate strength, $f_{um}$ | 700 MPa |
| | Ultimate strain, $\varepsilon_{su}$ | 0.048 |
| Reinforcement steel D4 (stirrups) | Diameter, $\phi$ | 3.7 mm |
| | Cross section area, $A_s$ | 25.7 mm$^2$ |
| | Young's modulus, $E$ | 200000 MPa |
| | Yielding strength, $f_{ym}$ | 600 MPa |
| | Ultimate strength, $f_{um}$ | 651 MPa |
| | Ultimate strain, $\varepsilon_{su}$ | 0.047 |
| Steel for plates | Young's modulus, $E$ | 200000 MPa |
| | Poisson's ratio, $\nu$ | 0.3 |

### 10.1.3 Experimental results

The experimental ultimate value of the applied load before failure was $P_{exp} = 265$ kN, with a corresponding deflection of 44.3 mm at midspan [25]. The beam experienced a flexural-compressive failure mode [28]. This means that failure was caused by crushing of concrete at the compression side, after yielding of the reinforcement steel [1]. The failure mechanism of B1 is shown in Figure 10.3. The load-deflection response is shown in Figure 10.4.


VS-C3

**Figure 10.3:** Case B1: Failure mechanisms at experimental ultimate value of applied load [28]

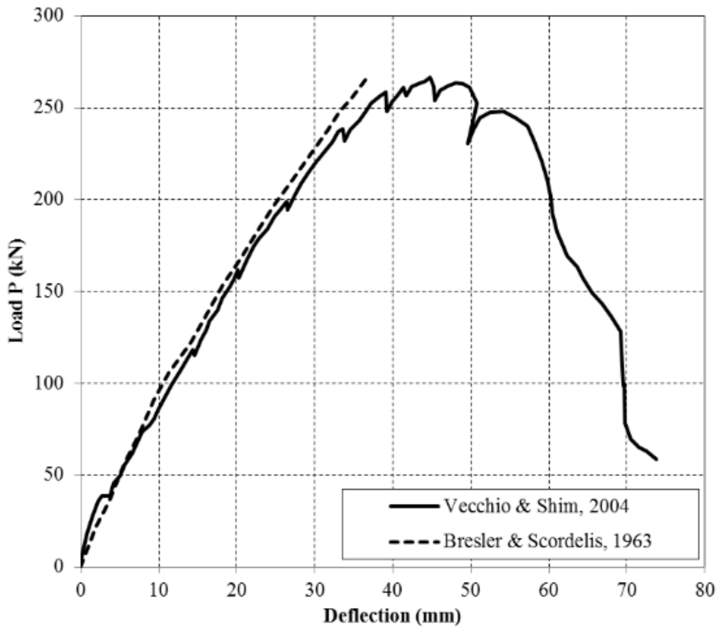



**Figure 10.4:** Case B1: Experimental load-deflection at midspan [25]

## 10.2 Finite element model

### 10.2.1 Geometry and loading

The finite element model has been generated by changing the parameters in the Script A in accordance with Table 10.1 and the geometry presented in Section 10.1. The created input script for Case B1 can be found in Appendix E. Longitudinal and transverse reinforcement have been added, and the template for a three-point bending test has been used. Hence, due to symmetry, only half the beam has been modelled. The model is shown in Figure 10.5. The stirrups have been modelled with a large spacing of 168 mm, as seen in Figure 10.1, and a small spacing of 68 mm near the steel plates. Dead weight loading has been applied in a single step as the first load case. In load case 2, an initial point load of P = 1 kN has been applied at the midpoint of the loading plate. The applied point load, P, is marked on Figure 10.5.

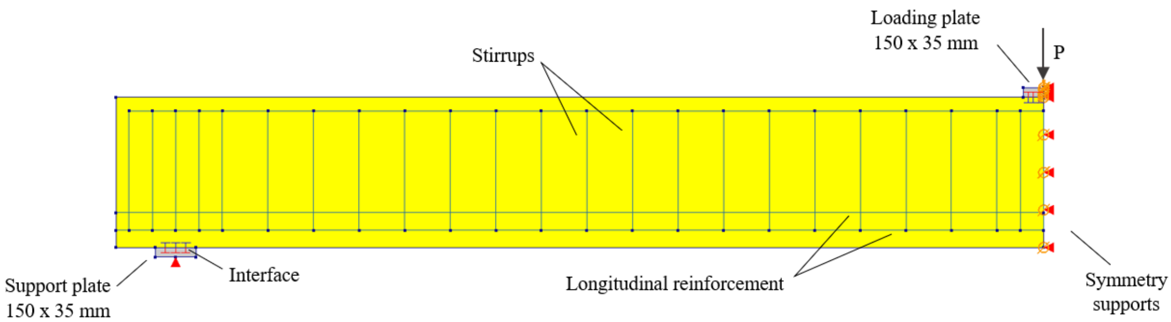



**Figure 10.5:** Case B1: View of the model

### 10.2.2 Material properties

Based on the given compressive strength, the concrete class is set to C35. The material properties which are not given in Table 10.1, have been estimated according to EC2 and MC2010 with reference to the concrete class. A total strain rotating crack model has been used. Table 10.2 presents the input for the arguments of the concrete material model.

**Table 10.2:** Case B1: Input of concrete material model

| | |
|---|---|
| Young's modulus, $E_{cm}$ | 34077 N/mm$^2$ |
| Poisson's ration, $\nu$ | 0.2 |
| Mass density | 2.4e-09 T/mm$^3$ |
| Crack model | TSCM |
| Crack orientation | Rotating |
| Tensile curve | Exponential |
| Tensile strength, $f_{ctm}$ | 3.2 N/mm$^2$ |
| Tensile fracture energy, $G_F$ | 0.144 N/mm |
| Crack bandwidth specification | Govindjee |
| Possion's ratio reduction model | Damage based |
| Compression curve | Parabolic |
| Compressive strength, $f_{cm}$ | 43.5 N/mm$^2$ |
| Compressive fracture energy, $G_C$ | 28.96 N/mm |
| Compressive strength reduction | Vecchio & Collins 1993 |
| Lower bound of reduction curve | 0.4 |
| Stress confinement model | Selby & Vecchio |

Embedded reinforcement has been used to model both the longitudinal and the transverse reinforcement. Since a 2D model has been used, the area of the longitudinal reinforcement has been defined as equivalent to the cross section area of the total number of bars. For example: $3 \times$ M10 bars $= 3 \times 100$ mm$^2$ $= 300$ mm$^2$. A Von Mises Plasticity model has been used, and the properties used to define the bilinear stress-strain diagram for the different types of reinforcement are given in Table 10.1. Figure 10.6 shows the stress-strain curve used for M30.

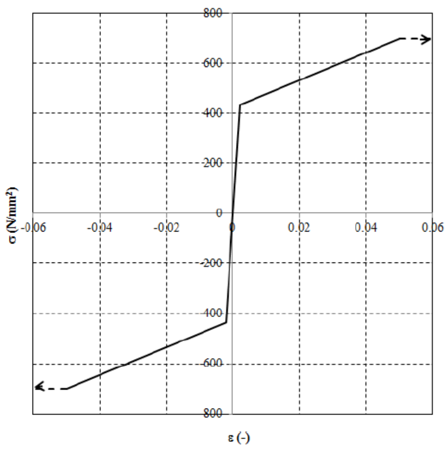



**Figure 10.6:** Case B1: Stress-strain curve for M30 [25]

For the steel plates, a linear elastic behaviour is assumed. The properties for the steel plates are given in Table 10.1.

Structural interface elements have been defined between the steel plates and the concrete beam. The interface properties have been taken from the validation studies by Rijkswaterstaat [25], and are presented in Table 10.3.

**Table 10.3:** Case B1: Interface properties

| $K_{nn}$ in tension | $K_{nn}$ in compression | $K_t$ |
|---|---|---|
| 3.42 e-08 N/mm$^3$ | 3.42 e+04 N/mm$^3$ | 3.42 e-08 N/mm$^3$ |

### 10.2.3 Mesh

The generated mesh for B1 is presented in Figure 10.7. The element size is set to 25 mm. Otherwise, default properties have been used for the mesh, these are summarised in Table 10.4.

**Table 10.4:** Case B1: Mesh properties

| | |
|---|---|
| Mesher type | Quadrilateral |
| Mesh order | Quadratic |
| Element type | Plane Stress |
| Element name | CQ16M |
| Integration scheme | 3×3 Gaussian |

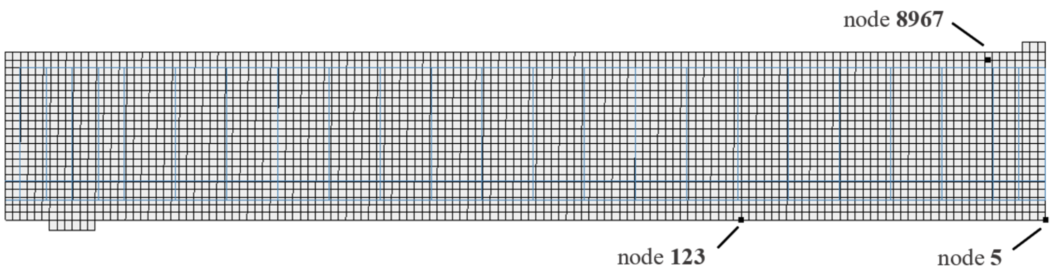



**Figure 10.7:** Case B1: Finite element mesh

The nodes marked on Figure 10.7 will be referred to in Section 10.4: Results of NLFEA.

## 10.3 Structural nonlinear analysis

The default properties for the nonlinear analysis have been kept, except for the number of steps which has been decreased. The suggested tolerances for the force norm and energy norm, presented in Table 8.6, have been chosen for the convergence criteria. Table 10.5 presents the input for the arguments of the NLFEA. The analysis has been performed until failure.

**Table 10.5:** Case B1: Properties of NLFEA

| | |
|---|---|
| Incrementation method | Energy based adaptive loading |
| Step size | Energy based |
| Factor for first load increment | 5 |
| Upper limit of step size | 10 |
| Lower limit of step size | 3 |
| Number of steps | 75 |
| Iteration method | Regular Newton-Raphson |
| Maximum number of iterations | 100 |
| Analysis control | Arclength + Line search |
| Convergence norms | $F0.05$, $E0.01$ (load case 1)<br>$F0.01$, $E0.001$ (load case 2) |
| All convergence norms have to be satisfied | False |
| Continue if no convergence | True |

## 10.4 Results of nonlinear finite element analysis

### 10.4.1 Load deflection

The NLFEA gives a peak load of **265** kN, and a maximum deflection at midspan of **47** mm. The load-deflection curve at midspan, node 5, is presented in Figure 10.8. The load values corresponding to yielding of the different reinforcement are indicated, as well as the load value corresponding to cracking of concrete. The post-peak branch of the load-deflection curve is plotted with a dashed line.

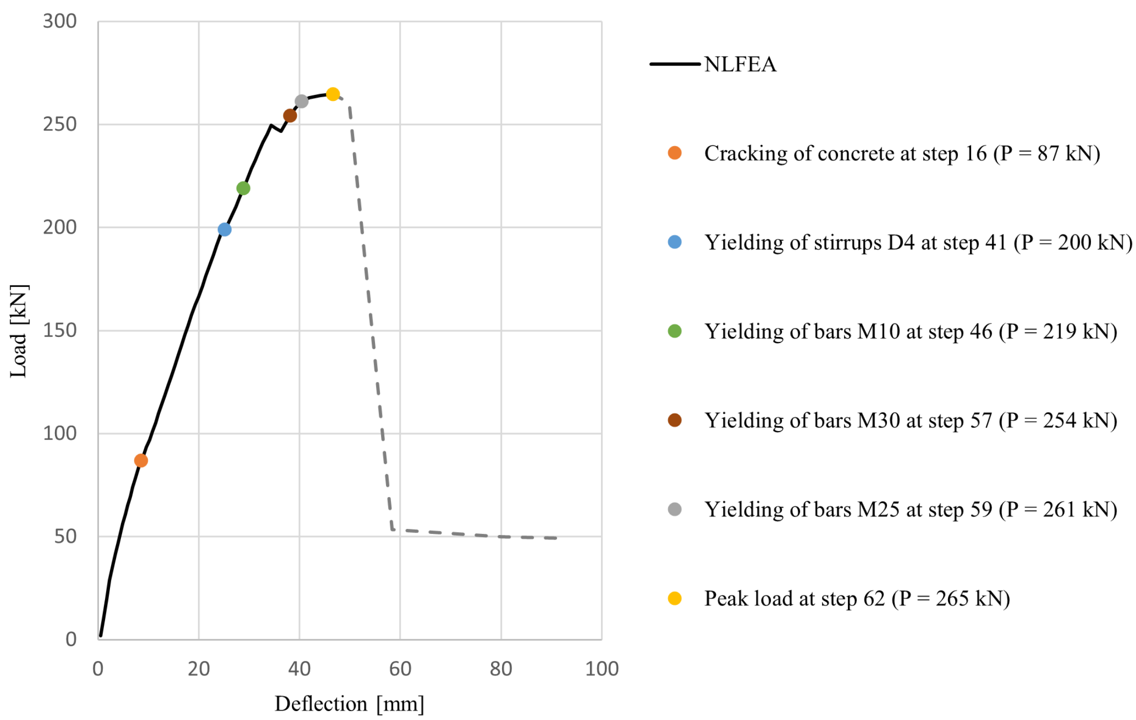



**Figure 10.8:** Case B1: Load-deflection curve at midspan (node 5)

### 10.4.2 Cracking

The first cracks are registered at load step 3. DIANA detects even very small crack widths. In load step 3 the largest crack has a width of $6 \cdot 10^{-4}$ mm, as seen in Figure 10.9.

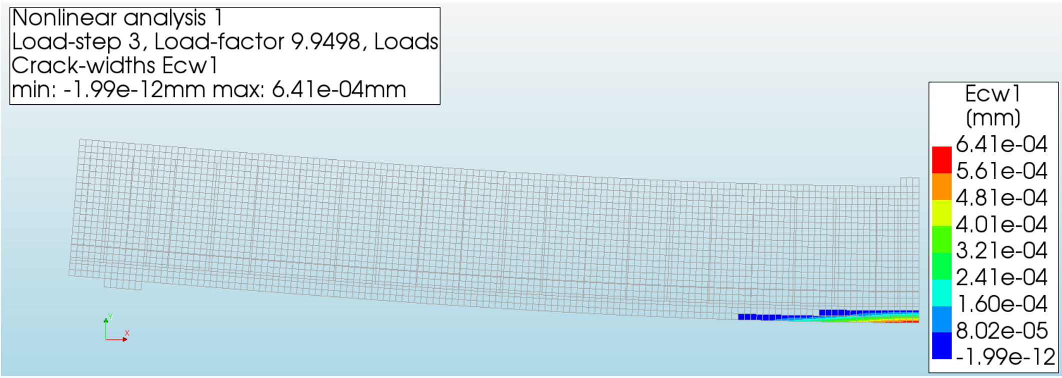



**Figure 10.9:** Case B1: Crack widths at load step 3 (P = 20 kN)

Figure 10.10 shows the crack widths at load step 16, here the applied load is 87 kN. At this load step most of the crack widths lay within the range of 0.01 - 0.14 mm, and the beginning of the expected crack pattern can be seen. Figure 10.10 corresponds to the cracking point marked in Figure 10.8.

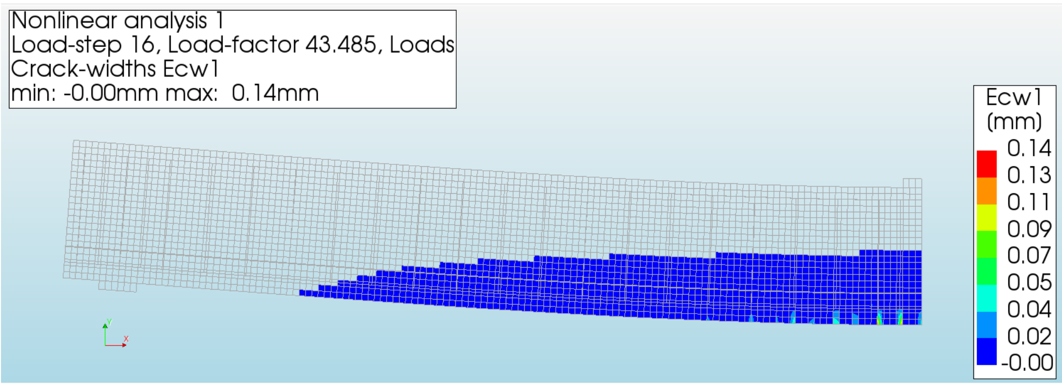



**Figure 10.10:** Case B1: Crack widths at load step 16 (P = 87 kN)

Figure 10.11 presents the crack widths and pattern for the peak load at load step 62.

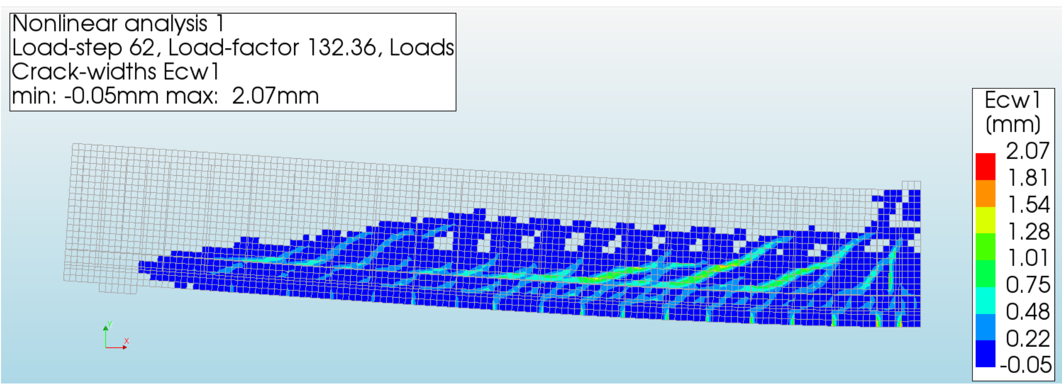



**Figure 10.11:** Case B1: Crack widths at load step 62 (P= 265 kN)

### 10.4.3 Crushing

Figure 10.12 shows the minimum principal stresses in the beam at peak load. Negative stresses indicate compression.

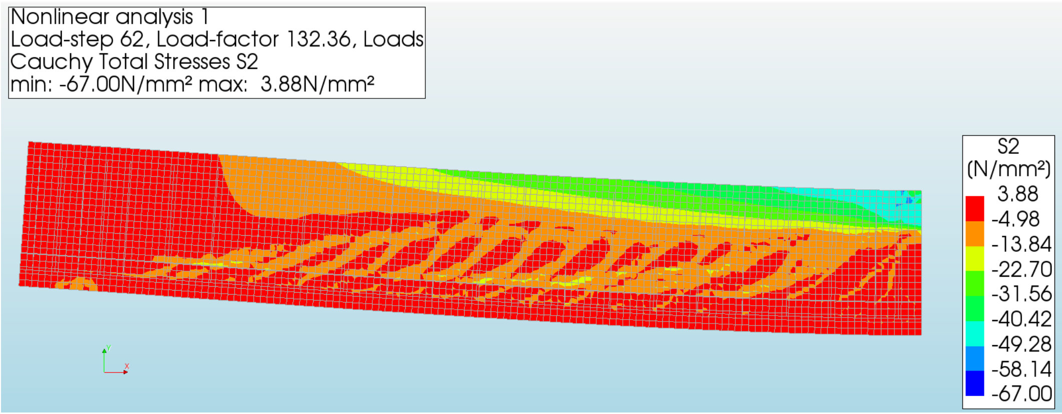



**Figure 10.12:** Case B1: Minimum principal stresses at load step 62 (P= 265 kN)

### 10.4.4 Yielding of reinforcement

The yielding of the different reinforcement types have been marked on Figure 10.8, and corresponds to the first load step where the reinforcement stress has exceeded the yielding strength of that reinforcement type. The subsequent figures display the reinforcement stresses of the different reinforcement types at the load steps marked on Figure 10.8. Positive stresses indicate tension, while negative stresses indicate compression.

The reinforcement steel D4 of the stirrups has a yielding strength of $f_{ym} = 600$ MPa. Figure 10.13 displays the reinforcement stresses of D4 at load step 41.

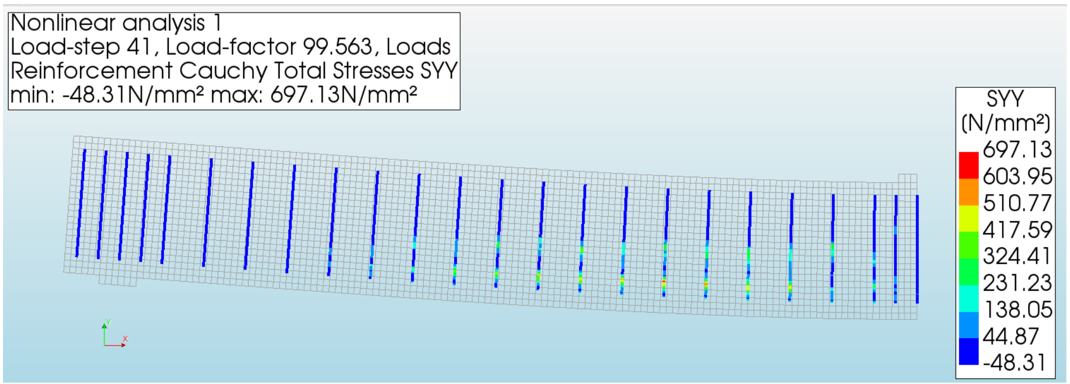



**Figure 10.13:** Case B1: Yielding of stirrups D4 at step 41 (P = 200 kN)

The reinforcement steel M10 has a yielding strength of $f_{ym} = 315$ MPa. Figure 10.14 displays the reinforcement stresses of M10 at load step 46.

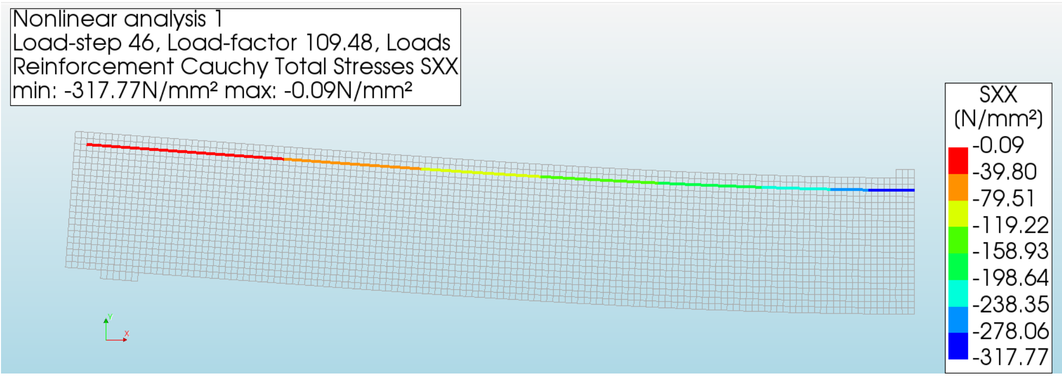



**Figure 10.14:** Case B1: Yielding of reinforcement bars M10 at step 46 (P = 219 kN)

The reinforcement steel M30 has a yielding strength of $f_{ym} = 436$ MPa. Figure 10.15 displays the reinforcement stresses of M30 at load step 57.

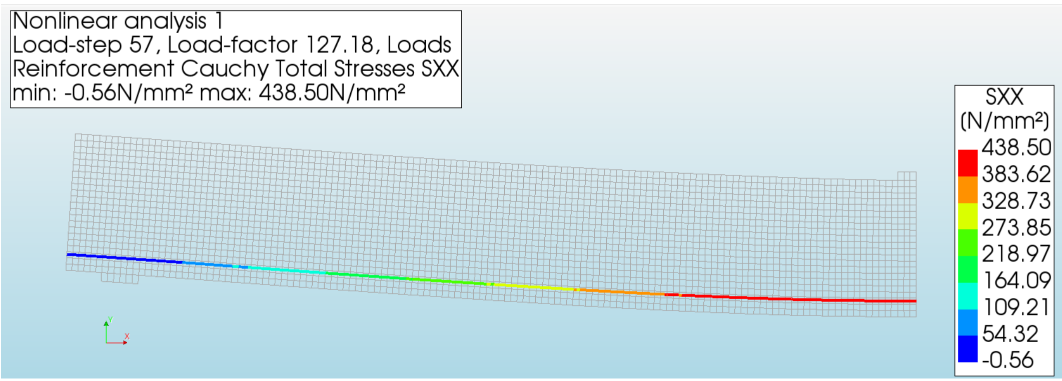



**Figure 10.15:** Case B1: Yielding of reinforcement bars M30 at step 57 (P = 254 kN)

The reinforcement steel M25 has a yielding strength of $f_{ym} = 445$ MPa. Figure 10.16 displays the reinforcement stresses of M25 at load step 59.

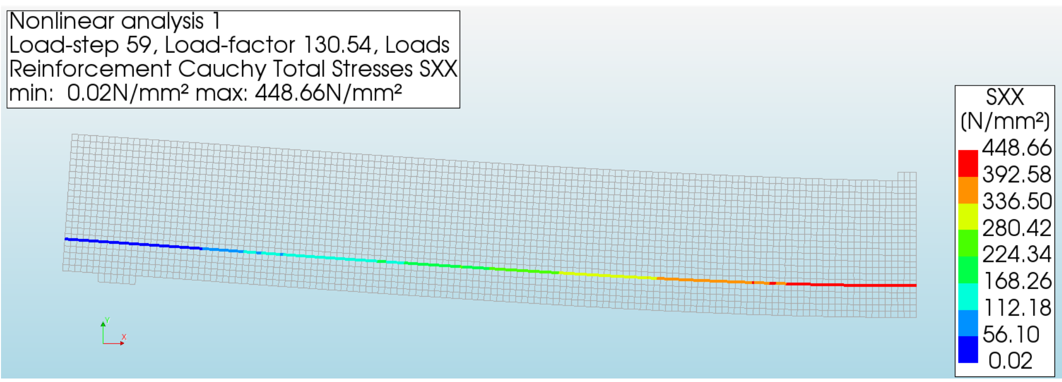



**Figure 10.16:** Case B1: Yielding of reinforcement bars M25 at step 59 (P = 261 kN)

### 10.4.5 Stress-strain curves of concrete

A stress-strain curve of concrete in tension, obtained from node 123, is shown in Figure 10.17. The dashed line indicates the post-peak behaviour.

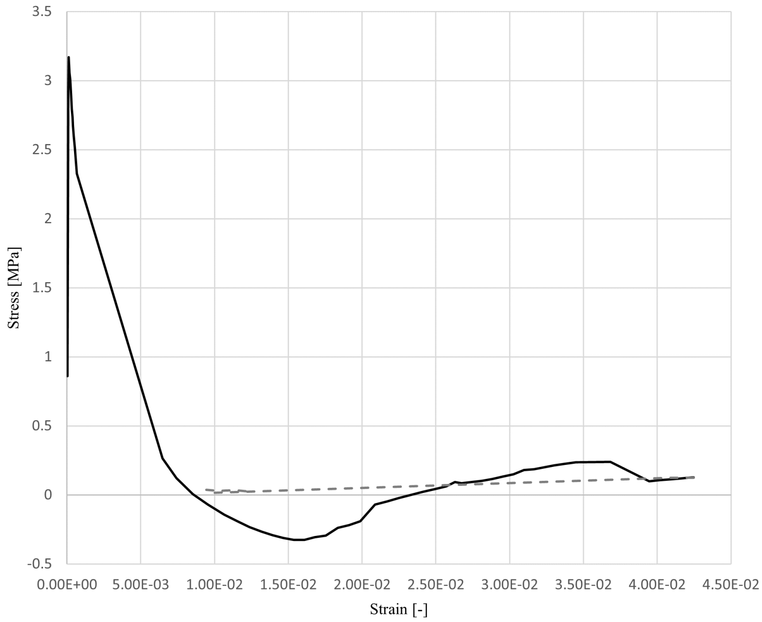



**Figure 10.17:** Case B1: Stress-strain curve of concrete in tension (node 123)

A stress-strain curve of concrete in compression, obtained from node 8967, is shown in Figure 10.18.

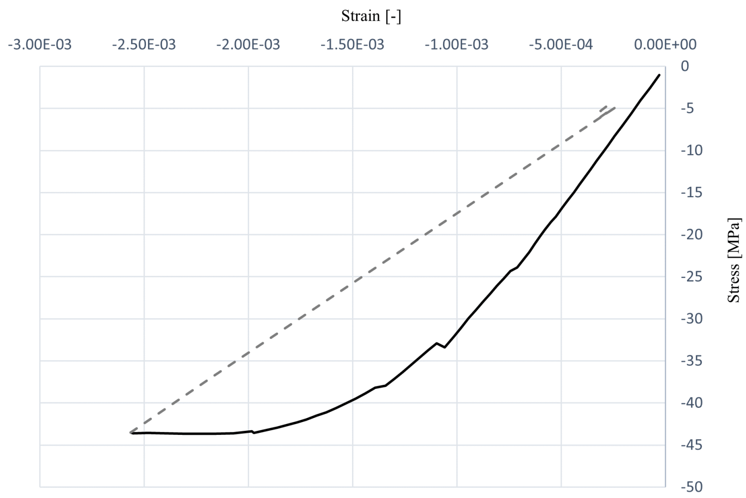



**Figure 10.18:** Case B1: Stress-strain curve of concrete in compression (node 8967)

### 10.4.6 Convergence behaviour

The convergence behaviour of the NLFEA is shown in Figure 10.19.

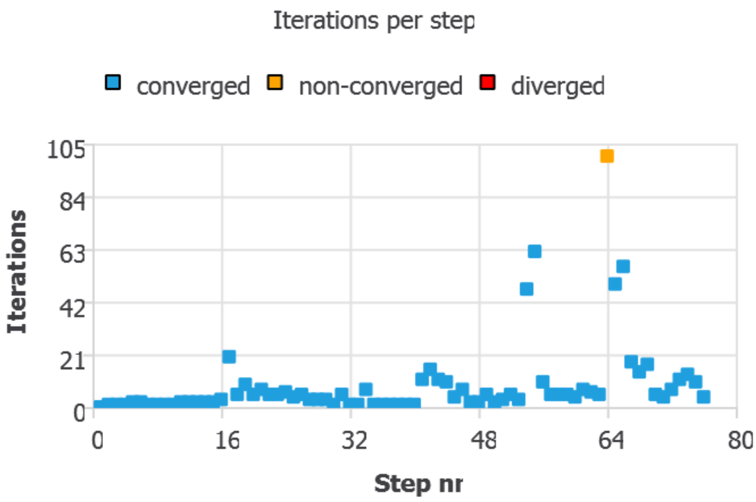



**Figure 10.19:** Case B1: Convergence of NLFEA

## 10.5 Discussion

The load-deflection curve, Figure 10.8, obtained from the NLFEA is in good agreement with the experimental results. A comparison of the load-deflection curves can be found in Figure 10.20. The peak and pre-peak response agrees very well with the experimental results. The post-peak response of the NLFEA is marked with a dotted line, due to the fact that few data points (load steps) are used to represent the shown post-peak behaviour. As can be seen from Figure 10.19, all load steps up to and including load step 63, which includes the peak load at load step 62, have converged within the chosen number of iterations.

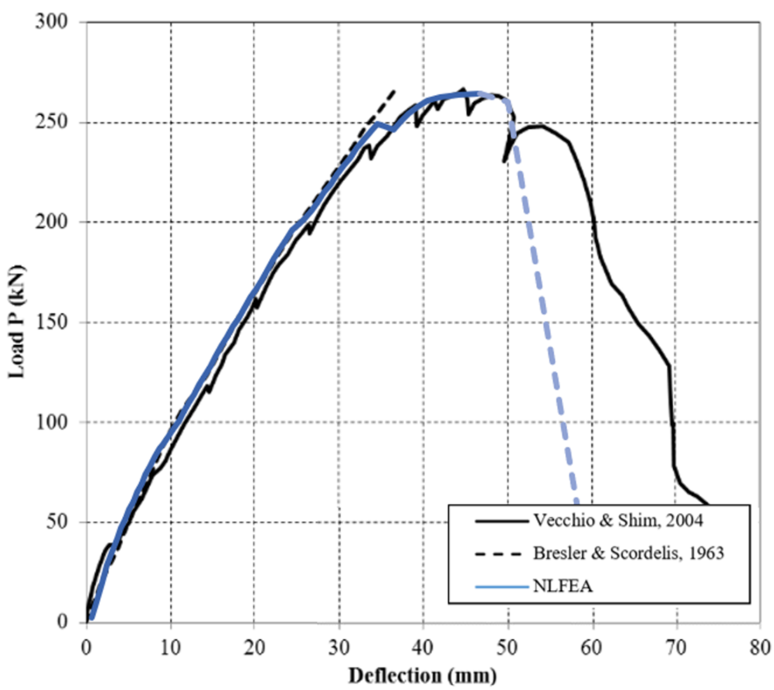



**Figure 10.20:** Case B1: Load-deflection curves for NLFEA and experimental results

The post-peak response of Figure 10.20 differs somewhat from the experimental results, and is modelled using very few load steps as shown in Figure 10.21. As seen in Figure 10.19, non-convergence occurs at load step 64. After this point the analysis fails, which can be seen from the sudden horizontal line in Figure 10.8 and Figure 10.21. The remaining load steps that can not be seen from Figure 10.21 lie on the same horizontal line. A more refined nonlinear analysis as well as an increased value for the maximum number of iterations might have captured the post-peak response even better. Furthermore, as the beam fails in bending with crushing of concrete, it is known that the value chosen for the compressive fracture energy, $G_C$, has a significant influence on the post-peak behaviour [25]. A parametric study of this parameter could be performed to find a more suitable value for this parameter, than the one calculated by default. The ductility of the beam is expected to increase if the value of the compressive fracture energy is increased. More of the default parameters might also need tweaking, in accordance with the specific experiment, in order to obtain a better modelling of the post-peak behaviour. The flexible structure of the framework, as well as the option to perform parametric studies allows for a closer inspection of the significance of the different input parameters.

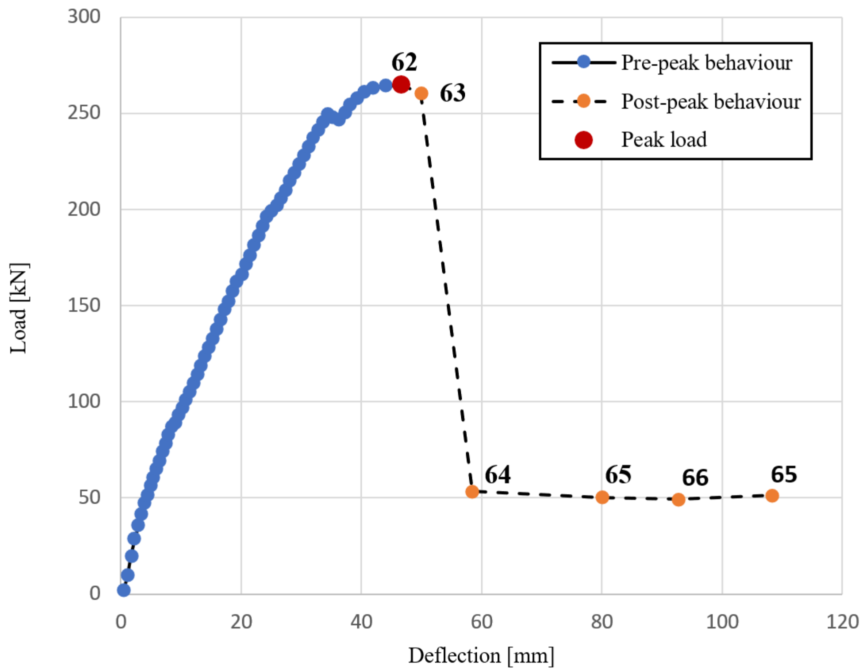



**Figure 10.21:** Case B1: Load steps marked on load-deflection curve at midpoint

The stress-strain curves presented in Section 10.4.5 exhibit the expected shapes up to and including the peak load. As the analysis fails shortly after this point, the results for the post-peak behaviour of the stress-strain curves are not valid and should be ignored. The post-peak behaviour is therefore marked with a dashed line. The pre-peak behaviour of the tensile curve, seen in Figure 10.17, has the expected exponential softening curve, after the expected tensile strength of 3.2 MPa has been reached. The compression curve has the expected parabolic shape and peaks at the expected maximum compressive strenght of 43.5 MPa. However, the softening branch of this curve is not modelled correctly, as the analysis fails shortly after reaching the peak load.

The modelled beam exhibits a flexural-compressive failure mode as expected. The different reinforcement types yield at the expected stress, as shown in Section 10.4.4, and all reinforcement types experience yielding at peak load as expected. Crushing of the concrete at peak load can also be observed from Figure 10.12. The crushing of the concrete occurs near the load application area, as expected from Figure 10.3.

The first cracks appear at midspan on the bottom the beam where the tensile forces are highest, as expected. However, at the point where DIANA first detects cracking, these values are too small to be observed in the experimental setup. Therefore, the point of cracking of concrete which is marked on the load-deflection graph, Figure 10.8, has been set to the point where the crack widths are likely to be detected in the experiment. The computed crack pattern shown in Figure 10.10 and Figure 10.11 match well with the expected crack pattern. The computed crack pattern at failure is remarkably similar to the experimental observation, as shown in Figure 10.22. The failure mechanisms are in other words as expected, and similar to that of the experimental results. It can therefore be concluded that the chosen total strain rotating crack model for this beam is indeed a suitable crack model. However, an even more correct model of the crack pattern for this specific beam could have been achieved by defining discrete cracking at midpoint, combined with smeared cracking with a rotating TSCM for the continuous part of the beam [15].

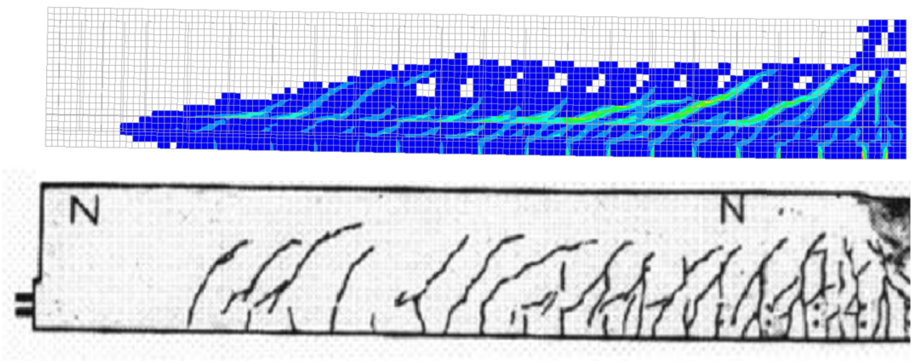



**Figure 10.22:** Case B1: Numerical and experimental crack patterns at failure

The element size, $l$, for the finite element model of B1 is set to 25 mm for all elements. This gives 22 elements over the height of the beam, which are more than enough according to the recommendations, see Section 2.2.2.1. The selected element size achieves mesh independence. The mesh independence has been checked by performing an analysis with elements of size $l/2$. Furthermore, a check of the correct modelling of a symmetric beam has been performed, by also modelling the whole beam. The same results were obtained from modelling the whole beam and half of the beam. However, as expected, modelling the whole beam resulted in a much more time-consuming analysis. Modelling only half the beam in case of symmetry is therefore the preferred option and a useful feature of the framework.

All in all, as mainly default properties have been used for the NLFEA, it can be concluded that the default properties ensure a robust NLFEA of Case B1. The peak load and failure mechanisms agree very well with the expected results. The recommended crack model also proves to be suitable for the type of beam presented in Case B1. Evidently, some tweaking of different parameters are needed for an even better agreement between the results of the NLFEA and the experimental results. However, the default parameters of the framework provide a great starting point for robust NLFEA.

# 11 Case B2: Collins and Kuchma (1999)

Case B2 is based on Beam SE-50A-45 from the experimental program of Collins & Kuchma from 1999. It is a four-point bending test of a RC beam which fails in shear. The beam exhibits a diagonal-tension failure mode [1].

## 11.1 Experimental setup and results

### 11.1.1 Geometry and loading

The geometry of the beam and the reinforcement, as well as the loading, is shown in Figure 11.1. As can be seen from Figure 11.1, extra longitudinal reinforcement bars have been added to the regions characterised by the maximum value of the applied bending moment. These sections are denoted B-B and C-C. The cross section of the beam is shown in Figure 11.2 for section A-A, B-B and C-C, which are all marked in Figure 11.1. The length of the beam is 5 m, the height is 0.5 m and the width is 0.169 m. The beam is subjected to four-point bending, where the right load is twice that of the left load. The longitudinal reinforcement has a diameter of 16 mm [27].

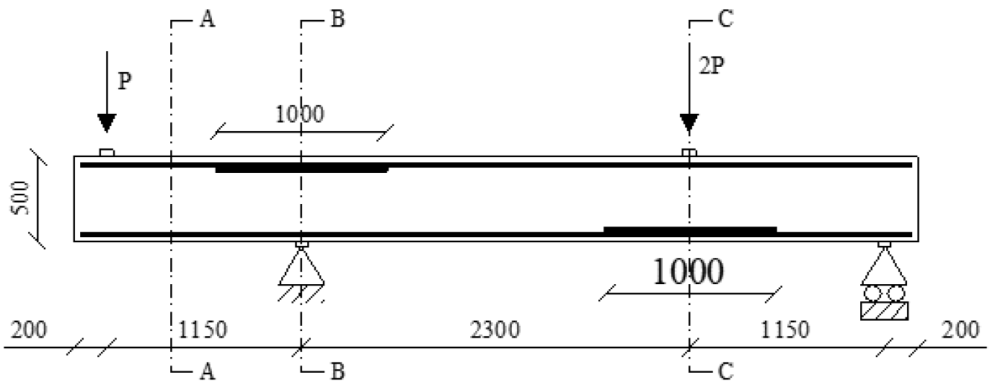



**Figure 11.1:** Case B2: Experimental setup. Dimensions in mm [27]

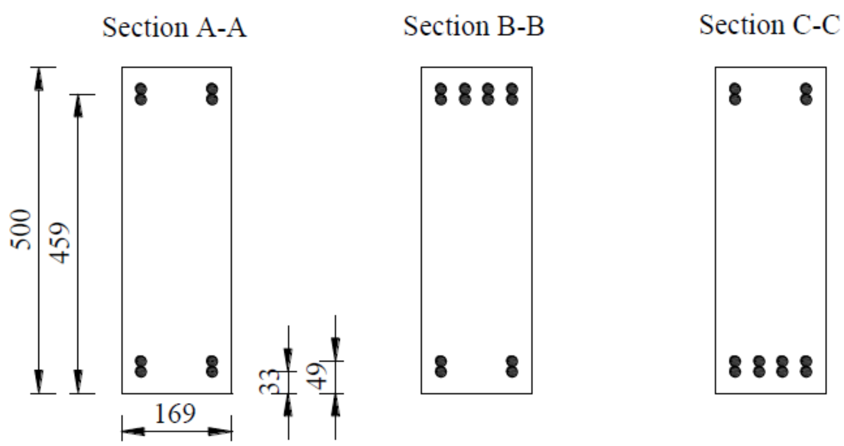



**Figure 11.2:** Case B2: Cross sections [25]

### 11.1.2 Material properties

Table 11.1 lists the most important material properties of the concrete and the steel. The same reinforcement steel has been used for all longitudinal reinforcement. The beam has no transverse reinforcement. The given parameter values for the reinforcement in Table 11.1 are for one single bar.

**Table 11.1:** Case B2: Material properties [25][27]

| Material | Parameter | Value |
|---|---|---|
| Concrete | Compressive strength, $f_{cm}$ | 53 MPa |
| Reinforcement steel | Diameter, $\phi$ | 16 mm |
| | Cross section area, $A_s$ | 200 mm$^2$ |
| | Young's modulus, $E$ | 200000 MPa |
| | Yielding strength, $f_{ym}$ | 400 MPa |
| | Ultimate strength, $f_{um}$ | 600 MPa |
| | Ultimate strain, $\varepsilon_{su}$ | 0.05 |
| Steel for plates | Young's modulus, $E$ | 200000 MPa |
| | Poissons's ratio, $\nu$ | 0.3 |

### 11.1.3 Experimental results

The beam was tested twice, resulting in two different failure loads. The experimental ultimate value of the applied load before failure was $P_{exp,1} = 69$ kN for the first test, and $P_{exp,2} = 81$ kN for the second test [25]. The beam experienced a diagonal-tension failure mode. This is a brittle failure, which means that the shear collapse occurs suddenly with little to no warning. The failure mechanism of Case B2 is shown in Figure 11.3, and is a typical failure mechanism for a beam with no shear reinforcement. The load-deflection curve is not given in the references [25].

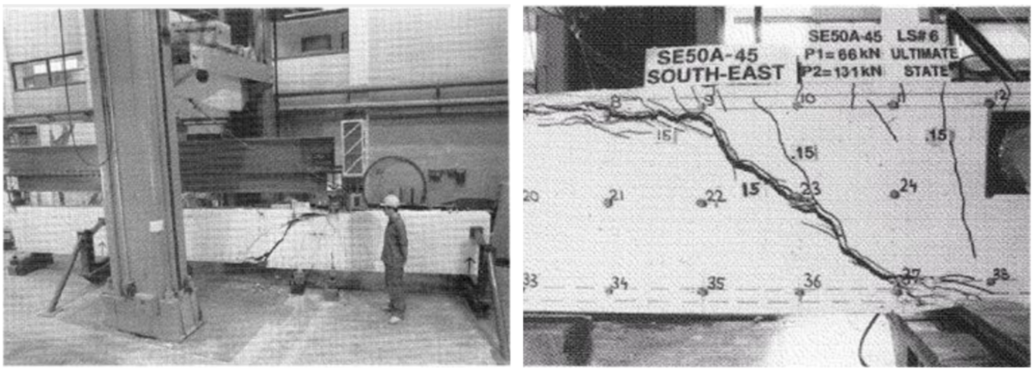



**Figure 11.3:** Case B2: Failure mechanisms [25]

## 11.2 Finite element model

### 11.2.1 Geometry and loading

The finite element model has been generated by changing the parameters in Script A in accordance with Table 11.1 and the geometry presented in Section 11.1. The created input script for Case B2 can be found in Appendix F. Longitudinal reinforcement have been added, and the options to create the loads and plates individually have been chosen in Script A. The model is shown in Figure 11.4. Dead weight loading has been applied in a single step as the first load case. In load case 2, the point loads have been applied. An initial point load of P = 1 kN has been applied at the middle of the left loading plate, and a point load of 2P = 2 kN has been applied at the middle of the right loading plate.

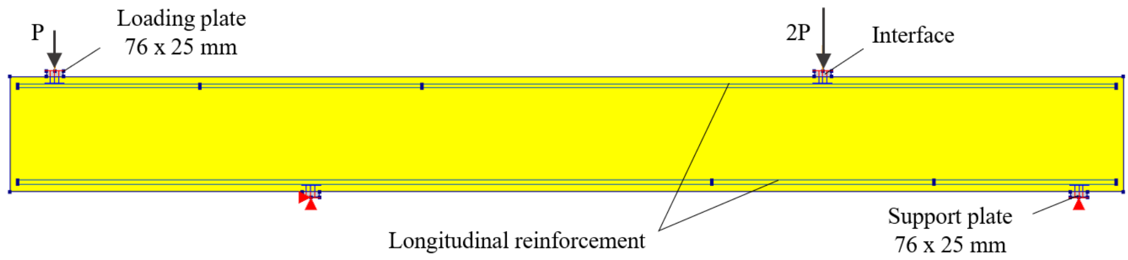



**Figure 11.4:** Case B2: View of the model

### 11.2.2 Material properties

Based on the given compressive strength, the concrete class is set to C45. The material properties which are not given in Table 11.1, have been estimated according to EC2 and MC2010 with reference to the concrete class. A total strain fixed crack model has been used. Table 11.2 presents the input for the arguments of the concrete material model.

**Table 11.2:** Case B2: Input of concrete material model

| | |
|---|---|
| Young's modulus, $E_{cm}$ | 36283 N/mm$^2$ |
| Poisson's ration, $\nu$ | 0.2 |
| Mass density | 2.4e-09 T/mm$^3$ |
| Crack model | TSCM |
| Crack orientation | Fixed |
| Tensile curve | Exponential |
| Tensile strength, $f_{ctm}$ | 3.8 N/mm$^2$ |
| Tensile fracture energy, $G_F$ | 0.149 N/mm |
| Crack bandwidth specification | Govindjee |
| Possion's ratio reduction model | Damage based |
| Compression curve | Parabolic |
| Compressive strength, $f_{cm}$ | 53 N/mm$^2$ |
| Compressive fracture energy, $G_C$ | 31.66 N/mm |
| Compressive strength reduction | Vecchio and Collins 1993 |
| Lower bound of reduction curve | 0.6 |
| Stress confinement model | Selby and Vecchio |
| Shear retention model | Damage based |

Embedded reinforcement has been used to model the longitudinal reinforcement. Eight reinforcement objects have been created. Since a 2D model has been used, each reinforcement object has been defined with a cross section equal to 2 × 200 mm$^2$ = 400 mm$^2$. A Von Mises Plasticity model has been used, and the properties used to define the bilinear stress-strain diagram for the reinforcement are given in Table 11.1. Figure 11.5 shows the stress-strain curve used for the reinforcement.

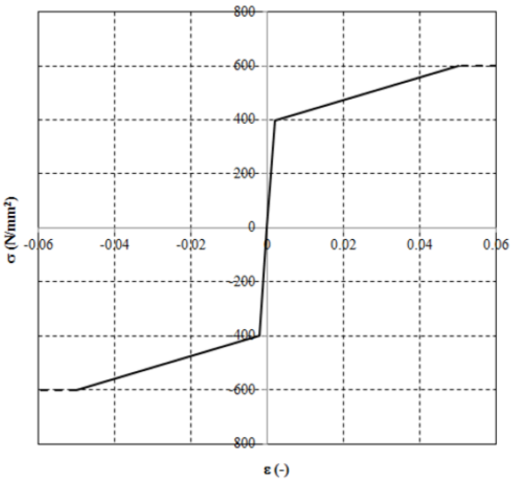



**Figure 11.5:** Case B2: Stress-strain curve for reinforcement [25]

For the steel plates, a linear elastic behaviour is assumed. The properties for the steel plates are given in Table 11.1.

Structural interface elements have been defined between the steel plates and the concrete beam. The interface properties have been taken from the validation studies by Rijkswaterstaat [25]. The chosen interface properties are presented in Table 11.3.

**Table 11.3:** Case B2: Interface properties

| $K_{nn}$ in tension | $K_{nn}$ in compression | $K_t$ |
|---|---|---|
| 3.63 e-08 N/mm$^3$ | 3.63 e+04 N/mm$^3$ | 3.63 e-08 N/mm$^3$ |

### 11.2.3 Mesh

The mesh for B2 is presented in Figure 11.6. The element size is set to 25 mm. Otherwise, default properties have been used for the mesh, these are summarised in Table 11.4.

**Table 11.4:** Case B2: Mesh properties

| | |
|---|---|
| Mesher type | Quadrilateral |
| Mesh order | Quadratic |
| Element type | Plane Stress |
| Element name | CQ16M |
| Integration scheme | 3×3 Gaussian |

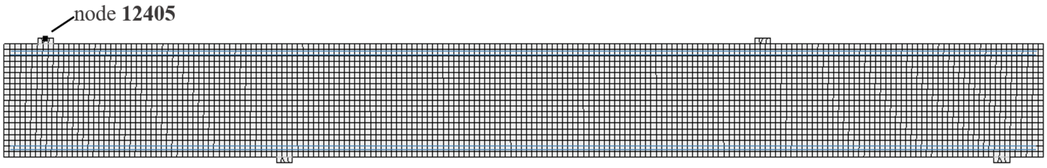



**Figure 11.6:** Case B2: Finite element mesh

The node marked on Figure 11.6 will be referred to in Section 11.4: Results of NLFEA.

## 11.3 Structural nonlinear analysis

The default properties for the nonlinear analysis have been kept, except for the number of steps and the lower limit for the step size, which have been decreased. The option in Script A with the suggested tolerances for the force norm and energy norm has been chosen for the convergence criteria. Table 10.5 presents the input for the arguments of the NLFEA. The analysis has been performed until failure.

**Table 11.5:** Case B2: Properties of NLFEA

| | |
|---|---|
| Incrementation method | Energy based adaptive loading |
| Step size | Energy based |
| Factor for first load increment | 5 |
| Upper limit of step size | 10 |
| Lower limit of step size | 0.5 |
| Number of steps | 130 |
| Iteration method | Regular Newton-Raphson |
| Maximum number of iterations | 100 |
| Analysis control | Arc-length + Line search |
| Convergence norms | $F0.05$, $E0.01$ (load case 1)<br>$F0.01$, $E0.001$ (load case 2) |
| All convergence norms have to be satisfied | False |
| Continue if no convergence | True |

## 11.4 Results of nonlinear finite element analysis

### 11.4.1 Load deflection

The NLFEA gives a peak load of **90** kN. In other words, P = 90 kN and 2P = 180 kN. The load-deflection curve at the left loading plate, node 12405, is presented in Figure 11.7. The load value corresponding to cracking of concrete is marked, and dotted lines have been used to indicate the values of the experimental failure loads.

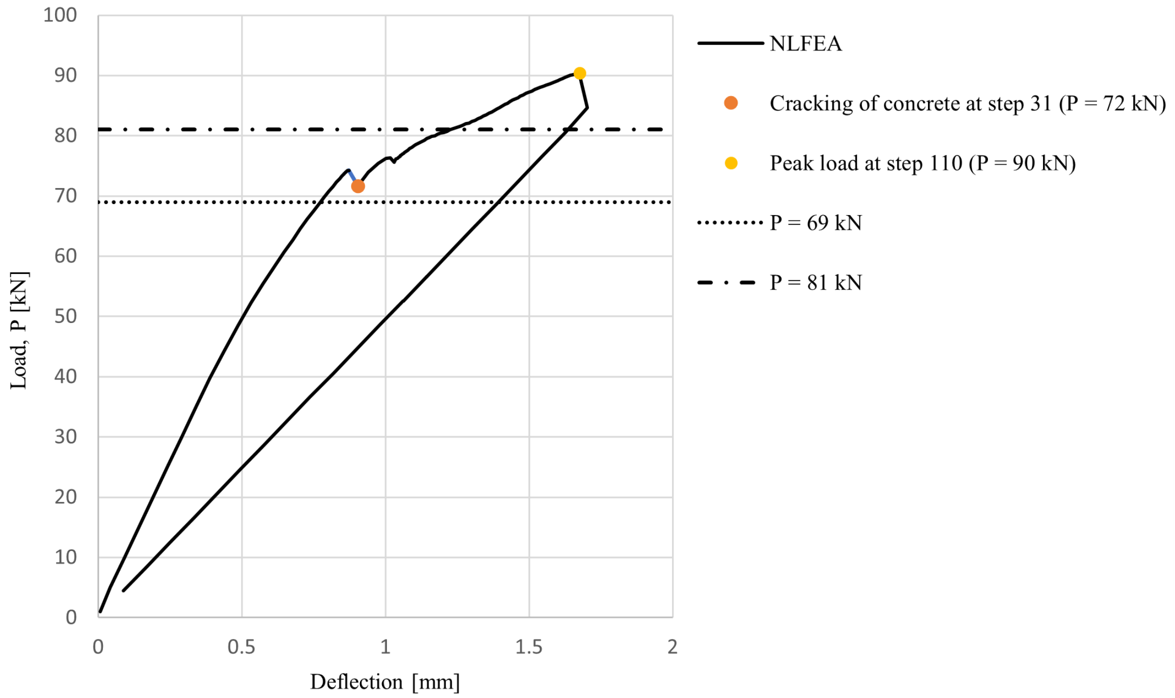



**Figure 11.7:** Case B2: Load-deflection curve (node 12405)

### 11.4.2 Cracking

The following figures show the crack widths at three loading points: just after the start of the crack localisation (which corresponds to the first local decrease of the load-deflection graph), at the maximum loading and at the last load step already in the post-peak regime.

Figure 11.8 corresponds to the point of cracking marked in Figure 11.7. Figure 11.8 shows the crack widths at load step 31, here the applied load, P, is 72 kN. At this load step the crack widths lay within the range of 0.01 - 0.17 mm.

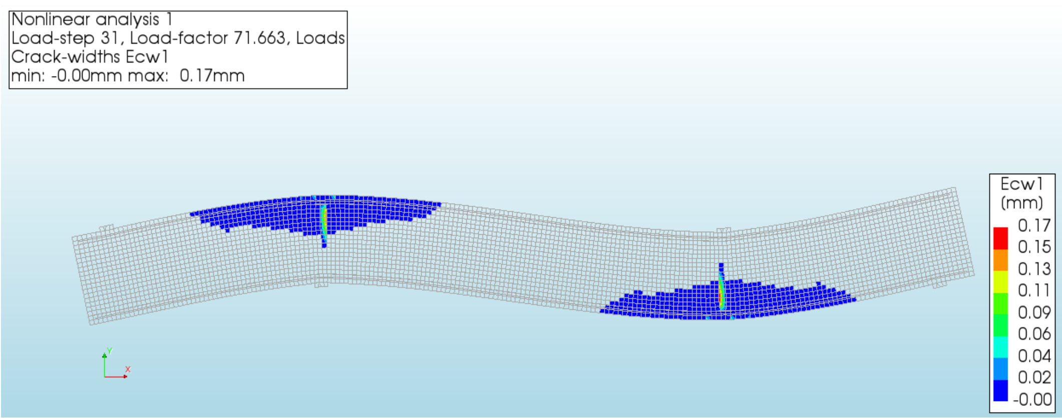



**Figure 11.8:** Case B2: Crack widths at load step 31 (P = 72 kN)

Figure 11.9 presents the crack widths and pattern for the peak load at load step 110.

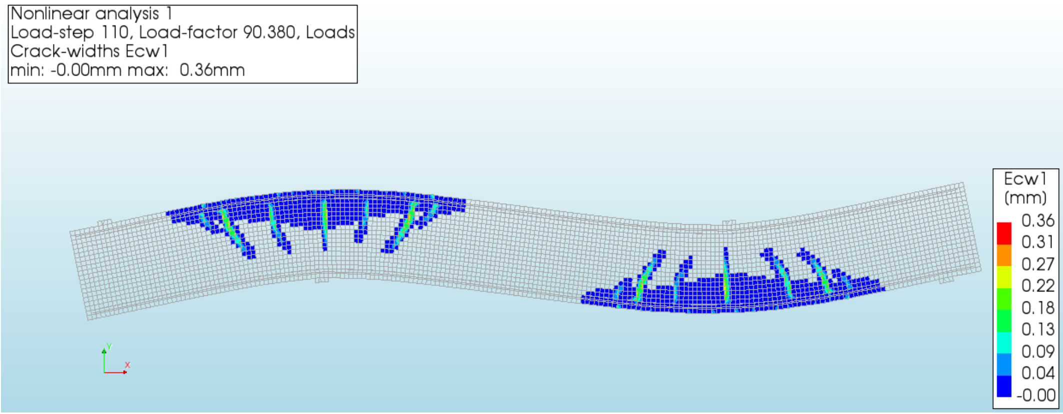



**Figure 11.9:** Case B2: Crack widths at load step 110 (P = 90 kN)

Figure 11.10 shows the crack widths and pattern at load step 131.

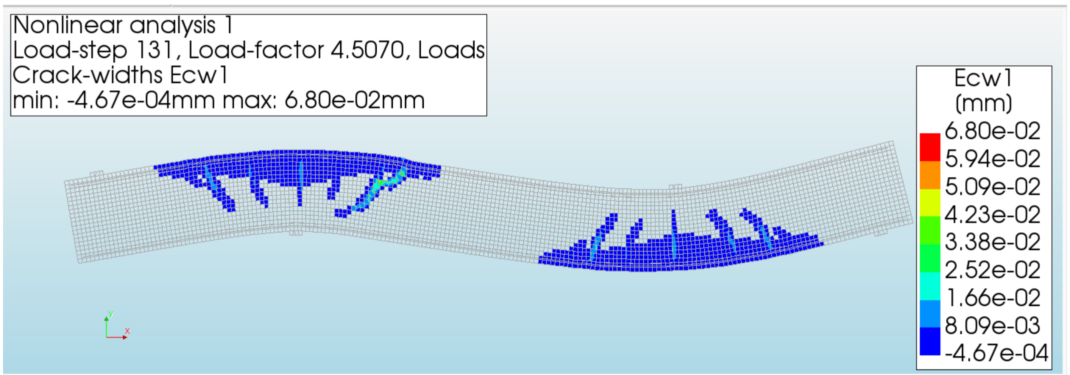



**Figure 11.10:** Case B2: Crack widths at load step 131

### 11.4.3 Minimum principal stress

Figure 11.12 shows the minimum principal stresses in the concrete beam at peak load. Negative stresses indicate compression.

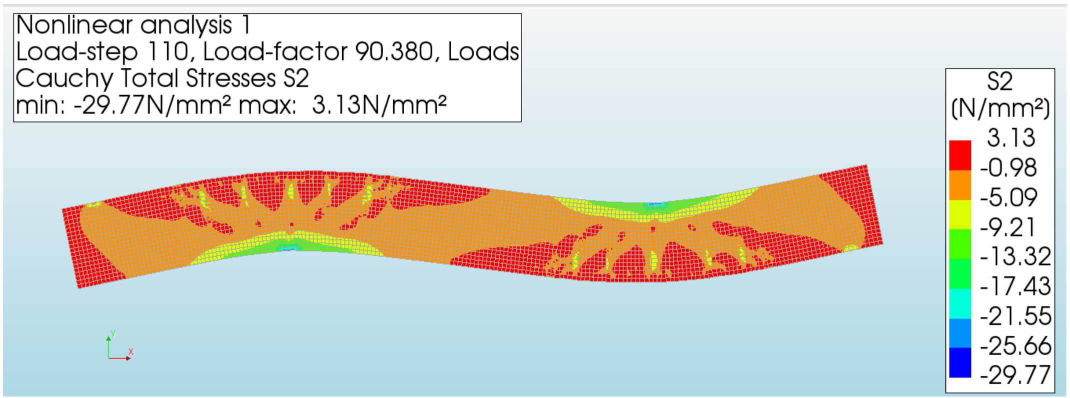



**Figure 11.11:** Case B2: Minimum principal stresses at load step 110 (P= 90 kN)

### 11.4.4 Reinforcement stresses

The reinforcement does not yield. Figure 11.12 shows the stress distribution in the reinforcement at load step 110, when the peak load is reached.

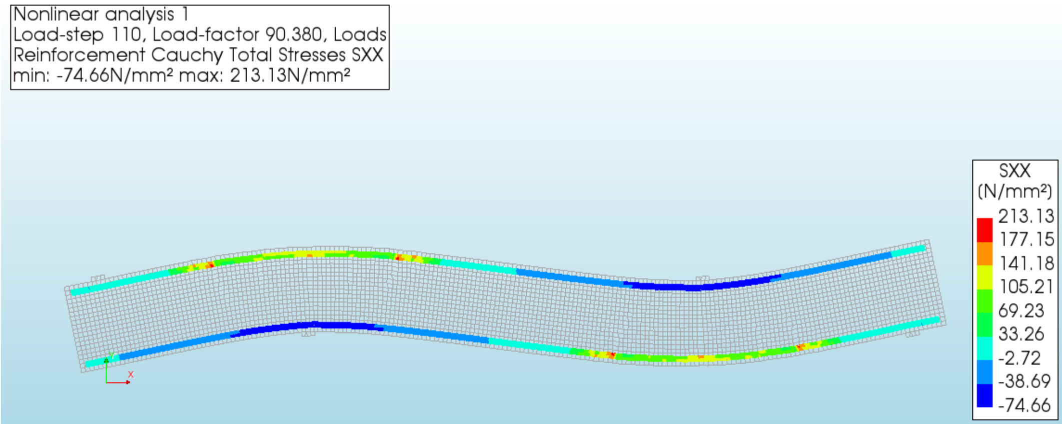



**Figure 11.12:** Case B2: Reinforcement stresses at load step 110 (P = 90 kN)

### 11.4.5 Convergence behaviour

The convergence behaviour of the nonlinear analysis is shown in Figure 11.13.

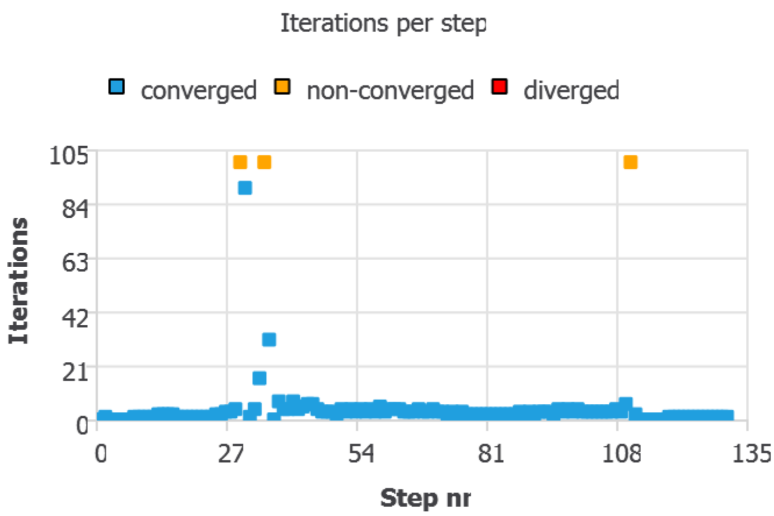



**Figure 11.13:** Case B2: Convergence of NLFEA

## 11.5 Discussion

As can be seen in Figure 11.7, the NLFEA overestimates the peak load slightly. Even so, the outcome of the analysis and the failure mechanisms are as expected. The beam failed in diagonal tension, and as can be seen from the load-deflection curve presented in Figure 11.7, no ductility is displayed after the peak load has been reached, due to the brittle nature of this failure.

The crack propagation displayed in Section 11.4.2 is in agreement with the expected failure mode. The initial horizontal bending cracks starts to localise in load step 31, as shown in Figure 11.8, which also corresponds to the local snap-back in the load-deflection curve shown in Figure 11.7. After this load step, additional loading results in increasing opening of more bending cracks that in time transform into diagonal shear cracks. Figure 11.10 clearly shows how the critical crack propagates rapidly after the peak load has been reached and continues as a large diagonal shear crack towards the end of the beam. The typical 45° crack angle that is caused by diagonal tension as a result of shear stresses, is marked on Figure 11.14 with red lines.

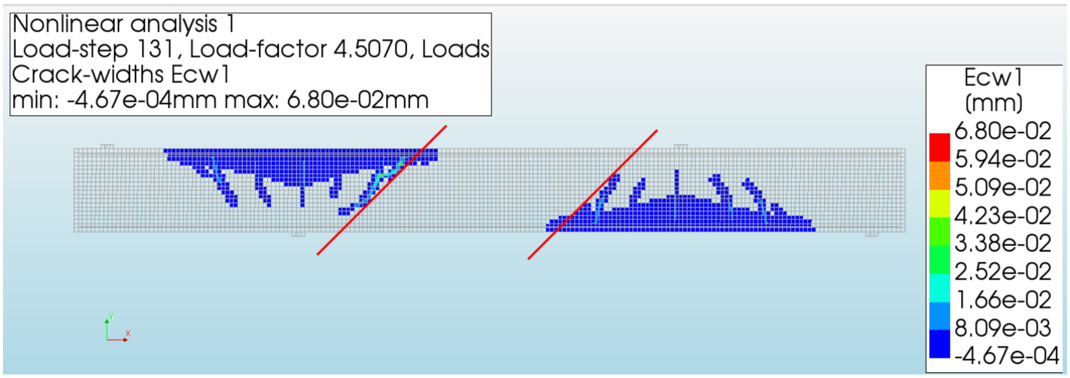



**Figure 11.14:** Case B2: Crack widths at load step 131 with lines indicating the 45 degree crack angle

The convergence of the NLFEA is presented in Figure 11.13. Convergence is achieved for the peak load at load step 110. The nonconvergence is caused by local cracking effects in load step 30, 35 and 111, and does not indicate failure. As can be seen from Figure 11.13, each case of nonconvergence is followed by a converged step.

In accorance with the expected failure mechanism, the shear failure was not accompanied by crushing of the concrete, which can be seen from the minimum principal stresses reported in Figure 11.11. All stresses shown in Figure 11.11 have a higher value than the concrete compressive strength of -53 MPa. The reinforcement does also not experience yielding, as flexural failure is not the governing failure mechanism. Figure 11.12 shows how all reinforcement stresses are below the yield strength of 400 MPa. Since the reinforcement does not experience yielding, a linear elastic material model could have been assumed for the reinforcement instead of the bilinear diagram that was chosen. However, as of now, the only implemented material model for the reinforcement in the Beam Script is the Von Mises Plasticity model.

Since the beam fails due to diagonal tension, the results of the NLFEA are heavily dependent on the chosen crack model, tensile strength and tensile fracture energy, $G_F$ [25]. It may seem as though a fixed crack model slightly overestimates the results, and it would have been interesting to model the same beam using a rotating crack model. At the current stage, the Beam Script does not allow the user the change the crack model. In the future, this feature could be implemented, allowing the user to perform a parametric study on the use of different crack models.

Furthermore, the validation studies by Rijkswaterstaat shows that the value of the crack bandwidth has a significant influence on the load-deflection curve of Case B2 [25]. The reason for this might be related to the control procedures or convergence criteria. Figure 11.15 shows the load-deflection curves for case B2 obtained by using three different values for the crack-bandwidth. Based on the curves presented in Figure 11.15, it can be assumed that a lower peak load, which would have been in even better agreement with the experimental peak loads, might have been achieved using a higher value for the crack bandwidth than the one estimated using Godvindjee's method [25].

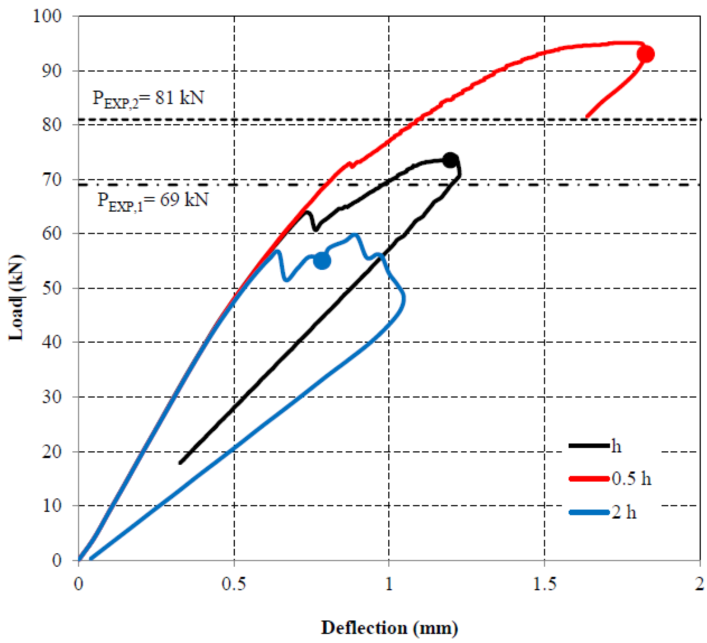



**Figure 11.15:** Case B2: Load-deflection curves obtained for different crack bandwidth values[25]

Overall, the framework provides a good starting point for performing a NLFEA of Case B2, and the speed, convergence and quality of the analysis are ensured by the provided default properties. The expected crack pattern is obtained using the framework, however it seems as though the chosen fixed crack model might overestimate the peak load slightly. In future work, the framework could be extended to include amongst others the option to choose a linear material model for the reinforcement steel and an option to switch between the fixed and rotating crack model.

# Part V

# Parametric study

# 12 General information

The framework allows for the execution of a parametric study, where one of the input parameters can be varied. To demonstrate this feature, a very simple parametric study has been performed, where the length of a RC beam has been varied from 3m to 6m. However, using the Beam Script to perform a parametric study is not limited to changing the geometry, and could also be performed for all input parameters, both required and optional, of Script A. For example could the tensile curve for the concrete material model or the crack bandwidth specification be varied when performing the parametric study.

It is well-known that the results of a NLFEA can be substantially influenced by the modelling choices and input parameters [1]. Even small changes of the value of one of the input parameters, for example the value of the fracture energy as mentioned in Part IV, might have a noticeably effect on the results. Therefore, even though default values for the properties are provided in Script A, using Script B to perform sensitivity studies is a useful and arguably imperative feature of the framework. Furthermore, even though nonlinear analysis is allowed in EC2, the analysis is required to contain relevant parameter studies to demonstrate that the model can appropriately cover all relevant failure modes [1]. Script B makes this possible.

Evidently, the user could perform a parametric study without running Script B, by creating a number of input files, changing one single parameter, and running Script D for each of the created input files. However, Script B allows the user to do this in a much more efficient manner. Utilizing Script B, the user only have to define the user input and the values for the parametric study once. After clicking run Script B in DIANA, the user no longer has to interact with the program. The parametric study will be carried out by itself.

A parametric study where the length of a beam has been varied, will be presented and discussed in subsequent sections. The main focus of Part V is to demonstrate how a parametric study could be executed using the framework and how results of a parametric study could be presented. In other words, the geometry of the beam has been chosen at random, and the results of the parametric study have no real purpose. As the parameter to be varied is the main focus in a parametric study, the input for the parameters to be kept constant will not be presented to the same extent as the input for the benchmark studies in Part IV.

The performed parametric study on beam length will be covered in the following sections, describing respectively: the geometry and material properties of the selected beam, the finite element model, the results of the parametric study and a discussion.

# 13 Parametric study on beam length

## 13.1 The beam

### 13.1.1 Geometry and loading

The geometry of the chosen beam and the reinforcement, as well as the loading, are shown in Figure 13.1 and Figure 13.2. The length of the beam is to be varied, the height is 0.4 m and the width is 0.25 m. The beam is subjected to three point bending, as shown in Figure 13.1. The geometry is chosen at random, and has not been designed so as to fulfil any EC2-specifications.

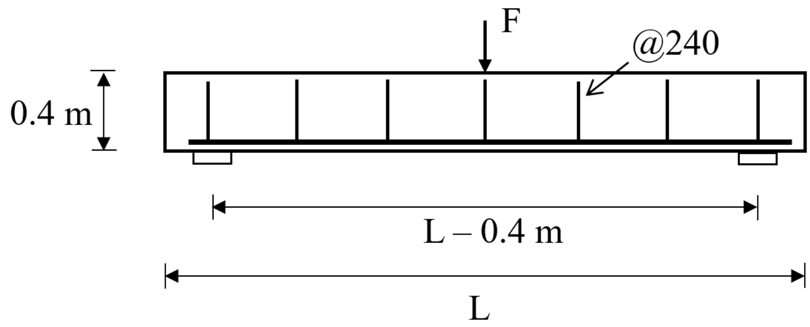



**Figure 13.1:** Geometry and setup

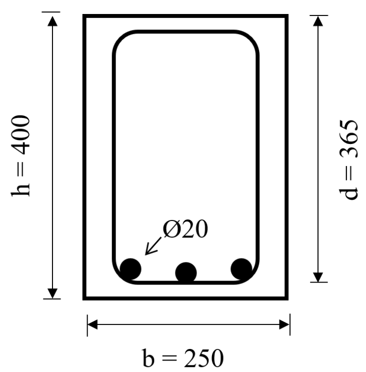



**Figure 13.2:** Cross section (dimensions in mm)

### 13.1.2 Material properties

Table 13.1 lists the most important material properties of the concrete and the steel. The given parameter values for the longitudinal reinforcement in Table 10.1 are for one single bar.

**Table 13.1:** Material properties

| **Material** | **Parameter** | **Value** |
|---|---|---|
| Concrete | Characteristic compressive strength, $f_{ck}$ | 30 MPa |
| Reinforcement steel B500C (longitudinal) | Diameter, $\phi$ | 20 mm |
| | Cross section area, $A_s$ | 314 mm$^2$ |
| | Young's modulus, $E$ | 200000 MPa |
| | Yielding strength, $f_{yk}$ | 500 MPa |
| | Ultimate strength, $f_{uk}$ | 540 MPa |
| | Ultimate strain, $\varepsilon_{uk}$ | 0.05 |
| Reinforcement steel D4 (stirrups) | Diameter, $\phi$ | 3.7 mm |
| | Cross section area, $A_s$ | 25.7 mm$^2$ |
| | Young's modulus, $E$ | 220000 MPa |
| | Yielding strength, $f_{ym}$ | 600 MPa |
| | Ultimate strength, $f_{um}$ | 651 MPa |
| | Ultimate strain, $\varepsilon_{um}$ | 0.047 |
| Steel for plates | Young's modulus, $E$ | 200000 MPa |
| | Poisson's ratio, $\nu$ | 0.3 |

## 13.2 Finite element model

The view of the model is shown in Figure 13.3. A template for a three point bending test has been used. Dead weight loading has been applied in a single step as the first load case. In load case 2, an initial point load of P = 1 kN has been applied at the midpoint of the loading plate. The applied point load, P, is marked in Figure 13.3. Appendix G presents the input script created for this parametric study. The length of the beam has been defined as the parameter to be varied in the parametric study, as shown in line 1 of Script 13.1. The chosen values for the beam length are 3m, 4m, 5m and 6m, as shown in line 4 of Script 13.1. Appendix H presents how these values have been input in Script B. The concrete class is set to C30. The material properties have been estimated with reference to the concrete class. A total strain rotating crack model has been chosen by default for the beam. The span length has been defined as dependent on the beam length. For further information about the constant parameters used in the parametric study, see Appendix G.

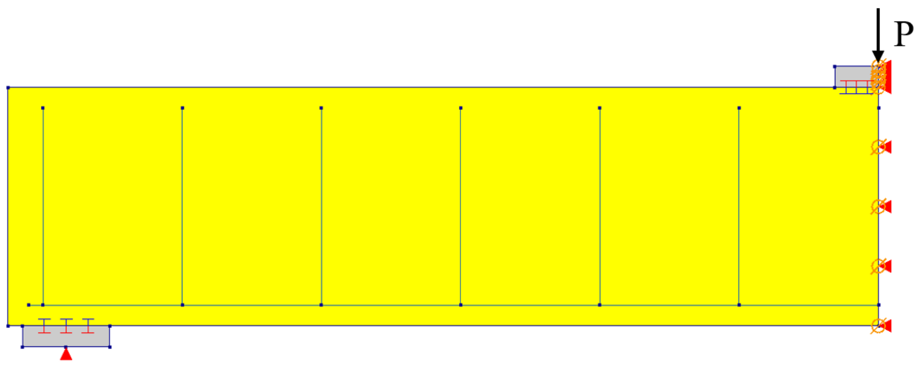



**Figure 13.3:** View of the model

```
beam.geometry.length = parametricValue #[mm]

##INPUT LIST OF PARAMETRIC VALUES HERE:
parametricStudy([3000,4000,5000,6000])
```

**Script 13.1:** Input for parametric study

## 13.3 Nonlinear finite element analysis

Default values and properties have been used for the NLFEA, except for the number of steps which has been reduced to 100. Please see Table 8.5 for more information about the default properties of the NLFEA.

## 13.4 Results of parametric study

Figure 13.4 shows the load-deflection curves at midpoint of the same beam with varying length. The peak load is marked.

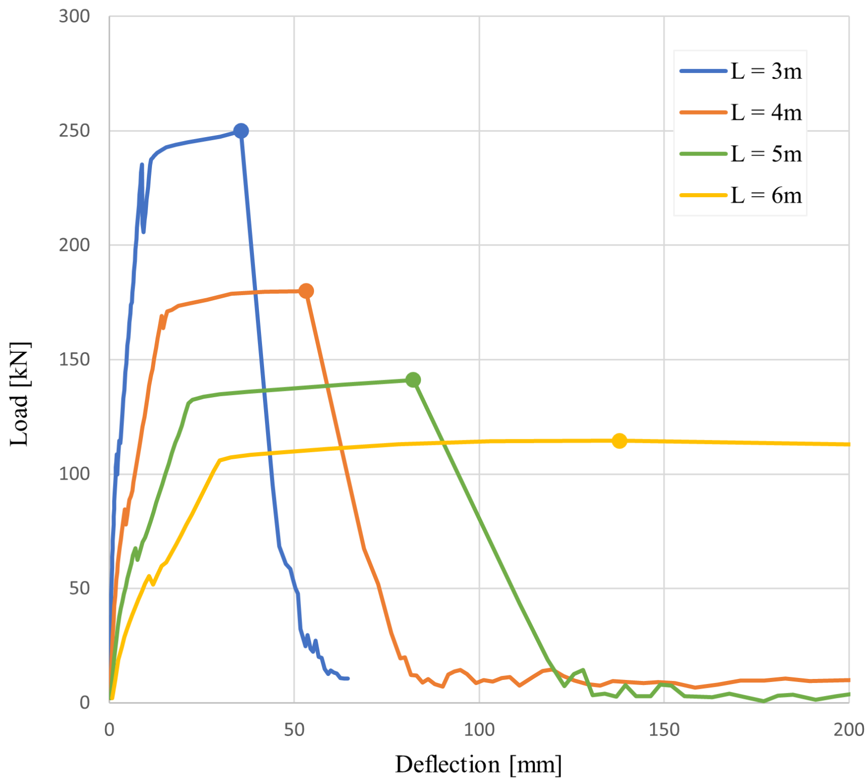



**Figure 13.4:** Load-deflection curves with varying beam length (midpoint)

A simple sectional analysis has been used to decide the critical loads for the different beam lengths. The calculations can be found in Appendix I. In Figure 13.5, the obtained peak loads using NLFEA are compared with the analytical results for the critical loads.

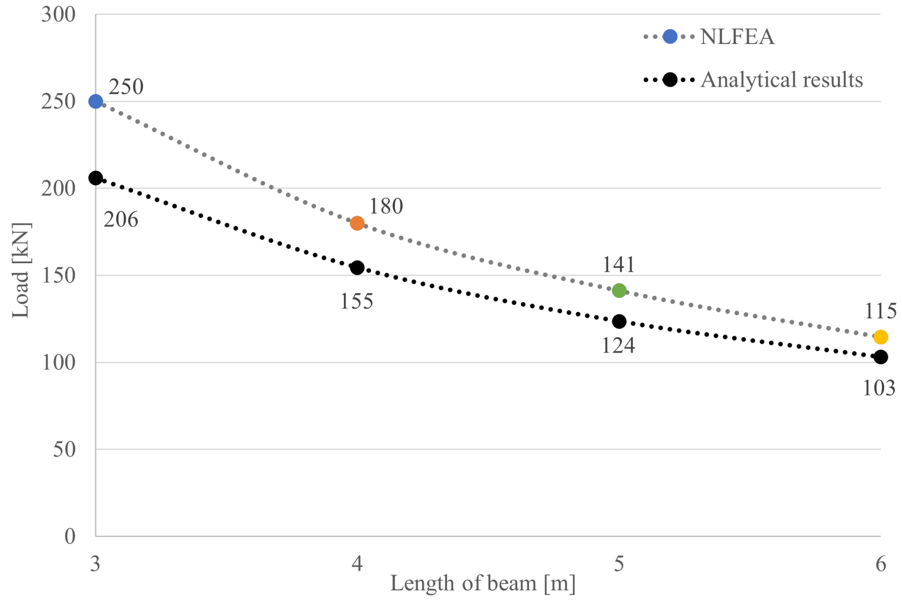



**Figure 13.5:** Critical load values for the different beam lengths

## 13.5 Discussion of parametric results

The main purpose of Section 13.4 is to demonstrate possible ways to present the results from a parametric study. Figure 13.4 displays a comparison of the load-deflection curves obtained when varying the beam length. The convergence has not been checked for this parametric study, however this should be done to evaluate the quality of the analysis. If the analysis for example fails after the peak load has been reached, the post-peak results are not valid. As can be seen from Figure 13.4, increasing the beam length results in a lower peak load and increased deflection. This is expected, as a longer beam will experience at larger bending moment at midspan than a shorter beam.

In general, by comparing the load-deflection curves from a parametric study, observations can be made on how the varied parameter affects the results of the NLFEA. The effect on the peak load can for example be studied. Furthermore, by looking at the load-deflection curves it could be evaluated to which degree the failure mode is dependent on the studied parameter. The role the parameter plays on the ductility of the beam could also be evaluated. Essentially, the sensitivity of the performed NLFEA in regards to different parameters could be studied. Nonlinear analysis is complex and it might be hard to know, especially with limited expertise, which factors that influence the analysis the most. Script B of the framework can be utilized for this purpose.

Figure 13.5 displays a possible way to compare analytical or experimental results with the results of a NLFEA. In Figure 13.5, the peak load obtained for the different beam lengths is compared to the calculated critical loads. The trend of both curves are similar, which supports that the NLFEAs have been carried out correctly up to and including the peak load. As expected, the failure loads obtained from the NLFEAs are higher than the analytical results. The nonlinear analysis is expected to give a higher load capacity, as the nonlinear behaviour and effects of the reinforced concrete is taken into consideration. In other words, the NLFEA allows for a more optimal design of the RC beam than 'convential' sectional analysis. The created framework is therefore a useful addition to the approach for more optimal design of reinforced concrete structures, since a robust NLFEA can be executed in an efficient and user-friendly manner.

The option in Script A to generate load-displacement CSV-files was chosen for the executed parametric study. Instead of having to open each of the created DIANA projects in order to look at the output of the NLFEA, the saved CSV-files were opened directly in Excel. Thereafter, they were used to plot the graphs shown in Section 13.4. The generated CSV-files allowed for an efficient post-processing of the results from the parametric study. Furthermore, the user-friendly nature of the framework also allowed the user input to be defined efficiently. Less than five minute were needed to define the user input, before the parametric analysis could be run and carried out by itself in DIANA.

# 14 Conclusions and recommendations

## 14.1 Conclusions

In this thesis, a DIANA/Python framework for robust NLFEA of RC beams has been created. The framework provides a reliable way of approaching NLFEA of RC beams. It relies on recommended properties, which will be applied by default. The created framework also provides a more efficient way of modelling, than through the DIANA graphical interface, as the user only has to define selected input. In general, performing NLFEA requires experience to ensure quality, robustness and speed of the analysis. However, the provided default properties make it possible for users with limited expertise to perform complex analysis.

The object-oriented structure of the framework allows for flexible user input and a more user-friendly way of generating the DIANA workflow. For example does the framework allow the user to create an unlimited number of reinforcement bars and point loads, including zero. The user-friendliness of the framework has been further improved by adding templates and example blocks. The symmetry option, as well as the possibility to generate CSV-files for the load-deflection response, also contribute to a more efficient analysis and workflow. Furthermore, the framework can be used to perform parametric studies. This is a very useful feature, which can be used to perform sensitivity studies. The results from the executed parametric study, supports the fact that a nonlinear analysis will obtain an increased capacity for a concrete beam compared to 'conventional' sectional analysis. Hence, it can be concluded that the DIANA/Python framework for robust NLFEA allows for more optimal design of RC beams.

The default solution strategy achieves a high level-of-approximation and reliability. This can be concluded based on the validations studies of Case B1 and B2. Overall, the results of the NLFEAs obtained for Case B1 and B2 agree well with the experimental results. The NLFEAs obtain the expected pre-peak behaviour for the load-deflection curves. The failure mechanisms are also modelled as expected for both cases. The NLFEA of Case B1 fails shortly after the peak load, and the post-peak response differs somewhat from the experimental results. However, an even better agreement between the numerical and experimental results could probably have been obtained using the framework, if the significant input parameters were studied in greater detail and tweaked in accordance with the experimental results. The flexible structure of the framework and the option to perform parametric studies, allows for a closer inspection of the significance of the different input parameters. The peak load of Case B2 was slightly overestimated. Also here, a better agreement between the numerical and experimental results could have been obtained using the framework, if a closer inspection of significant input parameters had been performed. Especially the fracture energy, the crack bandwidth and the crack model seemed to have a significant influence on the results of the performed NLFEAs. In general, the benchmark studies highlight how the implemented default parameters provide a great starting point for robust NLFEA of RC beams.

## 14.2 Recommendations for future work

For future work, the framework can be expanded to include beams with more advanced geometry, like more complicated cross sections or web-openings. Additional types of loading and different boundary conditions could also be included. Furthermore, the framework could be extended with more optional input parameters, as for example allowing the user to optionally select the crack model to be used. Having said that, if more optional parameters are to be included, one should be aware of the fact that this might reduce the user-friendliness and robustness of the framework.

The framework could also be utilized as is. For example could it be interesting to perform parametric studies on the influence of certain input parameters on the results of NLFEA. The influence of the fracture energy on the post-peak deformation could for example be studied in more detail, as mentioned in Part IV. In general, the created framework for NLFEA could be applied to optimize the design of RC beams.

The default parameters based on recommendations ensures robust NLFEA. This framework in Python could be adapted to other finite element analysis software, as the same recommendations will also apply here. The specific DIANA commands would have to be changed, however the object-oriented structure could be kept as is.

# Bibliography


[1] Max A.N. Hendriks and Marco A. Roosen (editors). *Guidelines for Nonlinear Finite Element Analysis of Concrete Structures*. Rijkswaterstaat Centre for Infrastructure, Report RTD:1016-1:2019, 2019. URL: http://homepage.tudelft.nl/v5p05/RTD%201016-1(2020)%20version%202.2%20(final%2020200402)%20Guidelines%20for%20Nonlinear%20Finite%20Element%20Analysis%20of%20Concrete%20Structures.pdf (visited on 9th June 2022).

[2] CEN. *NS-EN 1992-1-1:2004+A1:2014+NA:2021 Eurocode 2: Design of concrete structures Part 1-1: General rules and rules for buildings*. Standard Norge, 2021. URL: https://www.standard.no/no/Nettbutikk/produktkatalogen/Produktpresentasjon/?ProductID=1365302 (visited on 9th June 2022).

[3] fédération internationale du béton/International Federation for Structural Concrete (fib). *fib Model Code for Concrete Structures 2010*. 2013. DOI: 10.1002/9783433604090.

[4] Arjen de Putter. *Towards a uniform and optimal approach for safe NLFEA of reinforced concrete beams*. Master's thesis. Delft University of Technology, 2020. URL: http://resolver.tudelft.nl/uuid:f0282508-ff25-4043-98cf-4b7bfc395a4b (visited on 9th June 2022).

[5] DIANA FEA BV. *Automated beam script*. 2021. URL: https://www.researchgate.net/lab/Ab-van-den-Bos-Diana-Fea-Lab (visited on 9th June 2022).

[6] DIANA FEA BV. *DIANA Documentation - release 10.5*. DIANA FEA BV, 2021. URL: https://manuals.dianafea.com/d105/Diana.html (visited on 9th June 2022).

[7] Robert D. Cook et al. *Concept and applications of finite element analysis*. 4th edition. John Wiley & Sons Inc., 2001.

[8] Kjell Magne Mathisen. Lecture notes - Nonlinear Finite Element Analysis. *TKT4197*. URL: https://ntnu.blackboard.com/ultra/courses/_29372_1/cl/outline (visited on 20th Nov. 2021).

[9] Jan Arve Øverli and Svein I. Sørensen. *Concrete Structures 3 - Compendium*. TKT4222. Trondheim: Department of Structural Engineering, NTNU, 2013.

[10] Schuster Engineering. *FEA 32: Nonlinear Analysis 1*. URL: https://www.youtube.com/watch?v=Jrflw1veZeo (visited on 20th Nov. 2021).

[11] Kesio Palacio. *Practical Recommendations for Nonlinear Structural Analysis in DIANA*. The Netherlands: TNO DIANA BV, 2013. URL: https://dianafea.com/sites/default/files/2018-04/Nonlinear_Structural_Analysis_in_DIANA.pdf (visited on 13th Apr. 2022).

[12] PTC. *Include Snap-Through*. URL: http://support.ptc.com/help/creo/creo_pma/italian/index.html#page/simulate/simulate/analysis/struct/reference/inc_snap_thru.html (visited on 20th Nov. 2021).

[13] A. de Putter et al. Quantification of the resistance modeling uncertainty of 19 alternative 2D nonlinear finite element approaches benchmarked against 101 experiments on reinforced concrete beams. *Structural Concrete*. Structural Concrete. 2022;1-15, 2022. DOI: 10.1002/suco.202100574.

[14] Kjell Magne Mathisen. Lecture notes - Finite Element Methods in Strength Analysis. *TKT4192*. URL: https://ntnu.blackboard.com/ultra/courses/_25468_1/cl/outline (visited on 14th Apr. 2022).

[15] DIANA FEA BV. Advanced DIANA Training course. *Concrete Crack Models Discrete vs. Smeared.* 2022. URL: https://dianafea.com/index.php/2022-Adv-ConCrack (visited on 16th Feb. 2022).

[16] Xiong Zhang, Zhen Chen and Yan Liu. 'Chapter 6 - Constitutive Models'. In: *The Material Point Method.* Oxford: Academic Press, 2017, pp. 175–219. DOI: https://doi.org/10.1016/B978-0-12-407716-4.00006-5.

[17] F. J. Vecchio and M. P. Collins. The modified compression field theory for reinforced concrete elements subjected to shear. *ACI Journal.* 83, 22, 1986. URL: http://vectoranalysisgroup.com/journal_publications/jp2.pdf.

[18] Ahmed Elkady. *21 ABAQUS Tutorial: Defining Concrete Damage Plasticity Model + Failure and Element Deletion.* URL: https://www.youtube.com/watch?v=wy84XGamn3g (visited on 15th Apr. 2022).

[19] Robert G. Selby and Frank J. Vecchio. 'A constitutive model for analysis of reinforced concrete solids'. In: *Canadian Journal of Civil Engineering* 24 (1997), pp. 460–470. URL: https://tspace.library.utoronto.ca/bitstream/1807/10030/1/Vecchio_11330_3309.pdf (visited on 9th June 2022).

[20] Martin Hallberg. *Numerisk simulering av ikke-lineær oppførsel av armert betong.* Master's thesis. NTNU, 2014.

[21] Department of Structural Engineering. Formula sheet - Structural Mechanics part 1 and 2. *TKT4116.* NTNU, 2018.

[22] Maurício Prado Martins et al. 'Modelling of tension stiffening effect in reinforced recycled concrete'. In: *Revista IBRACON de Estruturas e Materiais* 13 (2020). DOI: 10.1590/s1983-41952020000600005.

[23] DIANA FEA BV. Online course. *Nonlinear behaviour of reinforced concrete structures.* 2019. URL: https://dianafea.com/2019-10-rc-course (visited on 3rd Oct. 2019).

[24] David Amos. *Object-Oriented Programming (OOP) in Python 3.* URL: https://realpython.com/python3-object-oriented-programming/ (visited on 15th Dec. 2021).

[25] M.A.N. Hendriks, A. de Boer and B. Belletti. *Validation of the Guidelines for Nonlinear Finite Element Analysis of Concrete Structures. Part: Reinforced beams.* Rijkswaterstaat Centre for Infrastructure, Report RTD:1016-3A:2017, 2017. URL: http://homepage.tudelft.nl/v5p05/RTD%201016-3A(2017)%20version%201.0%20Validation%20of%20the%20guidelines%20for%20NLFEA%20of%20RC%20structures%20Part%20Reinforced%20beams.pdf (visited on 9th June 2022).

[26] F.J. Shim W. Vecchio. *Experimental and Analytical Reexamination of Classic Concrete Beam Tests.* J. Struct. Engrg. ASCE 130(3) 460-496, 2004.

[27] DIANA FEA BV. *DIANA Verification Report - release 10.5.* DIANA FEA BV, 2021. URL: https://manuals.dianafea.com/d105/Diana.html (visited on 9th June 2022).

[28] DIANA FEA BV. *Reinforced Concrete Beam: Simulation of an Experimental Test.* Tutorial. URL: https://dianafea.com/system/files/rcb_0.pdf (visited on 9th Dec. 2021).

[29] Svein Ivar Sørensen. *Betongkonstruksjoner - Beregning og dimensjonering etter Eurocode 2-* 2nd edition. Fagbokforlaget, 2017.

# Appendix

## A User input

```python
def functionOfUserInput(parametricValue): #Input inside function allows for parametric study
    #-----------------------------------------------------------------
    #---------------------- 2D BEAM SCRIPT ---------------------------
    #--------- DIANA/Python framework for robust NLFEA of RC beams ---
    #-----------------------------------------------------------------
    #By Katja Hansen
    #If questions, please contact: katjahansen.mail@gmail.com
    #This script is part of Katja Hansen's Master's Thesis, June 2022.

    #-----------------------------------------------------------------------------------------
    #TABLE OF CONTENTS
    #1. INFO BEFORE CHANGING THE SCRIPT
        #1.1 FIRST TIME use of beam script
        #1.2 General project information
        #1.3 Units
        #1.4 Commenting and uncommenting
        #1.5 Empty templates
    #2. PARAMETRIC STUDY
    #3. EXTRA INFO
        #3.1 About coordinates
        #3.2 Creation of objects from a class
        #3.3 Explanation of lines
    #4. CREATING PROJECT
        #4.1 Modelling choices
    #5. CREATING THE GEOMETRY
        #5.1 The beam
            #5.1.1 Geometry
        #5.2 Reinforcement
            #5.2.1 Cover
            #5.2.2 User defined variables
        #5.3 Longitudinal reinforcement
            #5.3.1 Creation of longitudinal reinforcement
        #5.4 Transverse reinforcement (shear)
            #5.4.1 Creation of transverse reinforcement
            #5.4.2 Spacing/positions of transverse reinforcement
        #5.5 Plates
            #5.5.1 Option 1: using template for 3-point bending
            #5.5.2 Option 2: using template for 4-point bending
            #5.5.3 Option 3: Creation of individual plates
    #6. LOADS
        #6.1 Option 1: using template for 3- or 4-point bending
        #6.2 Option 2: Creation of individual loads
    #7. MATERIAL MODELS
        #7.1 Concrete material model
            #7.1.1 Concrete properties
        #7.2 Steel plates material model
        #7.3 Reinforcement material model
    #8. STRUCTURAL INTERFACES
    #9. MESH
    #10.ANALYSIS
        #10.1 Linear Analysis
            #10.1.1 RUN LINEAR ANALYSIS
        #10.2 Nonlinear Analysis
            #10.2.1 Arc length control
            #10.2.2 Load incrementation
            #10.2.3 Iteration method
            #10.2.4 Line search
            #10.2.5 Convergence criteria
            #10.2.6 Additional choices for analysis
            #10.2.7 RUN NONLINEAR ANALYSIS
    #11. OUTPUTS FROM NONLINEAR ANALYSIS
        #11.1 Analysis output
        #11.2 Load-displacement graph
        #11.3 Load-displacement CSV
    #---------------------------------------------------------------------------------------

    ###1. INFO BEFORE CHANGING THE SCRIPT
```

```
#-#1.1 FIRST TIME use of Beam Script:
## You have to instal python modules to run the script in DIANA.
## This is done in the following manner:
## Open the DIANA Command Box (via the start menu on your computer)
## Choose "Run as administrator"
## Paste the following lines into the command box:
# %DIAPATH_W%\python\python -m pip install -t %DIAPATH_W%\modules numpy
# %DIAPATH_W%\python\python -m pip install -t %DIAPATH_W%\modules matplotlib

#-#1.2 General project information:
## Script A, B and C (A: UserInput, B: parametricStudy, C: classesAndFunctions, D: main),
##should all be saved in the same working folder.
## Script A: UserInput (and B: parametricStudy) can be changed by the user.
## A parametric study can be performed for one variable in the input script. See section 1.4.
## To run a script in DIANA, choose File -> Run saved scripts
## Select and run either:
## Script D: main #For creation of a beam
## Script B: parametricStrudy #For a parametric study
## Specified outputs, ex. csv-files, plots etc., will be created automatically and saved to a
##user-specified directory path.

#-#1.3 Units
##All input have to be in accordance with the chosen units for this script.
##The following units are used for this script:
## length = "MM"
## force = "N"
## mass = "T"
## temperature = "CELSIU"

#-#1.4 Commenting and uncommenting:
## Lines that can be commented/uncommented are marked with: **Optional**
## The default input will be used, unless an optional input is given.
## Lines that are not relevant for your script should stay commented, using ctrl + k
## Relevant lines can be uncommented, using crtl + shift + k

#-#1.5 Empty templates:
## Empty templates for creation of different objects are given as follows:
''' Empty template:
Text that can be copied and filled with desired input
'''
## crtl + f can be used to replace template object name, with user defined name.

###2. PARAMETRIC STUDY
## For a parametric studiy, multiple analyses will be performed. One parameter will be varied with
##different inputs, while the others stay the same.
## A parametric study is performed by editing Script B: parametricStudy and Script A: UserInput.
## In Script B, the input for the varying parameter is defined.
## In Script A, the varying parameter has to be marked with the input parametricValue. See Example:
'''Example
    beam.geometry.length = parametricValue
'''

#--------------------------------------------------------------------------------------------------
###3. EXTRA INFO (Do not have to read before using script)

#-#3.1 About coordinates:
## Beam has coordinates [x,y] = [0,0] in lower left corner.
## Coordinate system starts in lower left corner of beam, with positive X to the right
##and positive Y upwards.
## Plates: x-coordinates refers to midpoint of plate. (= loadingpoint)
## Span: length between x-coordinate/midpoints of plates/loadingpoints.
## All coordinates has 0 as its default value

#-#3.2 Creation of objects from a class:
## An object oriented structure is used for this script.
## All classes are defined in Script C: classesAndFunctions
## When an empty template is used to create objects of a class, the user is free to choose
##the name of the object.
## Unless otherwise specified, the user can create as many objects as desired. zero is also allowed.
## The objectname of the object beam from the class Beam should not be changed.

#-#3.3 Explanation of lines:
###  NEW SECTION
#-# Subsection
```

```
    #'# Subsubsection
    ## Text
    #remark/explanation
    #** optional **
    ##Option i: choose one of the available options if required
    ''' Empty template
    Text that can be copied and filled with desired input
    crtl + f can be used to replace template object name, with user defined name.
    '''
    #\\ Add your own comments as a user

    #-----------------------------------------------------------------------------------------------
    ###4. CREATING PROJECT
    ##dirPath is the directory path of the project (= Where your files should be saved)
    ##path=r.....don't forget r before the working path. For example:
    ##Project.dirPath = r"C:\Users\katja\OneDrive - NTNU\Documents\master"
    ##By default, the project is saved in the same folder as the where the Python scripts are saved.
    ##By default the project name is generated as follows:
    ##Project.name = Project.modelName + "_" + TodaysDate + "_" + Project.modelExtraInfo

    Project.modelName = "TestBeam"
    Project.modelExtraInfo = "default"

    ##The following can be changed:
    # Project.dirPath = #**Optional**
    # Project.name = #**Optional**

    Project.size_in_m = 1000

    #-#4.1 Modelling choices
    ## The beam will be modelled in 2D
    ## The whole beam is modelled by default.

    ## If the following option is chosen, only half the beam will be modelled:
    # Options.symmetric_Beam =  True #**Optional**, default is False

    ###5. CREATING THE GEOMETRY
    #-#5.1 The beam
    #'#5.1.1 Geometry:
    beam = Beam() #creating beam object
    beamGeometry = Geometry() #creating geometry object
    beam.add_geometry(beamGeometry) #adding the geometry to the beam

    beam.geometry.length = 6000 #[mm]
    beam.geometry.height = 600 #[mm]
    beam.geometry.width = 400 #[mm] #thickness

    #-#5.2 Reinforcement
    ##Embedded reinforcement bars is the only implemented type of reinforcement.

    #'#5.2.1 Cover: #**Optional input**
    ##Cover is an optional object, which can be used for your convinience:
    cover = Cover() #creating cover object
    ##Choose one of the options.
    ##The section with the chosen option should be uncommented,
    ##while the other options should be commented.

    # ##**Option 1**: definition of each cover
    cover.side = 48 #[mm]
    cover.top = 50 #[mm]
    cover.bot = 64 #[mm]

    ##**Option 2**: all sides have same cover:
    # cover.length = 100 #[mm]
    # cover.side = cover.top = cover.bot = cover.length

    #'#5.2.2 User defined variables
    ##Here new parameters can be defined by the user to easier create the reinforcement.
    ##The following parameter is suggested:
    # diameterOfBars =  #**Optional**

    #-#5.3 Longitudinal reinforcement
    ##Define as many longitudinal reinfocement bars as desired. (To create 0 is also allowed))
    ##The user is free to choose the name of each object.
    ##Longitudinal reinforcement is created as follows:
```

```
##nameOfBar = Longitudinal_Reinfo(Name, As, F_y, F_u, Epsilon_u, (E_mod))
##As = cross section area[mm^2], F_y = yield strength [N/mm^2], F_u = ultimate strength [N/mm^2],
##Epsilon_u = ultimate strain []
##Default value of 200000 for Youngs modulus (E_mod) if no value is given. [N/mm^2]
##x- and y- coordinates does also have to be defined, for each reinfo bar.

## About coordinates:
##Beam has coordinates [x,y] = [0,0] in lower left corner.
##Coordinate system starts in lower left corner of beam,
##with positive X to the right and positive Y upwards.

##Examples of the creation of longitudinal reinforcement bars:
'''Example block (can be copied and changed to create your own objects)
#
upper_reinfo = Longitudinal_Reinfo("upper", 300, 315, 460, 2.3425E-2)
upper_reinfo.y_coord = beam.geometry.height - cover.top
upper_reinfo.x_coord_start = cover.side
# upper_reinfo.x_coord_end = beam.geometry.length - cover.side #**Input if beam is not symmetric**

#
ReinfoLow = Longitudinal_Reinfo("lower", 1400, 436, 700, 4.78E-2, 220000)
ReinfoLow.y_coord = 20
ReinfoLow.x_coord_start = 0
# ReinfoLow.x_coord_end = beam.geometry.length #**Input needed if beam is not symmetric**
'''

##Copy this block to create long.reinfo objects:
''' Empty template
templateName = Longitudinal_Reinfo( )
templateName.y_coord =
templateName.x_coord_start =
# templateName.x_coord_end = #**Input needed if beam is not symmetric**
'''

#'#5.3.1 Creation of longitudinal reinforcement
##CREATE YOUR LONG.REINFO OBJECTS HERE

#-#5.4 Transverse reinforcement (shear)
##All shear bars will have the same material properties.
##, which means only one object should be created for shear reinforcement.
##nameOfBar = Transverse_Reinfo(Name, As, F_ym, F_um, Epsilon_u, (E_mod)).
##As = cross section area[mm^2], F_y = yield strength [N/mm^2], F_u = ultimate strength [N/mm^2],
##Epsilon_u = ultimate strain []
##Default value of 200000 for Youngs modulus (E_mod) if no value is given. [N/mm^2]
##y-coordinates does also have to be defined.

##Examples of the creation of transverse reinfo:
'''Example block (can be copied and changed to create your own objects)
shear_reinfo = Transverse_Reinfo("Shear_reinfo", 25.7, 600, 651, 4.70E-2,220000)
shear_reinfo.y_coord_bot = cover.bot
shear_reinfo.y_coord_top = beam.geometry.height - cover.top
'''

##Copy this block to create trans.reinfo objects:
''' Empty template
##You should maximum create one object!
templateName = Transverse_Reinfo( )
templateName.y_coord_bot =
templateName.y_coord_top =
'''

#'#5.4.1 Creation of transverse reinforcement
##CREATE YOUR TRANS. REINFO OBJECT HERE

#'#5.4.2 Spacing/positions of transverse reinforcement:
##The transverse reinforcement is divided into sections with the same spacing.
##You can create one or multiple sections.
##If the transverse reinforcement object is created, at least one section have to be defined.
##If the beam is symmetric, you only have to create sections for half the beam.
##A section is created as follows:
## section_i = Section(spacing, x_coord_start, x_coord_end) #All three inputs are int

## About coordinates:
##Beam has coordinates [x,y] = [0,0] in lower left corner.
##Coordinate system starts in lower left corner of beam,
```

```
    ##with positive X to the right and positive Y upwards.

    ##Examples of the creation of sections:
    '''Example block (can be copied and changed to create your own objects)
    #Ex. 1:
    section = Section(10, cover.side, beam.geometry.length - cover.side)

    #Ex. 2:
    spacing_large = 168
    spacing_small = 86
    section_1 = Section(spacing_small, cover.side, cover.side + 4*spacing_small)

    section_2 = Section(spacing_large, cover.side + 4*spacing_small,
    (beam.geometry.length/2) - 2*spacing_small)

    section_3 = Section(spacing_small, (beam.geometry.length/2) - 2*spacing_small, beam.geometry.length)
    '''

    ##Copy this block to create section objects:
    ''' Empty template
    templateName = Section( )
    '''

    ##CREATE YOUR SECTION OBJECTS HERE

    #-#5.5 Plates
    ##Available plates are loading plates (L) and support plates (S)
    ##All plates will have the same geometry.
    ##The width/thickness of the plates will be equal to the beam thickness
    plateGeometry = Geometry() #creating geometry object for plates

    plateGeometry.length = 150  #[mm]
    plateGeometry.height = 35 #[mm]

    ##Choose one of the following options to create the plates.
    ##The section with the chosen option should be uncommented,
    ##while the other options should be commented.
    ##Option 1 and 2 are templates for bending tests. Setup is shown in master thesis.

    #'#5.5.1 Option 1: using template for 3-point bending
    Options.template_3PointBending = True
    beam.geometry.lengthOfSpan = beam.geometry.length - 400 #[mm]

    #'#5.5.2 Option 2: using template for 4-point bending
    # Options.template_4PointBending = True
    # beam.geometry.lengthOfSpan = 5000 #[mm]
    # beam.geometry.lengthOfInnerSpan = 3000 #[mm]

    #'#5.5.3 Option 3: Creation of individual plates
    ##Plates are creating by specifying the x-coordinate of the midpoint (loading point)
    ##The user is free to choose the name of each object.
    ##Each plate is created as follows:
    ##nameOfPlate = Plates(typeOfPlate, name, x_coord)
    ##typeOfPlate = "S" (supportplate)  or "L" (loadingplate)
    ##name = string, x_coord = int
    ##By default translations are fixed in both x- and y-direction for support plates. Can be changed:
    ##To change BC's for support plates, use:
    ##plateObject.fixedTranslation_y = False or plateObject.fixedTranslation_x = False

    ## About coordinates:
    ##Beam has coordinates [x,y] = [0,0] in lower left corner.
    ##Coordinate system starts in lower left corner of beam,
    ##with positive X to the right and positive Y upwards.

    ##Examples of the creation of plates:
    '''Example block (can be copied and changed to create your own objects)
    sup_plate1 = Plates("S", "Support plate 1", 50)
    sup_plate1.fixedTranslation_y = False #**Optional**
    # sup_plate1.fixedTranslation_x = False #**Optional**
    load_plate1 = Plates("L", "Loading plate 1", 70)
    load_plate2 = Plates("L", "Loading plate 2", 100)
    '''

    ##Copy this block to create plate objects:
    '''Empty template
```

```
    templateName = Plates(  )
    # nameOfPlate.fixedTranslation_y = False #**Optional**
    # nameOfPlate.fixedTranslation_x = False #**Optional**
    '''

    ##CREATE YOUR PLATE OBJECTS HERE (option 3)

    ###6. LOADS
    ##The dead weight is applied in the first load step.
    ##Then point loads can be applied. (Only point loads are implemented)
    ##Loads are defined as follows:
    ##load = Loads(typeOfLoad, name, value, direction)
    ##typeOfLoad = "Point" (Only point loads are implemented)
    ##name = str
    ##value: int or double. OBS! positive y-axis is upwards, NEGATIVE LOADS ACTS DOWNWARDS
    ##direction = "X", "Y".

    #-#6.1 Option 1: using template for 3- or 4-point bending
    ##Both loads are assumed to have same value for 4-point bending. Only one has to be defined.
    load = Loads("Point", "Load", -1000, "Y")

    #-#6.2 Option 2: Creation of individual loads
    ##Each load should have a plate object as its target.
    ##A target is defined as follows:
    #load.target = plateobject

    ##Examples of the creation of point loads:
    '''Example block (can be copied and changed to create your own objects)
    load1 = Loads("Point", "Load 1", -1000, "Y")
    load1.target = plate1
    load2 = Loads("Point", "Load 2", -200, "Y")
    load2.target = plate2
    '''

    ##Copy this block to create point load objects:
    '''Empty template
    templateName_Load = Loads()
    templateName_Load.target = templateName_Plate
    '''

    ##CREATE YOUR LOAD OBJECTS HERE (option 2)

    ###7. MATERIAL MODELS
    #-#7.1 Concrete material model
    #The concrete material is defined as follows: concrete = Concrete(f_ck)
    concrete = Concrete(30)

    #'#7.1.1 Concrete properties:
    ##The recommended material model used recommended material properties by default.
    ##These are based on EC2, fib2010 and RTD*.
    ##See Script C, Concrete class for default parameters
    ##The properties can be changed by uncommenting the desired lines.
    ##The properties that remains commented, will use recommended values.
    ##If recommended curves are changed, the script is not guaranteed to run
    ##since additional parameters might be needed.

    ## The following properties can be changed:
    # concrete.Youngs_modulus = #**Optional**
    # concrete.poissons_ratio = #**Optional**
    # concrete.mass_density = #**Optional**
    # concrete.tensile_strength = #**Optional**
    # concrete.compressive_strength =  #**Optional**
    # concrete.tensile_FractureEnergy = #**Optional**
    # concrete.compressive_FractureEnergy = #**Optional**
    # concrete.aggregate_type = #**Optional**
    # concrete.tensile_curve = "HORDYK" #**Optional**
    # concrete.crackBandwidt_specification = #**Optional**
    # concrete.poissonsRatio_reductionModel = #**Optional**
    # concrete.compression_curve = #**Optional**
    # concret.lateralCracking_reductionModel = #**Optional**
    # concrete.lateralCracking_reductionCurve_lowerBound =  #**Optional**
    # concrete.confinementModel = "NONE" #**Optional**

    #-#7.2 Steel plates material model
    #A linear-elastic behaviour is assumed for the plates
```

```
    Steel_Plates.e_mod = 2000000
    Steel_Plates.poissons_ratio = 0.3

    #-#7.3 Reinforcement material model
    ##Only default material for reinforcement has been implemented.
    ##The chosen material model is Von Mises Plasticity, with a bilinear strain-stress diagram.
    ##DIANA default settings for hardening behaviour are used.

    ###8. STRUCTURAL INTERFACES
    ##A no-tension/no-friction 2D line interface is defined between
    ##the steel plates and the concrete beam.
    ##A high value for the normal compressive stiffness and a low value for the shear stiffness
    ##and normal tensile stiffness will be used as default.
    ##A diagram based on the normal compressive and tensile stiffness is used
    ##to define the nonlinear relation.

    ## The following properties can be changed:
    # Interface.Knn_tension     =   # [N/mm3] **Optional**
    # Interface.Knn_compression =   # [N/mm3] **Optional**
    # Interface.Kt              =   # [N/mm3] **Optional**

    ###9. MESH
    ##Recommended mesh is used as default.
    ##Maximum elementsize = min(beam.Length/50, beam.Height/6)
    Mesh.elementsize = 25 #[mm]

    ## The following properties can be changed:
    # Mesh.meshorder =  #**Optional**
    # Mesh.meshertype =  #**Optional**

    ###10. ANALYSIS
    ##It is recommended to run a linear analysis before running the nonlinear analysis.
    ##runsolver = True runs the analysis. Default value is false.
    ##Linear and nonlinear analysis have been implemented.
    ##An analysis is defined as follows:
    ##Analysis(_name, _typeOfAnalysis)
    ##_name = str, _typeOfAnalysis = "Linear" or "Nonlinear"

    #-#10.1 Linear Analysis
    ##DIANA Primary output is chosen for the linear analysis.
    LFEA = Analysis("Linear analysis", "Linear") #creating linear static analysis

    ## The following properties can be changed:
    # LFEA.method = **Optional input**
    # LFEA.output = **Optional input**

    #'#10.1.1 RUN LINEAR ANALYSIS
    ##To run the linear analysis, uncomment the following line:
    # LFEA.runSolver = True #**Optional**

    #-#10.2 Nonlinear Analysis
    NLFEA = Analysis("Nonlinear analysis", "Nonlinear") #creating nonlinear analysis

    #'#10.2.1 Arc length control:
    ##Arc length control is strongly recommended.
    ##The arc length control will be applied over the whole beam.
    NLFEA.arcLengthControl = True #**Default and strongly recommended as true**

    #'#10.2.2 Load incrementation:
    ##Load control is applied (in combination with arc length control).
    ##As recommended, the load incrementation uses an automatic procedure.
    ##Energy based adaptive loading is the only implemented method. This method must
    ##be combined with arc length control.
    incrementation = Incrementation() #Creating an incrementation object
    NLFEA.setIncrementalMethod(incrementation) #adding incremental method to the analysis

    ##Option 1: Energy based adaptive loading
    ##The energy based method MUST be combined with arc length control.
    incrementation.method = "ENERGY"
    incrementation.initial_step_size = 5    #initial size for the first step
    incrementation.max_step_size = 10       #upper limit of the step size
    incrementation.min_step_size = 3        #lower limit of the step size
    incrementation.nrOfSteps = 150          #maximum number of steps
```

```
    #'#10.2.3 Iteration method:
    ##Only Newton-Raphson methods have been implemented.
    ##Choose one of the options. The section with the chosen option should be uncommented,
    ##while the other options should be commented.
    iteration = Iteration() #creating iteration object
    NLFEA.setIterationMethod(iteration) #adding iterative procedure to the analysis

    ## Option 1: Regular Newton-Raphson
    iteration.method = "NEWTON-RAPHSON"
    iteration.typeOfMethod = "REGULA"
    iteration.nrOfIterations = 100

    ## Option 2: Modified Newton-Raphson
    # iteration.method = "NEWTON-RAPHSON"
    # iteration.typeOfMethod = "MODIFI"
    # iteration.nrOfIterations = 100

    ## The following properties can be changed:
    # iteration.firstStiffnessMatrix = **Optional input**

    #'#10.2.4 Line search:
    ##Using a line search algorithm is recommended.
    NLFEA.lineSearch = True #**Default and recommended as true**

    #'#10.2.5 Convergence criteria:
    ##One or multiple convergence criteria MUST be chosen.
    ##Recommended to use a force norm and an energy norm.
    ##Choose one of the options. Optional to choose one or more convergence criteria.
    convergence = Convergence()          #creating convergence object
    NLFEA.setConvergence(convergence)   #adding convergence criteria to the analysis

    convergence.useForceNorm = True      #**Default and recommended as true**
    convergence.useEnergyNorm = True     #**Default and recommended as true**
    convergence.useDispNorm = False      #**Default and recommended as false**

    ## Option 1: Suggested tolerances
    ##Force norm:
    convergence.forceNorm = 0.01
    convergence.forceNorm_deadLoad = 0.05
    ##Energy norm:
    convergence.energyNorm = 0.001
    convergence.energyNorm_deadLoad = 0.01

    ## Option 2: Choose your own tolerances:
    ##For this option, one or multiple convergence criteria MUST be chosen.

    ##Force norm:
    # convergence.forceNorm =
    # convergence.forceNorm_deadLoad =

    ##Energy norm:
    # convergence.energyNorm =
    # convergence.energyNorm_deadLoad  =

    ##Displacement norm:
    # convergence.dispNorm =
    # convergence.energyNorm_deadLoad  =

    #'#10.2.6 Additional choices for analysis
    NLFEA.allConvergenceNormsHaveToBeSatisfied = False  #**Default and recommended as false**
    NLFEA.continueIfNoConvergence = True                #**Default and recommended as true**

    #'#10.2.7 RUN NONLINEAR ANALYSIS
    ##To run the analysis, uncomment the following line:
    # NLFEA.runSolver = True #**optional**

    ###11. OUTPUTS FROM NONLINEAR ANALYSIS
    ##Outputs will be saved to the specified directory at beginning.
    ## All outputs are optional. Comment or write False to not include output.

    #-#11.1 Analysis output:
    NLFEA.output = {
        "DISPLA TOTAL TRANSL GLOBAL" : True,    #**Optional**
        "FORCE REACTI TRANSL GLOBAL" : True,    #**Optional**
        "STRAIN TOTAL GREEN GLOBAL" : True,     #**Optional**
```

```
        "STRAIN TOTAL GREEN PRINCI" : True,     #**Optional**
        "STRAIN CRKWDT GREEN GLOBAL" : True,    #**Optional**
        "STRAIN CRACK GREEN" : True,            #**Optional**
        "STRAIN CRKWDT GREEN PRINCI" : True,    #**Optional**
        "STRESS TOTAL CAUCHY GLOBAL" : True,    #**Optional**
        "STRESS TOTAL CAUCHY PRINCI" : True     #**Optional**
    }

    #-#11.2 Load-displacement graph (displacement in y-dir) **Optional**
    #Edit and uncomment this section to generate a load-displacement graph.
    ##Downward displacement is defined as positive by default.

    # #Specify the coordinates of the point where the displacement will be retrived.
    # LoadDispY_Graph.x_coord = beam.geometry.length/2 #[mm]
    # LoadDispY_Graph.y_coord = 0 #[mm]

    # ##The following options can be changed:
    # ##x_label, y_label = str
    # ##xLim, y_Lim = [lowerBound, upperBound]; lowerBound, upperBound = int
    # # Scalefactor.loadFactor_plot = #**optional**
    # # Scalefactor.displacement_plot = #**optional**
    # # LoadDispY_Graph.xlabel = #**optional**
    # # LoadDispY_Graph.ylabel = #**optional**
    # # LoadDispY_Graph.xLim = #**optional**
    # # LoadDispY_Graph.yLim =  #**optional**

    #-#11.3 Load-displacement CSV (displacement in y-dir)**Optional**
    ##Edit and uncomment this section to generate a
    ##CSV-file for load-displacement at specified coordinate.
    ##Downward displacement is defined as positive by default.
    ##The CSV-file will have the following format:
    ##  (Scaled) Loadfactor;(Scaled) Displacement
    ##  1.00;0.50
    ##  2.30;0.65

    # ##Specify the coordinates of the point for which the CSV-file will be generated:
    # LoadDispY_CSV.x_coord = beam.geometry.length/2 #[mm]
    # LoadDispY_CSV.y_coord = 0 #[mm]

    # #The following options can be changed:
    # # Scalefactor.loadFactor_CSV =  #**optional**
    # # Scalefactor.displacement_CSV = #**optional**
    # # LoadDispY_CSV.decimalPlaces = #**optional**


    #-----------------------------------------------------------------------------------
    ## Additional info:
    #*RTD = Rijkwaterstaat Technincal Document: Guidelines for NLFEA of Concrete Structures
    globals().update(locals()) #Allows main script to access all variables in function as globals
    return
```

# B Parameric Study

```python
def parametricStudy(valuesList):
    for val in valuesList:
        wd = sys.path[0] #Working directory of main.py
        exec(open(wd+"\\C_classesAndFunctions.py").read(), globals())
        LoadDispY_Graph.parametric = val
        LoadDispY_CSV.parametric = val
        exec(open(wd+"\\A_UserInput.py").read(), globals())
        functionOfUserInput(val)
        parametric_Study = True
        if not hasattr(Project, 'name'):
            #Default projectname
            Project.name = Project.modelName + "_" + date + "_" + Project.modelExtraInfo + "_" + str(val)
        else:
            Project.name = Project.name + "_" + str(val)
        exec(open(wd+"\\D_main.py").read())
    return locals()

#This is the script to be run using DIANA to perform a parametric study
##INPUT LIST OF PARAMETRIC VALUES HERE:
parametricStudy([3000,4000])
```

# C Classes and Functions

```
#-------------------------------------------------------------------
#--------- Definitions of classes and functions - 2D BEAM SCRIPT ----
#-------------------------------------------------------------------
##SHOULD NOT BE CHANGED BY USER
#If this script (C) is changed, both script D (main) and A (userInput) will be affected.
numPy = 1

try:
    import numpy as np
except:
    numPy = 0
    print("Numpy not detected, fillets and other features are not supported!!")

if numPy == 1:
    import numpy as np

# Importing libraries
import math
from math import *
import matplotlib
matplotlib.use('AGG')
import matplotlib.pyplot as plt
import time
import os, inspect
import csv

### Preparations, do not change this as an user
date = (time.strftime("%Y%m%d"))

### CLASS DEFINITIONS:
#Setting up classes to structure input, do not change as user
class Project():
    pass

class Units():
    length = "MM"
    force = "N"
    mass = "T"
    temperature = "CELSIU"
    pass

class Options():
    symmetric_Beam = False #**Optional input, default value is false**
    dimension = "2D" #Only 2d is implemented
    template_3PointBending = False #**Optional input, default value is false**
    template_4PointBending = False #**Optional input, default value is false**
    includeSelfWeight = True #only true is implemented
    parametricStudy = False
    pass

class Beam():
    name = "Concrete Beam"
    allObjects = []
    def __init__(self):
       self.allObjects.append(self)
    def add_geometry(self, _geometry):
        self.geometry = _geometry

class Plates():
    allObjects = []
    y_coord = 0 #default value
    fixedTranslation_x = True
    fixedTranslation_y = True
    def __init__(self, _typeOfPlate, _name, _x_coord):
        self.allObjects.append(self)
        self.typeOfPlate = _typeOfPlate
        self.name = _name
        self.x_coord = _x_coord
    def add_geometry(self, _geometry):
        self.geometry = _geometry

class Geometry():
```

```
    pass

class Cover():
    side = 0
    top = 0
    bot = 0

class Longitudinal_Reinfo():
    allObjects = []
    material = "REINFO"
    typeOfReinfo = "Long"
    y_coord = 0 #default value
    x_coord_start = 0 #default value
    x_coord_end = 0 #default value
    def __init__(self, _name, _As, _f_ym, _f_um, _e_u, _E_mod = 200000):
        self.allObjects.append(self)
        self.name = _name
        self.As = _As #[mm^2]
        self.f_ym = _f_ym
        self.f_um = _f_um
        self.e_u = _e_u
        self.e_mod = _E_mod
        self.setname = _name + "_Set"

class Transverse_Reinfo():
    allNames = []
    allObjects = []
    material = "REINFO"
    typeOfReinfo = "Trans"
    y_coord_bot = 0 #default value
    y_coord_top = 0 #default value

    def __init__(self, _name, _As, _f_ym, _f_um, _e_u, _E_mod = 200000):
        self.allObjects.append(self)
        self.name = _name
        self.As = _As #[mm^2]
        self.f_ym = _f_ym
        self.f_um = _f_um
        self.e_u = _e_u
        self.e_mod = _E_mod
        self.setname = _name + "_Set"

class Section():
    allObjects = []
    def __init__(self, _spacing, _x_coord_start, _x_coord_end):
        self.spacing = _spacing
        self.x_coord_start = _x_coord_start
        self.x_coord_end = _x_coord_end
        self.allObjects.append(self)

def createShearReinfo(_reinfo):
    bool_shearBarInMiddle = False
    allShearBars = []
    addSet( "GEOMETRYREINFOSET", _reinfo.setname )
    setCurrentShapeSet( _reinfo.setname )
    y_coord_bot = _reinfo.y_coord_bot
    y_coord_top = _reinfo.y_coord_top
    nameOfBar = "ShearBar 1"
    old_x_coord_end = None
    for obj in Section.allObjects:
        x_coord_start = obj.x_coord_start
        spacing = obj.spacing

        x_coord_end = obj.x_coord_end

        if Options.symmetric_Beam :
            #if length of sections is longer than half beam, ignore rest
            x_coord_mid = beam.geometry.length/2
            if x_coord_end > x_coord_mid:
                x_coord_end = x_coord_mid

        nrOfBars = math.floor((x_coord_end - x_coord_start)/spacing) #rounding down
        if x_coord_start != old_x_coord_end:
            if nameOfBar != "ShearBar 1":
                nameOfBar = "ShearBar " + str(len(allShearBars) + 1)
```

```
            createLine( nameOfBar, [ x_coord_start, y_coord_bot ], [ x_coord_start, y_coord_top ] )
            allShearBars.append(nameOfBar)
            nameOfBars = arrayCopy( [ nameOfBar ], [ spacing, 0 ], [ 0, 0 ], [ 0, 0, 0 ], nrOfBars )
            allShearBars.extend(nameOfBars)
            nameOfBar = nameOfBars[-1]
        else:
            nameOfBars = arrayCopy( [ nameOfBar ], [ spacing, 0 ], [ 0, 0 ], [ 0, 0, 0 ], nrOfBars )
            allShearBars.extend(nameOfBars)
            nameOfBar = nameOfBars[-1]
        if Options.symmetric_Beam :
            x_coord = edgeCoordinates(nameOfBar)[0][0]
            if x_coord == beam.geometry.length/2:
                bool_shearBarInMiddle = True
                addSet( "GEOMETRYREINFOSET", "ShearBar_mid" )
                moveToShapeSet( [ nameOfBars[-1] ], "ShearBar_mid" )
        old_x_coord_end = x_coord_end
        Transverse_Reinfo.allNames.extend(allShearBars)
    return bool_shearBarInMiddle

class Concrete():
    #Material Properties:
    Name = "Concrete" #Name of the material
    material = "CONCR"
    aggregate_type = "QUARTZ" #For normal weight concrete quartzite aggregates are assumed.
    #Cement type is set to N
    material_model = "TSCR" #Total strain based cracking.
    crack_orientation = "ROTATE" #Rotating as default, will be changed to FIXED for beams without stirrups.
    tensile_curve = "EXPONE"
    crackBandwidt_specification = "GOVIND"
    poissonsRatio_reductionModel = "DAMAGE" #Poisson's ratio reduction, damage based
    compression_curve = "PARABO"
    lateralCracking_reductionModel = "VC1993" #Reduction curve due to lateral cracking
    lateralCracking_reductionCurve_lowerBound = 0.4 #lower bound of reduction curve
    confinementModel = "VECCHI" #Stress confinement model, "NONE" is conservative


    def __init__(self, _f_ck):
        self.f_ck = _f_ck
        T31_f_ck = np.array([12,16,20,25,30,35,40,45,50,55,60,70,80,90])
        T31_f_ck_cube = np.array([15,20,25,30,37,45,50,55,60,67,75,85,95,105])
        self.f_ck_cube = np.interp(_f_ck,T31_f_ck,T31_f_ck_cube)
        self.EC2 = "C" + str(int(self.f_ck)) + "/" + str(int(self.f_ck_cube))    # Table entries
        self.fib = "C" + str(int(self.f_ck))

class Steel_Plates(): #Material model for steel plates
    #A linear-elastic behaviour is assumed
    material = "MCSTEL"
    material_model = "ISOTRO"
    density = 0 #Do not wish to take the weight of the plates into concideration
    pass


def concreteFromEC2():
    addMaterial( "Concrete_EC2", "CONCDC", "EN1992", [ "TOTCRK" ] )
    setParameter( "MATERIAL", "Concrete_EC2", "EC2CON/NORMAL/CLASS", concrete.EC2)
    setParameter( "MATERIAL", "Concrete_EC2", "EC2CON/NORMAL/AGGTYP", concrete.aggregate_type)
    return "Concrete_EC2"

def concreteFromFib2010():
    addMaterial( "Concrete_fib2010", "CONCDC", "MC2010", [ "TOTCRK" ] )
    setParameter( "MATERIAL", "Concrete_fib2010", "MC10CO/NORMAL/GRADE", concrete.fib)
    setParameter( "MATERIAL", "Concrete_fib2010", "MC10CO/NORMAL/AGGTYP", concrete.aggregate_type )
    return "Concrete_fib2010"

class Selfweight():
    name = "gravity"
    setname = "Gravity"

class Loads():
    allObjects = []
    setname = "Loads"
    def __init__(self, _typeOfLoad, _name, _value, _direction = "Normal"):
        self.allObjects.append(self)
        self.typeOfLoad = _typeOfLoad
        self.value = _value
```

```
        self.direction = _direction
        self.name = _name
    def attachLoadToTargets(self, _target, _beam):
        if self.typeOfLoad == "Point":
            #Target is a loadplate:
            #addSet( "GEOMETRYLOADSET", self.setname )
            createPointLoad( self.name, self.setname )
            setParameter( "GEOMETRYLOAD", self.name, "FORCE/VALUE", self.value )
            if self.direction == "X":
                setParameter( "GEOMETRYLOAD", self.name, "FORCE/DIRECT", 1 )
            elif self.direction == "Y":
                setParameter( "GEOMETRYLOAD", self.name, "FORCE/DIRECT", 2 )
            attach( "GEOMETRYLOAD", self.name, _target.name,
            [[   _target.x_coord, _beam.geometry.height+_target.geometry.height ]] )
        elif self.typeOfLoad == "Distributed":
            #Target is an edge of a shape, ex. part top of beam:
            #Direction = normal to surface
            createLineLoad( self.name, self.setname )
            setParameter( "GEOMETRYLOAD", self.name, "FORCE/VALUE", self.value )
            attach( "GEOMETRYLOAD", self.name, _target.name, [ _target.x_coord, _target.geometry.height] )

class Interface():
    #Default values retrieved from de Putter's beamscript.
    name = "Interface"
    Knn_tension     = 1e-9    # N/mm3
    Knn_compression = 1e+3    # N/mm3
    Kt              = 1.0     # N/mm3
    pass

class Mesh():
    meshorder = "QUADRATIC"
    meshertype = "HEXQUAD"
    pass


class Analysis():
    allObjects = []
    def __init__(self, _name, _typeOfAnalysis):
        self.name = _name
        self.typeOfAnalysis = _typeOfAnalysis
        if self.typeOfAnalysis == "Linear":
            self.command = "Structural linear static"
            self.method = "ITERAT"
            self.output = "PRIMAR"
            self.runSolver = False
        elif self.typeOfAnalysis == "Nonlinear":
            self.command = "Structural nonlinear"
            self.allConvergenceNormsHaveToBeSatisfied = False
            self.continueIfNoConvergence = True
            self.runSolver = False
            self.lineSearch = True
            self.arcLengthControl = True
        self.allObjects.append(self)
    def setConvergence(self,_Convergence):
        self.convergence = _Convergence
    def setIterationMethod(self, _Iteration):
        self.iteration = _Iteration
    def setIncrementalMethod(self, _Incrementation):
        self.incrementation = _Incrementation


class Incrementation():
    pass

class Iteration():
    firstStiffnessMatrix = "TANGEN"
    pass

class Convergence():
    useForceNorm = True
    useEnergyNorm = True
    useDispNorm = False
    pass

class Scalefactor():
```

```python
    loadFactor_plot = 1
    displacement_plot = 1
    loadFactor_CSV = 1
    displacement_CSV = 1
    pass

class LoadDispY_Graph():
    xlabel = "Displacement [mm]"
    ylabel = "Loadfactor"

class LoadDispY_CSV():
    decimalPlaces = 2

def getLoadDispYArrays(x_coord,y_coord, _analysis, scaleFactor_loadFactor, scaleFactor_displacement):
    nodeVector = findNodesCloseTo((x_coord,y_coord), Mesh.elementsize/4)
    nodeToBeUsed = nodeVector[0]
    results_table = resultsTable({"analysis": _analysis.name, "result": "Displacements",
    "components": ["TDtY"], "nodes": [nodeToBeUsed], "cases": (resultCases(_analysis.name))})

    loadFactor_vec = []
    displacement_vec = []
    for k in range(1,len(results_table)):
        caseName = results_table[k][0]
        dum = caseName.split(", ")
        dum = dum[1]
        dum = dum.split()
        loadfac = float(dum[1])
        loadFactor_vec.append(loadfac)
        displacement_vec.append(results_table[k][2])
    # Transforming into arrays
    loadFactor_arr = np.array(loadFactor_vec)
    displacement_arr = np.array(displacement_vec)
    #Scaling arrays
    loadFactor_arr  = loadFactor_arr*scaleFactor_loadFactor
    displacement_arr = displacement_arr*scaleFactor_displacement
    return loadFactor_arr, displacement_arr

def loadDispYPlot(x_coord,y_coord, _analysis, scaleFactor_loadFactor, scaleFactor_displacement):
    loadFactor_arr, displacement_arr = getLoadDispYArrays(x_coord,y_coord, _analysis,
    scaleFactor_loadFactor, scaleFactor_displacement)
    # Generating plot
    fig = plt.figure()
    ax1 = fig.add_subplot(1, 1, 1)
    ax1.set_xlabel(LoadDispY_Graph.xlabel)
    ax1.set_ylabel(LoadDispY_Graph.ylabel)
    if hasattr(LoadDispY_Graph, 'xLim'):
        ax1.set_xlim(LoadDispY_Graph.xLim)
    if hasattr(LoadDispY_Graph, 'yLim'):
        ax1.set_ylim(LoadDispY_Graph.yLim)
    ax1.plot(-displacement_arr, loadFactor_arr, label = "Load_deflection_graph")
    # Saving figure
    #  Creating target directory
    dirName = Project.dirPath + "/" + Project.modelName + "/Plots"
    if not os.path.exists(dirName):
        os.makedirs(dirName)
    if Options.parametricStudy == True:
        file_name = dirName + "/LoadDispY" + "_" + str(LoadDispY_Graph.parametric)
    else:
        file_name = dirName + "/LoadDispY"
    plt.savefig(file_name + ".png", dpi=400)
    plt.close(fig)
    return


def getLoadDisplacementCSV(x_coord,y_coord,_analysis,scaleFactor_loadFactor, scaleFactor_displacement,
    decimalPlaces):
        # Exporting results to CSV
        loadFactor_arr, displacement_arr = getLoadDispYArrays(x_coord,y_coord, _analysis,
        scaleFactor_loadFactor,scaleFactor_displacement)
        displacement_arr = displacement_arr*-1 #Defining downwards displacement as positive
        if Options.parametricStudy == True:
            file_name = r"loadDispY" + "_" + str(LoadDispY_CSV.parametric) + ".csv"
        else:
            file_name = r'loadDispY.csv'
        with open(file_name, 'w', newline='') as f:
```

```
            # f.write(Project.name; Project.modelName;  Project.modelExtraInfo; Nodenr: str(_node)'\n')
            writer = csv.writer(f, delimiter = ";")
            if scaleFactor_loadFactor != 1 and scaleFactor_displacement != 1:
                f.write("Scaled Displacement [mm];Scaled Loadfactor\n")
            elif scaleFactor_loadFactor != 1:
                f.write("Displacement [mm];Scaled Loadfactor\n")
            elif scaleFactor_displacement != 1:
                f.write("Scaled Displacement [mm];Loadfactor\n")
            else:
                f.write("Displacement [mm];LoadFactor\n")
            decimals = "%." + str(decimalPlaces)+"f"
            np.savetxt(f, np.column_stack((displacement_arr,loadFactor_arr)), decimals, delimiter = ";")


def StressStrainXArrays(x_coord,y_coord, _analysis):
    nodeVector = findNodesCloseTo((x_coord,y_coord), Mesh.elementsize/4)
    nodeToBeUsed = nodeVector[0]
    strain_table = resultsTable({"analysis": _analysis.name, "result": "Total Strains",
    "components": ["EXX"], "nodes": [nodeToBeUsed], "cases": (resultCases(_analysis.name))})
    stress_table = resultsTable({"analysis": _analysis.name, "result": "Cauchy Total Stresses",
    "components": ["SXX"], "nodes": [nodeToBeUsed], "cases": (resultCases(_analysis.name))})
    strain_vec = []
    stress_vec = []
    for k in range(1,len(strain_table)):
        stress_vec.append(stress_table[k][-1])
        strain_vec.append(strain_table[k][-1])
    # Transforming into arrays
    stress_arr = np.array(stress_vec)
    strain_arr = np.array(strain_vec)
    return stress_arr, strain_arr

def getStressStrainXCSV(_node,_element, CSVname, _analysis, decimalPlaces):
    nodeToBeUsed = _node
    elementToBeUsed = _element
    strain_table = resultsTable({"analysis": _analysis.name, "result": "Total Strains",
    "components": ["EXX"], "nodes": [nodeToBeUsed],"elements": [elementToBeUsed],
    "cases": (resultCases(_analysis.name))})
    stress_table = resultsTable({"analysis": _analysis.name, "result": "Cauchy Total Stresses",
    "components": ["SXX"], "nodes": [nodeToBeUsed],"elements": [elementToBeUsed],
    "cases": (resultCases(_analysis.name))})
    strain_vec = []
    stress_vec = []
    for k in range(1,len(strain_table)):
        stress_vec.append(stress_table[k][-1])
        strain_vec.append(strain_table[k][-1])
    # Transforming into arrays
    stress_arr = np.array(stress_vec)
    strain_arr = np.array(strain_vec)
    # Exporting results to CSV
    file_name = CSVname
    with open(file_name, 'w', newline='') as f:
        print("open")
        writer = csv.writer(f, delimiter = ";")
        f.write("Strain [MPa];Stress []\n")
        decimals = "%." + str(decimalPlaces)+"f"
        np.savetxt(f, np.column_stack(( strain_arr,stress_arr)), decimals, delimiter = ";")

##Other useful BUILT-IN DIANA functions that can be used:
#nodeVector = findNodesCloseTo((x_coord,y_coord),radius)
```

# D Main

```
#-------------------------------------------------------------------
#--------- Main script - 2D BEAM SCRIPT ----
#--------- DIANA/Python framework for robust NLFEA of RC beams ---
#-------------------------------------------------------------------
#This is the script to be run using DIANA
#DO NOT CHANGE THIS SCRIPT AS AN USER
#For more info and user input, open script A: userInput
#This script includes a combination of DIANA commands and python commands,
##and can not be run in any other program than DIANA.

### IMPORTING CLASSES; FUNCTIONS AND USERINPUT
if not 'parametric_Study' in locals():
    ##All scripts have to be saved in the same folder (Have the same working directory)
    wd = sys.path[0] #Working directory of main.py
    exec(open(wd+"\\C_classesAndFunctions.py").read())
    exec(open(wd+"\\A_UserInput.py").read())

    functionOfUserInput([0])
        ### CREATING PROJECT
    if not hasattr(Project, 'name'):
        #**Default projectname**:
        Project.name = Project.modelName + "_" + date + "_" + Project.modelExtraInfo
else:
    Options.parametricStudy = True

if not hasattr(Project, 'dirPath'):
        Project.dirPath = sys.path[0]

newProject( Project.dirPath + "\\" + Project.modelName + "\\" +Project.name, Project.size_in_m )
setModelAnalysisAspects( [ "STRUCT" ] )

setModelDimension(Options.dimension)
setDefaultMeshOrder( "QUADRATIC" ) #Default
setDefaultMesherType( "HEXQUAD" )#Default
setDefaultMidSideNodeLocation( "ONSHAP" )#Default

setUnit( "LENGTH", Units.length)
setUnit( "FORCE", Units.force )
setUnit( "MASS", Units.mass )
setUnit( "TEMPER", Units.temperature )

### Checking if template is used:
if Options.template_3PointBending == True or Options.template_4PointBending == True:
    Options.symmetric_Beam = True

### CREATING THE GEOMETRY
# -- The beam ---
#Creating the Geometry:
addSet( "SHAPESET", "Beam" )
remove( "SHAPESET", "Shapes" )
beam.x_coord_mid = beam.geometry.length/2

#Coordinates:
if Options.symmetric_Beam == True:
    beam.coord =[[    0,    0 ],
                [ beam.x_coord_mid, 0 ],
                [ beam.x_coord_mid, beam.geometry.height],
                [    0, beam.geometry.height ]]
else:
    beam.coord =[[    0,    0 ],
                [ beam.geometry.length, 0 ],
                [ beam.geometry.length, beam.geometry.height],
                [    0, beam.geometry.height ]]


for obj in Beam.allObjects:
    createSheet(obj.name, obj.coord)

# -- Plates ---
addSet( "SHAPESET", "Plates" )
if Options.template_3PointBending == True: #and Symmetric_Beam:

```

```
    xCoord_sup = (beam.geometry.length - beam.geometry.lengthOfSpan)/2
    xCoord_load = beam.geometry.length/2

    supPlate = Plates("S", "Support plate", xCoord_sup)
    supPlate.fixedTranslation_x = False
    loadPlate = Plates("L", "Loading plate", xCoord_load)


elif Options.template_4PointBending == True: #and Symmetric_Beam:

    xCoord_sup = (beam.geometry.length - beam.geometry.lengthOfSpan)/2
    xCoord_load = (beam.geometry.length - beam.geometry.lengthOfInnerSpan)/2

    supPlate = Plates("S", "Support plate", xCoord_sup)
    supPlate.fixedTranslation_x = False
    loadPlate = Plates("L", "Loading plate", xCoord_load)

#All plate objects have same geometry:
for obj in Plates.allObjects:
    obj.add_geometry(plateGeometry)

#Coordinates:
for obj in Plates.allObjects:

    if obj.typeOfPlate == "S":
        obj.coord = [[ obj.x_coord-(obj.geometry.length/2), 0 ],
                 [ obj.x_coord+(obj.geometry.length/2), 0 ],
                 [ obj.x_coord+(obj.geometry.length/2),  -obj.geometry.height],
                 [ obj.x_coord, -obj.geometry.height],
                 [ obj.x_coord-(obj.geometry.length/2), -obj.geometry.height ]]

    elif obj.typeOfPlate == "L":
        obj.coord = [[ obj.x_coord-(obj.geometry.length/2), beam.geometry.height ],
                 [ obj.x_coord+(obj.geometry.length/2), beam.geometry.height ],
                 [ obj.x_coord+(obj.geometry.length/2), beam.geometry.height+ obj.geometry.height ],
                 [ obj.x_coord, beam.geometry.height+ obj.geometry.height ],
                 [ obj.x_coord-(obj.geometry.length/2), beam.geometry.height+ obj.geometry.height  ]]

    setCurrentShapeSet("Plates")
    createSheet(obj.name, obj.coord)

    if Options.symmetric_Beam == True and obj.x_coord == beam.x_coord_mid:
        addSet( "SHAPESET", "DividerSet" )
        createLine( "Divider", [ obj.x_coord, -10000], [obj.x_coord, 10000 ] )
        dum = cut( obj.name, [ "Divider" ], False, True )
        removeShape(dum[1])
        renameShape(dum[0], obj.name)
        remove( "SHAPESET", "DividerSet" )

# -- Loads ---
#DEAD WEIGHT
if Options.includeSelfWeight: #only true is implemented
    selfweight = Selfweight()
    addSet( GEOMETRYLOADSET, selfweight.setname )
    createModelLoad( selfweight.name, selfweight.setname )

# #POINT LOAD:
#Load
addSet( "GEOMETRYLOADSET", Loads.setname )
for obj in Loads.allObjects:
    if Options.template_3PointBending == True or Options.template_4PointBending == True:
        obj.target = loadPlate
    obj.attachLoadToTargets((obj.target), beam)

# -- Longitudinal Reinfo ---
for obj in Longitudinal_Reinfo.allObjects:
    addSet( GEOMETRYREINFOSET, obj.setname)
    setCurrentShapeSet( obj.setname)
    if Options.symmetric_Beam == True:
        obj.x_coord_end = beam.x_coord_mid
    pos1 = [ obj.x_coord_start, obj.y_coord] #[x,y]
    pos2 = [ obj.x_coord_end, obj.y_coord] #[x,y]
    createLine(obj.name, pos1, pos2)

# -- Shear Reinfo ---
```

```python
for obj in Transverse_Reinfo.allObjects:
    bool_shearBarInMiddle = createShearReinfo(obj)

### Preparations before iterations
ListOfAllElements = Beam.allObjects + Plates.allObjects
ListOfAllElements_names = [obj.name for obj in ListOfAllElements]
ListOfAllPlates_names = [obj.name for obj in Plates.allObjects]
ListOfAllReinfoElements = Longitudinal_Reinfo.allObjects + Transverse_Reinfo.allObjects
longReinfo_names = [obj.name for obj in Longitudinal_Reinfo.allObjects]
transReinfo_names = Transverse_Reinfo.allNames
allReinfo_names = longReinfo_names + transReinfo_names

### MATERIAL MODELS
# -- Concrete ---
#Properties:
#Finding recommended properties
concreteFromEC2()
concreteFromFib2010()
if not hasattr(concrete, 'Youngs_modulus'):
    concrete.Youngs_modulus = parameter("MATERIAL","Concrete_EC2","EC2CON/NORMAL/PARAME/YOUDER")
if not hasattr(concrete, 'poissons_ratio'):
    concrete.poissons_ratio =  0.2 #RTD
if not hasattr(concrete, 'mass_density'):
    concrete.mass_density  = parameter("MATERIAL","Concrete_EC2" ,"EC2CON/NORMAL/PARAME/DENDER")
if not hasattr(concrete, 'tensile_strength'):
    concrete.tensile_strength = parameter("MATERIAL","Concrete_EC2" ,
    "EC2CON/NORMAL/PARAME/TOTCRK/TENDER")
if not hasattr(concrete, 'compressive_strength'):
    concrete.compressive_strength = parameter("MATERIAL","Concrete_EC2" ,
    "EC2CON/NORMAL/PARAME/TOTCRK/COMDER")
if not hasattr(concrete, 'tensile_FractureEnergy'):
    concrete.tensile_FractureEnergy = parameter("MATERIAL", "Concrete_fib2010",
    "MC10CO/NORMAL/PARAME/TOTCRK/GF1DER" ) #based upon fib Model Code
if not hasattr(concrete, 'compressive_FractureEnergy'):
    concrete.compressive_FractureEnergy = 250*((concrete.f_ck)/
    (concrete.compressive_strength))*0.073*(concrete.compressive_strength)**(0.18) #RTD

#Changing from Rotating to fixed crack model if no shear reinforcement:
if len(Transverse_Reinfo.allObjects) == 0:
    concrete.crack_orientation = "FIXED"

#Creating material:
addMaterial( concrete.Name, concrete.material, concrete.material_model, [])
setParameter( "MATERIAL", concrete.Name, "LINEAR/ELASTI/YOUNG", concrete.Youngs_modulus)
setParameter( "MATERIAL", concrete.Name, "LINEAR/ELASTI/POISON", concrete.poissons_ratio )
setParameter( "MATERIAL", concrete.Name, "LINEAR/MASS/DENSIT", concrete.mass_density )
setParameter( "MATERIAL", concrete.Name, "MODTYP/TOTCRK", concrete.crack_orientation)
if concrete.crack_orientation == "FIXED":
    setParameter( "MATERIAL", concrete.Name, "SHEAR/SHRCRV", "DAMAGE" )
setParameter( "MATERIAL", concrete.Name, "TENSIL/TENCRV", concrete.tensile_curve)
setParameter( "MATERIAL", concrete.Name, "TENSIL/TENSTR", concrete.tensile_strength)
setParameter( "MATERIAL", concrete.Name, "TENSIL/CBSPEC", concrete.crackBandwidt_specification )
setParameter( "MATERIAL", concrete.Name, "TENSIL/POISRE/POIRED", concrete.poissonsRatio_reductionModel)
setParameter( "MATERIAL", concrete.Name, "TENSIL/GF1", concrete.tensile_FractureEnergy)
setParameter( "MATERIAL", concrete.Name, "COMPRS/COMCRV", concrete.compression_curve)
setParameter( "MATERIAL", concrete.Name, "COMPRS/COMSTR", concrete.compressive_strength )
setParameter( "MATERIAL", concrete.Name, "COMPRS/GC", concrete.compressive_FractureEnergy )
setParameter( "MATERIAL", concrete.Name, "COMPRS/REDUCT/REDCRV", concrete.lateralCracking_reductionModel)
setParameter( "MATERIAL", concrete.Name, "COMPRS/REDUCT/REDMIN",
concrete.lateralCracking_reductionCurve_lowerBound)
setParameter( "MATERIAL", concrete.Name, "COMPRS/CONFIN/CNFCRV", concrete.confinementModel )

#Assigning geometry and material to beam
beam_geo = beam.name
beam_shape = beam.name

addGeometry( beam_geo, "SHEET", "MEMBRA", [] )
setParameter( "GEOMET", beam_geo, "THICK", beam.geometry.width )
setParameter( "GEOMET", beam_geo, "LOCAXS", True )
setParameter( "GEOMET", beam_geo, "LOCAXS/XAXIS", [ 1, 0, 0 ] )

setElementClassType( "SHAPE", [ beam_shape ], "MEMBRA" )
assignMaterial( concrete.Name, "SHAPE", [ beam_shape ] )
assignGeometry( beam_geo, "SHAPE", [ beam_shape ] )
```

```
#STEEL FOR SUPPORTS AND LOADING PLATES
steel_mat = "Steel Plates"
steel_geo = "Steel Plates"

addMaterial( steel_mat, Steel_Plates.material, Steel_Plates.material_model, [] )
setParameter( "MATERIAL", steel_mat, "LINEAR/MASS/DENSIT", Steel_Plates.density )
setParameter( "MATERIAL", steel_mat, "LINEAR/ELASTI/YOUNG", Steel_Plates.e_mod)
setParameter( "MATERIAL", steel_mat, "LINEAR/ELASTI/POISON", Steel_Plates.poissons_ratio )

addGeometry( steel_geo, "SHEET", "MEMBRA", [] )
plateGeometry.width = beam.geometry.width
setParameter( "GEOMET", steel_geo, "THICK", plateGeometry.width)

setElementClassType( "SHAPE",  ListOfAllPlates_names , "MEMBRA" )
isSteel = assignMaterial( steel_mat, "SHAPE", ListOfAllPlates_names )
assignGeometry( steel_geo, "SHAPE", ListOfAllPlates_names )
if isSteel:
    setShapeColor( "#cccccc", ListOfAllPlates_names )

#Integration schemes:
addElementData( "integrationScheme")
setParameter( "DATA", "integrationScheme", "INTEGR", "HIGH" )
assignElementData( "integrationScheme", "SHAPE", ListOfAllElements_names)

# -- Reinforcement ---
for obj in ListOfAllReinfoElements:
    obj.kapsig = [ 0, obj.f_ym, obj.e_u, obj.f_um] #For use in the bilinear stress-strain diagram

for obj in ListOfAllReinfoElements:
    addMaterial( obj.name, "REINFO", "VMISES", [] )
    setParameter( "MATERIAL", obj.name, "LINEAR/ELASTI/YOUNG", obj.e_mod )
    setParameter( "MATERIAL", obj.name, "PLASTI/YLDTYP", "KAPSIG" )
    setParameter( "MATERIAL", obj.name, "PLASTI/HARDI2/KAPSIG", [] )
    setParameter( "MATERIAL", obj.name, "PLASTI/HARDI2/KAPSIG", obj.kapsig )
    addGeometry( obj.name, "RELINE", "REBAR", [] )
    setParameter( "GEOMET", obj.name, "REIEMB/CROSSE", obj.As )

    setReinforcementType( "GEOMETRYREINFOSET", obj.setname, "BAR" )

    assignMaterial( obj.name, "GEOMETRYREINFOSET", obj.setname )
    assignGeometry( obj.name, "GEOMETRYREINFOSET", [ obj.setname ] )

    if obj.typeOfReinfo == "Trans":
        setReinforcementDiscretization( obj.allNames, "ELEMENT" )
        if bool_shearBarInMiddle:
            #Creating a shear bear with half the area in middle of beam,
            #so it will be correct when modelling symmetric
            assignMaterial( obj.name, "GEOMETRYREINFOSET", "ShearBar_mid")
            addGeometry( obj.name + "_mid" , "RELINE", "REBAR", [] )
            assignGeometry( obj.name + "_mid", "GEOMETRYREINFOSET", [ "ShearBar_mid" ] )
            setParameter( "GEOMET", obj.name + "_mid" , "REIEMB/CROSSE", obj.As/2 )
            setContinuousInInterfaces( "GEOMETRYREINFOSET", "ShearBar_mid", False )
    else:
        setReinforcementDiscretization( [ obj.name], "ELEMENT" )
    setContinuousInInterfaces( "GEOMETRYREINFOSET", obj.setname, False )

# -- Supports ---
if Options.symmetric_Beam == True:
    #Symmetric support in middle of beam
    nameOfSupport = "Symmetry"
    addSet( "GEOMETRYSUPPORTSET", "Symmetric supports" )
    createLineSupport( nameOfSupport, "Symmetric supports" )
    setParameter( "GEOMETRYSUPPORT", nameOfSupport, "AXES", [ 1, 2 ] )
    setParameter( "GEOMETRYSUPPORT", nameOfSupport, "TRANSL", [ 1, 0 ] )
    setParameter( "GEOMETRYSUPPORT", nameOfSupport, "ROTATI", [ 0, 0, 1] )
    attach( "GEOMETRYSUPPORT", nameOfSupport, beam.name, [ [ beam.x_coord_mid, beam.geometry.height/2]])
    for obj in Plates.allObjects:
        if obj.x_coord == beam.x_coord_mid:
            if obj.typeOfPlate == "S":
                attach( "GEOMETRYSUPPORT", nameOfSupport, obj.name, [ [ obj.x_coord,
                beam.geometry.height - obj.geometry.height/2 ] ] )
            elif obj.typeOfPlate == "L":
                attach( "GEOMETRYSUPPORT", nameOfSupport, obj.name, [ [ obj.x_coord,
                beam.geometry.height + obj.geometry.height/2 ] ] )
```

```
addSet( "GEOMETRYSUPPORTSET", "Supports" )
counter = 1
for obj in Plates.allObjects:
    if obj.typeOfPlate == "S":
        x_trans = 1
        y_trans = 1
        nameOfSupport = "BC" + str(counter)
        createPointSupport( nameOfSupport, "Supports" )
        setParameter( "GEOMETRYSUPPORT", nameOfSupport, "AXES", [ 1, 2 ] )
        print(x_trans)
        if not obj.fixedTranslation_x:
            x_trans = 0
        if not obj.fixedTranslation_y:
            y_trans = 0
        setParameter( "GEOMETRYSUPPORT", nameOfSupport, "TRANSL", [ x_trans, y_trans ] )
        setParameter( "GEOMETRYSUPPORT", nameOfSupport, "ROTATI", [ 0, 0, 0 ] )
        attach( "GEOMETRYSUPPORT", nameOfSupport, obj.name, [[ obj.x_coord, -obj.geometry.height]] )
        counter += 1


# -- Interface ---
# Imprint the plates into the beam
for obj in Plates.allObjects:
    imprintIntersection( beam.name , obj.name, True )

#Creating the interface material:
addMaterial( Interface.name, "INTERF", "NONLIF", [] ) #nonlinear relation
setParameter( MATERIAL, Interface.name, "LINEAR/IFTYP", "LIN2D" )
setParameter( MATERIAL, Interface.name, "LINEAR/ELAS2/DSNY", Interface.Knn_compression )
setParameter( MATERIAL, Interface.name, "LINEAR/ELAS2/DSSX", Interface.Kt )
setParameter( MATERIAL, Interface.name, "NONLIN/IFNOTE", "DIAGRM" )
setParameter( MATERIAL, Interface.name, "NONLIN/NLEL2/DUSTNY",
[ -1000, -Interface.Knn_compression*1e3 ,
        0,      0,
        1000,  Interface.Knn_tension*1e3        ] )

#Creating the Interface geometry:
addGeometry( Interface.name, "LINE", "STLIIF", [] )
setParameter( GEOMET, Interface.name, "LIFMEM/THICK", beam.geometry.width)

#creating the Interface
createConnection( Interface.name, "INTER", "SHAPEEDGE", "SHAPEEDGE" ) #Interface between two edges
setParameter( "GEOMETRYCONNECTION", Interface.name, "MODE", "CLOSED" )
setElementClassType( "GEOMETRYCONNECTION", Interface.name, "STLIIF" )

assignMaterial( Interface.name, "GEOMETRYCONNECTION", Interface.name )
assignGeometry( Interface.name, "GEOMETRYCONNECTION", Interface.name )
setParameter( "GEOMETRYCONNECTION", Interface.name, "FLIP", False )
for obj in Plates.allObjects:
    if obj.x_coord == beam.x_coord_mid and Options.symmetric_Beam == True:
        if obj.typeOfPlate == "S":
            attachTo( "GEOMETRYCONNECTION", Interface.name, "SOURCE", obj.name,
            [[ obj.x_coord - obj.geometry.length/4, 0 ]] )
            attachTo( "GEOMETRYCONNECTION", Interface.name, "TARGET", beam.name,
            [[ obj.x_coord - obj.geometry.length/4, 0 ]] )
        elif obj.typeOfPlate == "L":
            attachTo( "GEOMETRYCONNECTION", Interface.name, "SOURCE", obj.name,
            [ [ obj.x_coord - obj.geometry.length/4, beam.geometry.height ] ] )
            attachTo( "GEOMETRYCONNECTION", Interface.name, "TARGET", beam.name,
            [[ obj.x_coord - obj.geometry.length/4, beam.geometry.height ]] )
    elif obj.typeOfPlate == "S":
        attachTo( "GEOMETRYCONNECTION", Interface.name, "SOURCE", obj.name, [[ obj.x_coord, 0 ]] )
        attachTo( "GEOMETRYCONNECTION", Interface.name, "TARGET", beam.name, [[ obj.x_coord, 0 ]] )
    elif obj.typeOfPlate == "L":
        attachTo( "GEOMETRYCONNECTION", Interface.name, "SOURCE", obj.name,
        [ [ obj.x_coord, beam.geometry.height ] ] )
        attachTo( "GEOMETRYCONNECTION", Interface.name, "TARGET", beam.name,
        [[ obj.x_coord, beam.geometry.height ]] )

#Adding element data to Interface
addElementData( "InterfaceData" )
setParameter( DATA, "InterfaceData", "INTEGR", "HIGH" )
assignElementData( "InterfaceData", "GEOMETRYCONNECTION", Interface.name )

# -- Mesh ---
```

```python
setElementSize( ListOfAllElements_names , Mesh.elementsize, -1, True )
setMesherType( ListOfAllElements_names , Mesh.meshertype )
#setMidSideNodeLocation( ListOfAllElements_names, Mesh.midSideNodeLocation )
generateMesh( [] )

# -- Linear Analysis ---
for analysis in Analysis.allObjects:
    if analysis.typeOfAnalysis == "Linear":

        addAnalysis( analysis.name)
        addAnalysisCommand( analysis.name, "LINSTA", analysis.command)
        setAnalysisCommandDetail(  analysis.name, analysis.command, "SOLVE/TYPE", analysis.method)
        setAnalysisCommandDetail(  analysis.name, analysis.command, "OUTPUT(1)/SELTYP", analysis.output)


        if analysis.runSolver == True:
            runSolver( [ analysis.name ] )

            if hasattr(analysis, 'expectedFailureLoad'):
                [midNodeBeam] = findNodesCloseTo(((beam.geometry.length)/2,0))
                #Displacement at midpoint:
                results_table = resultsTable({"analysis": analysis.name, "result": "Displacements",
                "components": ["DtY"], "nodes": [midNodeBeam], "cases": (resultCases(analysis.name))})
                dispAtExpectedLoad = analysis.expectedFailureLoad*results_table[2][2]
                print(dispAtExpectedLoad)


    # -- Nonlinear Analysis ---
    # First selfweight, then point loads.
    elif analysis.typeOfAnalysis == "Nonlinear":

        addAnalysis( analysis.name )
        addAnalysisCommand( analysis.name, "NONLIN", analysis.command )

        #Dead weight
        #All self_weight is applied in one increment
        renameAnalysisCommandDetail( analysis.name, analysis.command, "EXECUT(1)", selfweight.name)
        addAnalysisCommandDetail( analysis.name, analysis.command, "EXECUT(1)/LOAD/LOADNR" )
        setAnalysisCommandDetail( analysis.name, analysis.command, "EXECUT(1)/LOAD/LOADNR", 1)

        #Point loads
        setAnalysisCommandDetail( analysis.name, analysis.command, "EXECUT/EXETYP", "LOAD" )
        renameAnalysisCommandDetail( analysis.name, analysis.command, "EXECUT(2)", "Loads" )
        addAnalysisCommandDetail( analysis.name, analysis.command, "EXECUT(2)/LOAD/LOADNR" )
        setAnalysisCommandDetail( analysis.name, analysis.command, "EXECUT(2)/LOAD/LOADNR", 2)

        for i in [1,2]: #only two phases
            setAnalysisCommandDetail( analysis.name, analysis.command, "EXECUT("+str(i)+")/ITERAT/MAXITE",
            analysis.iteration.nrOfIterations )
            if analysis.iteration.method == "NEWTON-RAPHSON":
                setAnalysisCommandDetail( analysis.name, analysis.command,
                "EXECUT("+str(i)+")/ITERAT/METHOD/METNAM", "NEWTON" )
                setAnalysisCommandDetail( analysis.name, analysis.command,
                "EXECUT("+str(i)+")/ITERAT/METHOD/NEWTON/TYPNAM", analysis.iteration.typeOfMethod )
                setAnalysisCommandDetail( analysis.name, analysis.command,
                "EXECUT("+str(i)+")/ITERAT/LINESE", analysis.lineSearch ) #Bool value

            if analysis.convergence.useForceNorm == True:
                setAnalysisCommandDetail( analysis.name, analysis.command,
                "EXECUT("+str(i)+")/ITERAT/CONVER/FORCE", True )
                setAnalysisCommandDetail( analysis.name, analysis.command,
                "EXECUT(1)/ITERAT/CONVER/FORCE/TOLCON", analysis.convergence.forceNorm_deadLoad)
                setAnalysisCommandDetail( analysis.name, analysis.command,
                "EXECUT(2)/ITERAT/CONVER/FORCE/TOLCON", analysis.convergence.forceNorm )
                if analysis.continueIfNoConvergence == True:
                    setAnalysisCommandDetail( analysis.name, analysis.command,
                    "EXECUT("+str(i)+")/ITERAT/CONVER/FORCE/NOCONV", "CONTIN" )
                else:
                    setAnalysisCommandDetail( analysis.name, analysis.command,
                    "EXECUT("+str(i)+")/ITERAT/CONVER/FORCE/NOCONV", "TERMIN" )
            else:
                setAnalysisCommandDetail( analysis.name, analysis.command,
                "EXECUT("+str(i)+")/ITERAT/CONVER/FORCE", False )

            if analysis.convergence.useEnergyNorm == True:
```

```
                setAnalysisCommandDetail( analysis.name, analysis.command,
                "EXECUT("+str(i)+")/ITERAT/CONVER/ENERGY", True )
                setAnalysisCommandDetail( analysis.name, analysis.command,
                "EXECUT(1)/ITERAT/CONVER/ENERGY/TOLCON", analysis.convergence.energyNorm_deadLoad)
                setAnalysisCommandDetail( analysis.name, analysis.command,
                "EXECUT(2)/ITERAT/CONVER/ENERGY/TOLCON", analysis.convergence.energyNorm)
                if analysis.continueIfNoConvergence == True:
                    setAnalysisCommandDetail( analysis.name, analysis.command,
                    "EXECUT("+str(i)+")/ITERAT/CONVER/ENERGY/NOCONV", "CONTIN" )
                else:
                    setAnalysisCommandDetail( analysis.name, analysis.command,
                    "EXECUT("+str(i)+")/ITERAT/CONVER/ENERGY/NOCONV", "TERMIN" )
            else:
                setAnalysisCommandDetail( analysis.name, analysis.command,
                "EXECUT("+str(i)+")/ITERAT/CONVER/ENERGY", False )
            #OBS!!! IF dispNorm is removed, you still need to keep the line with dispNorm = False
            if analysis.convergence.useDispNorm == True:
                setAnalysisCommandDetail( analysis.name, analysis.command,
                "EXECUT("+str(i)+")/ITERAT/CONVER/DISPLA", True )
                setAnalysisCommandDetail( analysis.name, analysis.command,
                "EXECUT(1)/ITERAT/CONVER/DISPLA/TOLCON", analysis.convergence.dispNorm_deadLoad)
                setAnalysisCommandDetail( analysis.name, analysis.command,
                "EXECUT(2)/ITERAT/CONVER/DISPLA/TOLCON", analysis.convergence.dispNorm)
                if analysis.continueIfNoConvergence == True:
                    setAnalysisCommandDetail( analysis.name, analysis.command,
                    "EXECUT("+str(i)+")/ITERAT/CONVER/DISPLA/NOCONV", "CONTIN" )
                else:
                    setAnalysisCommandDetail( analysis.name, analysis.command,
                    "EXECUT("+str(i)+")/ITERAT/CONVER/DISPLA/NOCONV", "TERMIN" )
            else:
                 setAnalysisCommandDetail( analysis.name, analysis.command,
                 "EXECUT("+str(i)+")/ITERAT/CONVER/DISPLA", False ) #This line have to be kept!!!

            if analysis.allConvergenceNormsHaveToBeSatisfied == True:
                setAnalysisCommandDetail(analysis.name, analysis.command,
                "EXECUT("+str(i)+")/ITERAT/CONVER/SIMULT", True )

        ## Point loads:
        if analysis.incrementation.method == "ENERGY" and analysis.arcLengthControl == True:
            setAnalysisCommandDetail( analysis.name, analysis.command,
            "EXECUT(2)/LOAD/STEPS/STEPTY", "ENERGY" )
            setAnalysisCommandDetail( analysis.name, analysis.command,
            "EXECUT(2)/LOAD/STEPS/ENERGY/INISIZ", analysis.incrementation.initial_step_size )
            setAnalysisCommandDetail( analysis.name, analysis.command,
            "EXECUT(2)/LOAD/STEPS/ENERGY/MAXSIZ", analysis.incrementation.max_step_size )
            setAnalysisCommandDetail( analysis.name, analysis.command,
            "EXECUT(2)/LOAD/STEPS/ENERGY/MINSIZ", analysis.incrementation.min_step_size )
            setAnalysisCommandDetail( analysis.name, analysis.command,
            "EXECUT(2)/LOAD/STEPS/ENERGY/NSTEPS", analysis.incrementation.nrOfSteps )
            setAnalysisCommandDetail( analysis.name, analysis.command,
            "EXECUT(2)/LOAD/STEPS/ENERGY/ARCLEN", True )
            addAnalysisCommandDetail( analysis.name, analysis.command,
            "EXECUT(2)/LOAD/STEPS/ENERGY/ARCLEN/REGULA/SET" )
            #Arc length control is done over whole beam.
            nodes = list( nodeIds( ELEMENTSET, beam.name) )
            setAnalysisCommandDetail( analysis.name, analysis.command,
            "EXECUT(2)/LOAD/STEPS/ENERGY/ARCLEN/REGULA/SET(1)/NODES(1)/RNGNRS", nodes)
            setAnalysisCommandDetail( analysis.name, analysis.command,
            "EXECUT(2)/LOAD/STEPS/ENERGY/ARCLEN/REGULA/SET(1)/DIRECT", 2 ) #y-dir
            # Energy based method has to be combined with arc length

        #Output:
        setAnalysisCommandDetail( analysis.name, analysis.command, "OUTPUT(1)/SELTYP", "USER" )
        addAnalysisCommandDetail( analysis.name, analysis.command, "OUTPUT(1)/USER" )

        output_commands = {
            "DISPLA TOTAL TRANSL GLOBAL" : "OUTPUT(1)/USER/DISPLA(1)/TOTAL" ,
            "FORCE REACTI TRANSL GLOBAL" : "OUTPUT(1)/USER/FORCE(1)/REACTI",
            "STRAIN TOTAL GREEN GLOBAL" : "OUTPUT(1)/USER/STRAIN(1)/TOTAL/GREEN",
            "STRAIN TOTAL GREEN PRINCI" : "OUTPUT(1)/USER/STRAIN(2)/TOTAL/GREEN/PRINCI",
            "STRAIN CRKWDT GREEN GLOBAL" : "OUTPUT(1)/USER/STRAIN(3)/CRKWDT",
            "STRAIN CRACK GREEN" : "OUTPUT(1)/USER/STRAIN(4)/CRACK/GREEN" ,
            "STRAIN CRKWDT GREEN PRINCI" : "OUTPUT(1)/USER/STRAIN(5)/CRKWDT/GREEN/PRINCI",
            "STRESS TOTAL CAUCHY GLOBAL" : "OUTPUT(1)/USER/STRESS(1)/TOTAL/CAUCHY",
            "STRESS TOTAL CAUCHY PRINCI" : "OUTPUT(1)/USER/STRESS(2)/TOTAL/CAUCHY/PRINCI"
```

```
        }

        for key, boolValue in analysis.output.items():
            if boolValue == True:
                addAnalysisCommandDetail( analysis.name, analysis.command, output_commands[key] )

        saveProject()

        if analysis.runSolver == True:
            runSolver( [ analysis.name] )

            if hasattr(LoadDispY_Graph, 'x_coord'):
                loadDispYPlot(LoadDispY_Graph.x_coord,LoadDispY_Graph.y_coord, analysis,
                Scalefactor.loadFactor_plot, Scalefactor.displacement_plot)

            if hasattr(LoadDispY_CSV, 'x_coord'):
                getLoadDisplacementCSV(LoadDispY_CSV.x_coord,LoadDispY_CSV.y_coord, analysis,
                Scalefactor.loadFactor_CSV, Scalefactor.displacement_CSV, LoadDispY_CSV.decimalPlaces)

saveProject()
```

# E Case B1 - User input

```
def functionOfUserInput(parametricValue): #Input inside function allows for parametric study
    #------------------------------------------------------------------
    #--------- CASE B1----------------------------------------
    #-------------------------------------------------------------------

    ###4. CREATING PROJECT
    Project.modelName = "B1"
    Project.modelExtraInfo = "final"

    Project.size_in_m = 1000

    #-#4.1 Modelling choices
    Options.symmetric_Beam =  True #**Optional**, default is False

    ###5. CREATING THE GEOMETRY
    #-#5.1 The beam
    #'#5.1.1 Geometry:
    beam = Beam() #creating beam object
    beamGeometry = Geometry() #creating geometry object
    beam.add_geometry(beamGeometry) #adding the geometry to the beam

    beam.geometry.length = 6840 #[mm]
    beam.geometry.height = 552 #[mm]
    beam.geometry.width = 152 #[mm] #thickness

    #-#5.2 Reinforcement
    #'#5.2.1 Cover: #**Optional input**
    cover = Cover() #creating cover object
    # ####Option 1**: definition of each cover
    cover.side = 48 #[mm]
    cover.top = 50 #[mm]
    cover.bot = 64 #[mm]

    #'#5.2.2 User defined variables

    #-#5.3 Longitudinal reinforcement
    #'#5.3.1 Creation of longitudinal reinforcement
    ##CREATE YOUR LONG.REINFO OBJECTS HERE
    #\\ Reinforcement properties based on DIANA Verification & Rijkwaterstaat

    upper_reinfo = Longitudinal_Reinfo("M10", 300, 315, 460, 2.3425E-2)
    upper_reinfo.y_coord = beam.geometry.height - cover.top
    upper_reinfo.x_coord_start = cover.side
    # templateName.x_coord_end = #**Input needed if beam is not symmetric**

    lower_reinfo_M30 = Longitudinal_Reinfo("M30", 1400, 436, 700, 4.78E-2)
    lower_reinfo_M30.y_coord = cover.bot
    lower_reinfo_M30.x_coord_start = 0
    # templateName.x_coord_end = #**Input needed if beam is not symmetric**

    lower_reinfo_M25 = Longitudinal_Reinfo("M25", 1000, 445, 680, 4.80E-2, 220000)
    lower_reinfo_M25.y_coord = 2*cover.bot
    lower_reinfo_M25.x_coord_start = 0
    # templateName.x_coord_end = #**Input needed if beam is not symmetric**

    #-#5.4 Transverse reinforcement (shear)
    #'#5.4.1 Creation of transverse reinforcement
    ##CREATE YOUR TRANS. REINFO OBJECTS HERE
    shear_reinfo = Transverse_Reinfo("D4_shear", 25.7, 600, 651, 4.70E-2)
    shear_reinfo.y_coord_bot = cover.bot
    shear_reinfo.y_coord_top = beam.geometry.height - cover.top

    #'#5.4.2 Spacing/positions of transverse reinforcement:
    ##CREATE YOUR SECTION OBJECTS HERE
    spacing_large = 168
    spacing_small = 86
     section_1  = Section(spacing_small, cover.side, cover.side + 4*spacing_small)
     section_2  = Section(spacing_large, cover.side + 4*spacing_small,
    (beam.geometry.length/2) - 2*spacing_small)
     section_3  = Section(spacing_small, (beam.geometry.length/2) - 2*spacing_small,
    beam.geometry.length/2)

```

```
#-#5.5 Plates
plateGeometry = Geometry() #creating geometry object for plates

plateGeometry.length = 150  #[mm]
plateGeometry.height = 35 #[mm]

#'#5.5.1 Option 1: using template for 3-point bending
Options.template_3PointBending = True
beam.geometry.lengthOfSpan = 6400 #[mm]

###6. LOADS
#-#6.1 Option 1: using template for 3- or 4-point bending
##Both loads are assumed to have same value for 4-point bending. Only one has to be defined.
load = Loads("Point", "Load", -1000, "Y")

###7. MATERIAL MODELS
#-#7.1 Concrete material model
#The concrete material is defined as follows: concrete = Concrete(f_ck)
concrete = Concrete(35)

#'#7.1.1 Concrete properties:
## The following properties can be changed:
concrete.compressive_strength =  43.5 #**Optional**

#-#7.2 Steel plates material model
Steel_Plates.e_mod = 2000000
Steel_Plates.poissons_ratio = 0.3

#-#7.3 Reinforcement material model

###8. STRUCTURAL INTERFACES
## The following properties can be changed:
Interface.Knn_tension     = 3.63e-08   # [N/mm3] **Optional**
Interface.Knn_compression = 3.63e+04 # [N/mm3] **Optional**
Interface.Kt              = 3.63e-08 # [N/mm3] **Optional**

###9. MESH
Mesh.elementsize = 25 #[mm]

###10. ANALYSIS
#-#10.1 Linear Analysis
LFEA = Analysis("Linear analysis", "Linear") #creating linear static analysis

#'#10.1.1 RUN LINEAR ANALYSIS
# LFEA.runSolver = True #**Optional**

#-#10.2 Nonlinear Analysis
NLFEA = Analysis("Nonlinear analysis 1", "Nonlinear") #creating nonlinear analysis

#'#10.2.1 Arch length control:
NLFEA.arcLengthControl = True #**Default and strongly recommended as true**

#'#10.2.2 Load incrementation:
incrementation = Incrementation() #Creating an incrementation object
NLFEA.setIncrementalMethod(incrementation) #adding incremental method to the analysis

##Option 1: Energy based adaptive loading
##The energy based method MUST be combined with arch length control.
incrementation.method = "ENERGY"
incrementation.initial_step_size = 5    #initial size for the first step
incrementation.max_step_size = 10       #upper limit of yhe step size
incrementation.min_step_size = 3        #lower limit of the step size
incrementation.nrOfSteps = 75          #maximum number of steps

#'#10.2.3 Iteration method:
iteration = Iteration() #creating iteration object
NLFEA.setIterationMethod(iteration) #adding iterative procedure to the analysis

## Option 1: Regular Newton-Raphson
iteration.method = "NEWTON-RAPHSON"
iteration.typeOfMethod = "REGULA"
iteration.nrOfIterations = 100

#'#10.2.4 Line search:
##Using a line search algorithm is recommended.
```

```python
    NLFEA.lineSearch = True #**Default and recommended as true**

    #'#10.2.5 Convergence criteria:
    convergence = Convergence()          #creating convergence object
    NLFEA.setConvergence(convergence)   #adding convergence criteria to the analysis

    convergence.useForceNorm = True      #**Default and recommended as true**
    convergence.useEnergyNorm = True     #**Default and recommended as true**
    convergence.useDispNorm = False      #**Default and recommended as false**

    ## Option 1: Suggested tolerances
    ##Force norm:
    convergence.forceNorm = 0.01
    convergence.forceNorm_deadLoad = 0.05
    ##Energy norm:
    convergence.energyNorm = 0.001
    convergence.energyNorm_deadLoad = 0.01

    #'#10.2.6 Additional choices for analysis
    NLFEA.allConvergenceNormsHaveToBeSatisfied = False  #**Default and recommended as false**
    NLFEA.continueIfNoConvergence = True                #**Default and recommended as true**

    #'#10.2.7 RUN NONLINEAR ANALYSIS
    ##To run the analysis, uncomment the following line:
    NLFEA.runSolver = True #**optional**

    ###11. OUTPUTS FROM NONLINEAR ANALYSIS
    #-#11.1 Analysis output:
    NLFEA.output = {
        "DISPLA TOTAL TRANSL GLOBAL" : True,    #**Optional**
        "FORCE REACTI TRANSL GLOBAL" : True,    #**Optional**
        "STRAIN TOTAL GREEN GLOBAL" : True,     #**Optional**
        "STRAIN TOTAL GREEN PRINCI" : True,     #**Optional**
        "STRAIN CRKWDT GREEN GLOBAL" : True,    #**Optional**
        "STRAIN CRACK GREEN" : True,            #**Optional**
        "STRAIN CRKWDT GREEN PRINCI" : True,    #**Optional**
        "STRESS TOTAL CAUCHY GLOBAL" : True,    #**Optional**
        "STRESS TOTAL CAUCHY PRINCI" : True     #**Optional**
    }
    #----------------------------------------------------------------------------------------
    globals().update(locals()) #Allows main script to access all variables in function as globals
    return
```

# F Case B2 - User input

```
def functionOfUserInput(parametricValue): #Input inside function allows for parametric study
    #---------------------------------------------------------------
    #--------- CASE B2 --------------------------------------
    #----------------------------------------------------------------

    ###4. CREATING PROJECT
    Project.modelName = "RB2"
    Project.modelExtraInfo = "final"

    Project.size_in_m = 1000

    #-#4.1 Modelling choices

    ###5. CREATING THE GEOMETRY
    #-#5.1 The beam
    #'#5.1.1 Geometry:
    beam = Beam() #creating beam object
    beamGeometry = Geometry() #creating geometry object
    beam.add_geometry(beamGeometry) #adding the geometry to the beam

    beam.geometry.length = 5000 #[mm]
    beam.geometry.height = 500 #[mm]
    beam.geometry.width = 169 #[mm] #thickness

    #-#5.2 Reinforcement
    #'#5.2.1 Cover: #**Optional input**
    cover = Cover() #creating cover object

    ##**Option 2**: all sides have same cover:
    cover.length = 33 #[mm]
    cover.side = cover.top = cover.bot = cover.length

    #'#5.2.2 User defined variables
    diameterOfBars =  16#**Optional**

    #-#5.3 Longitudinal reinforcement
    #'#5.3.1 Creation of longitudinal reinforcement
    ##CREATE YOUR LONG.REINFO OBJECTS HERE
    #
     reinfo1  = Longitudinal_Reinfo("Reinfo", 2*200, 400, 600, 0.05)
     reinfo1 .y_coord = cover.bot
     reinfo1 .x_coord_start = cover.side
     reinfo1 .x_coord_end = beam.geometry.length - cover.side

    #
     reinfo2  = Longitudinal_Reinfo("Reinfo2", 2*200, 400, 600, 0.05)
     reinfo2 .y_coord = cover.bot + diameterOfBars
     reinfo2 .x_coord_start = cover.side
     reinfo2 .x_coord_end = beam.geometry.length - cover.side

    #
     reinfo3  = Longitudinal_Reinfo("Reinfo3", 2*200, 400, 600, 0.05)
     reinfo3 .y_coord = beam.geometry.height - cover.top
     reinfo3 .x_coord_start = cover.side
     reinfo3 .x_coord_end = beam.geometry.length - cover.side

    #
     reinfo4  = Longitudinal_Reinfo("Reinfo4", 2*200, 400, 600, 0.05)
     reinfo4 .y_coord = beam.geometry.height - cover.top - diameterOfBars
     reinfo4 .x_coord_start = cover.side
     reinfo4 .x_coord_end = beam.geometry.length - cover.side

    #
     reinfo5  = Longitudinal_Reinfo("Reinfo5", 2*200, 400, 600, 0.05)
     reinfo5 .y_coord = beam.geometry.height - cover.top
     reinfo5 .x_coord_start = 850
     reinfo5 .x_coord_end = 1850

    #
     reinfo6  = Longitudinal_Reinfo("Reinfo6", 2*200, 400, 600, 0.05)
     reinfo6 .y_coord = beam.geometry.height - cover.top - diameterOfBars
     reinfo6 .x_coord_start = 850
```

```
    reinfo6 .x_coord_end = 1850

   #
    reinfo7  = Longitudinal_Reinfo("Reinfo7", 2*200, 400, 600, 0.05)
    reinfo7 .y_coord = cover.bot
    reinfo7 .x_coord_start = 3150
    reinfo7 .x_coord_end = 4150

   #
    reinfo8  = Longitudinal_Reinfo("Reinfo8", 2*200, 400, 600, 0.05)
    reinfo8 .y_coord = cover.bot + diameterOfBars
    reinfo8 .x_coord_start = 3150
    reinfo8 .x_coord_end = 4150

   #-#5.4 Transverse reinforcement (shear)
   #\\ No shear reinfo

   #-#5.5 Plates
   plateGeometry = Geometry() #creating geometry object for plates

   plateGeometry.length = 76  #[mm]
   plateGeometry.height = 25 #[mm]

   #'#5.5.3 Option 3: Creation of individual plates
   ##CREATE YOUR PLATE OBJECTS HERE (option 3)
   sup_plate1= Plates("S", "Support plate 1", 1350)
   load_plate1= Plates("L", "Loading plate 1", 200)
   sup_plate2= Plates("S", "Support plate 2", beam.geometry.length - 200)
   load_plate2= Plates("L", "Loading plate 2", 3650)
   sup_plate2fixedTranslation_x = False

   ###6. LOADS
   #-#6.2 Option 2: Creation of individual loads
   ##CREATE YOUR LOAD OBJECTS HERE (option 2)
    load1  = Loads("Point", "Load 1", -1000, "Y")
    load1 .target = load_plate1
    load2  = Loads("Point", "Load 2", -2000, "Y")
    load2 .target = load_plate2

   ###7. MATERIAL MODELS
   #-#7.1 Concrete material model
   concrete = Concrete(45)

   #'#7.1.1 Concrete properties:
   concrete.compressive_strength =  53 #**Optional**
   concrete.lateralCracking_reductionCurve_lowerBound =  0.6 #**Optional**

   #-#7.2 Steel plates material model
   Steel_Plates.e_mod = 2000000
   Steel_Plates.poissons_ratio = 0.3

   #-#7.3 Reinforcement material model

   ###8. STRUCTURAL INTERFACES
   ## The following properties can be changed:
   Interface.Knn_tension     =  3.63e-08  # [N/mm3] **Optional**
   Interface.Knn_compression =   3.63e+04 # [N/mm3] **Optional**
   Interface.Kt              =   3.63e-08 # [N/mm3] **Optional**

   ###9. MESH
   Mesh.elementsize = 25 #[mm]

   ###10. ANALYSIS

   #-#10.1 Linear Analysis
   ##DIANA Primary output is chosen for the linear analysis.
   LFEA = Analysis("Linear analysis", "Linear") #creating linear static analysis

   #'#10.1.1 RUN LINEAR ANALYSIS
   LFEA.runSolver = True #**Optional**

   #-#10.2 Nonlinear Analysis
   NLFEA = Analysis("Nonlinear analysis 1", "Nonlinear") #creating nonlinear analysis

   #'#10.2.1 Arch length control:
```

```
NLFEA.arcLengthControl = True #**Default and strongly recommended as true**

#'#10.2.2 Load incrementation:
incrementation = Incrementation() #Creating an incrementation object
NLFEA.setIncrementalMethod(incrementation) #adding incremental method to the analysis

##Option 1: Energy based adaptive loading
incrementation.method = "ENERGY"
incrementation.initial_step_size = 5    #initial size for the first step
incrementation.max_step_size = 10       #upper limit of yhe step size
incrementation.min_step_size = 0.5        #lower limit of the step size
incrementation.nrOfSteps = 130          #maximum number of steps

#'#10.2.3 Iteration method:
iteration = Iteration() #creating iteration object
NLFEA.setIterationMethod(iteration) #adding iterative procedure to the analysis

## Option 1: Regular Newton-Raphson
iteration.method = "NEWTON-RAPHSON"
iteration.typeOfMethod = "REGULA"
iteration.nrOfIterations = 100

#'#10.2.4 Line search:
NLFEA.lineSearch = True #**Default and recommended as true**

#'#10.2.5 Convergence criteria:
convergence = Convergence()         #creating convergence object
NLFEA.setConvergence(convergence)   #adding convergence criteria to the analysis

convergence.useForceNorm = True     #**Default and recommended as true**
convergence.useEnergyNorm = True    #**Default and recommended as true**
convergence.useDispNorm = False     #**Default and recommended as false**


## Option 1: Suggested tolerances
##Force norm:
convergence.forceNorm = 0.01
convergence.forceNorm_deadLoad = 0.05
##Energy norm:
convergence.energyNorm = 0.001
convergence.energyNorm_deadLoad = 0.01

#'#10.2.6 Additional choices for analysis
NLFEA.allConvergenceNormsHaveToBeSatisfied = False  #**Default and recommended as false**
NLFEA.continueIfNoConvergence = True                #**Default and recommended as true**

#'#10.2.7 RUN NONLINEAR ANALYSIS
NLFEA.runSolver = True #**optional**

###11. OUTPUTS FROM NONLINEAR ANALYSIS
#-#11.1 Analysis output:
NLFEA.output = {
    "DISPLA TOTAL TRANSL GLOBAL" : True,    #**Optional**
    "FORCE REACTI TRANSL GLOBAL" : True,    #**Optional**
    "STRAIN TOTAL GREEN GLOBAL" : True,     #**Optional**
    "STRAIN TOTAL GREEN PRINCI" : True,     #**Optional**
    "STRAIN CRKWDT GREEN GLOBAL" : True,    #**Optional**
    "STRAIN CRACK GREEN" : True,            #**Optional**
    "STRAIN CRKWDT GREEN PRINCI" : True,    #**Optional**
    "STRESS TOTAL CAUCHY GLOBAL" : True,    #**Optional**
    "STRESS TOTAL CAUCHY PRINCI" : True     #**Optional**
}

#-#11.2 Load-displacement graph (displacement in y-dir) **Optional**
##Specify the coordinates of the point at where the displacement will be retrived.
LoadDispY_Graph.x_coord =  load_plate1x_coord #[mm]
LoadDispY_Graph.y_coord = beam.geometry.height + plateGeometry.height #[mm]

##The following options can be changed:
LoadDispY_Graph.xlabel = "Deflection (mm)"#**optional**
LoadDispY_Graph.ylabel = "Load (kN)"#**optional**
LoadDispY_Graph.xLim = [0,1.5]#**optional**
LoadDispY_Graph.yLim = [0,120] #**optional**

#-#11.3 Load-displacement CSV (displacement in y-dir)**Optional**
```

```python
    ##Specify the coordinates of the point for which the CSV-file will be generated:
    LoadDispY_CSV.x_coord = load_plate1x_coord #[mm]
    LoadDispY_CSV.y_coord = beam.geometry.height + plateGeometry.height #[mm]

    #-----------------------------------------------------------------------------------------
    globals().update(locals()) #Allows main script to access all variables in function as globals
    return
```

# G Parametric study - User input

```python
def functionOfUserInput(parametricValue): #Input inside function allows for parametric study
    #-------------------------------------------------------------------
    #--------- PARAMETRUC STUDY OF BEAM LENGTH --------------------------------------
    #--------------------------------------------------------------------

    ###4. CREATING PROJECT
    Project.modelName = "Parametric"
    Project.modelExtraInfo = "final"

    Project.size_in_m = 1000

    #-#4.1 Modelling choices
    Options.symmetric_Beam =  True #**Optional**, default is False

    ###5. CREATING THE GEOMETRY
    #-#5.1 The beam
    #'#5.1.1 Geometry:
    beam = Beam() #creating beam object
    beamGeometry = Geometry() #creating geometry object
    beam.add_geometry(beamGeometry) #adding the geometry to the beam

    beam.geometry.length = parametricValue #[mm]
    beam.geometry.height = 400 #[mm]
    beam.geometry.width = 250 #[mm] #thickness

    #-#5.2 Reinforcement
    #'#5.2.1 Cover: #**Optional input**
    cover = Cover() #creating cover object

    ##**Option 2**: all sides have same cover:
    cover.length = 35 #[mm]
    cover.side = cover.top = cover.bot = cover.length

    #'#5.2.2 User defined variables
    #\\ X_coordinate of first stirrup:
    #\\ Parameter to maintain the symmetry plane at midpoint for stirrups as well:
    a = beam.geometry.length/2
    firstPOS = a - floor(a/240)*240 #\\added by user

    #-#5.3 Longitudinal reinforcement
    #'#5.3.1 Creation of longitudinal reinforcement
    ##CREATE YOUR LONG.REINFO OBJECTS HERE

     reinfo2  = Longitudinal_Reinfo("Lower", 942, 500, 540, 0.05) #\\ 3 kam20 using f_yk,f_uk and ep_uk
     reinfo2 .y_coord = cover.bot
     reinfo2 .x_coord_start = cover.side
    #reinfo2.x_coord_end = beam.geometry.length - cover.side #**Input needed if beam is not symmetric**

    #-#5.4 Transverse reinforcement (shear)
    #'#5.4.1 Creation of transverse reinforcement
    ##CREATE YOUR TRANS. REINFO OBJECT HERE
    shear_reinfo =  Transverse_Reinfo("Shear_reinfo", 25.7, 600, 651, 4.70E-2,220000)
    shear_reinfo.y_coord_bot = cover.bot
    shear_reinfo.y_coord_top = beam.geometry.height - cover.top

    #'#5.4.2 Spacing/positions of transverse reinforcement:
    ##CREATE YOUR SECTION OBJECTS HERE
    section = Section(240, firstPOS, (beam.geometry.length - cover.side))

    #-#5.5 Plates
    plateGeometry = Geometry() #creating geometry object for plates

    plateGeometry.length = 150  #[mm]
    plateGeometry.height = 35 #[mm]

    #'#5.5.1 Option 1: using template for 3-point bending
    Options.template_3PointBending = True
    beam.geometry.lengthOfSpan = beam.geometry.length - 200#[mm]

    ###6. LOADS
    #-#6.1 Option 1: using template for 3- or 4-point bending
    ##Both loads are assumed to have same value for 4-point bending. Only one has to be defined.
```

```
    load = Loads("Point", "Load", -1000, "Y")

    ###7. MATERIAL MODELS
    #-#7.1 Concrete material model
    concrete = Concrete(30)

    #-#7.2 Steel plates material model
    Steel_Plates.e_mod = 2000000
    Steel_Plates.poissons_ratio = 0.3

    ###9. MESH
    Mesh.elementsize = 25 #[mm]

    ###10. ANALYSIS

    #-#10.1 Linear Analysis
    ##DIANA Primary output is chosen for the linear analysis.
    LFEA = Analysis("Linear analysis", "Linear") #creating linear static analysis

    #'#10.1.1 RUN LINEAR ANALYSIS
    # LFEA.runSolver = True #**Optional**

    #-#10.2 Nonlinear Analysis
    NLFEA = Analysis("Nonlinear analysis", "Nonlinear") #creating nonlinear analysis

    #'#10.2.1 Arc length control:
    NLFEA.arcLengthControl = True #**Default and strongly recommended as true**

    #'#10.2.2 Load incrementation:
    incrementation = Incrementation() #Creating an incrementation object
    NLFEA.setIncrementalMethod(incrementation) #adding incremental method to the analysis

    ##Option 1: Energy based adaptive loading
    incrementation.method = "ENERGY"
    incrementation.initial_step_size = 5    #initial size for the first step
    incrementation.max_step_size = 10       #upper limit of the step size
    incrementation.min_step_size = 3        #lower limit of the step size
    incrementation.nrOfSteps = 100          #maximum number of steps

    #'#10.2.3 Iteration method:
    iteration = Iteration() #creating iteration object
    NLFEA.setIterationMethod(iteration) #adding iterative procedure to the analysis

    ## Option 1: Regular Newton-Raphson
    iteration.method = "NEWTON-RAPHSON"
    iteration.typeOfMethod = "REGULA"
    iteration.nrOfIterations = 100

    #'#10.2.4 Line search:
    NLFEA.lineSearch = True #**Default and recommended as true**

    #'#10.2.5 Convergence criteria:
    convergence = Convergence()         #creating convergence object
    NLFEA.setConvergence(convergence)   #adding convergence criteria to the analysis

    convergence.useForceNorm = True     #**Default and recommended as true**
    convergence.useEnergyNorm = True    #**Default and recommended as true**
    convergence.useDispNorm = False     #**Default and recommended as false**


    ## Option 1: Suggested tolerances
    ##Force norm:
    convergence.forceNorm = 0.01
    convergence.forceNorm_deadLoad = 0.05
    ##Energy norm:
    convergence.energyNorm = 0.001
    convergence.energyNorm_deadLoad = 0.01

    #'#10.2.6 Additional choices for analysis
    NLFEA.allConvergenceNormsHaveToBeSatisfied = False  #**Default and recommended as false**
    NLFEA.continueIfNoConvergence = True                #**Default and recommended as true**

    #'#10.2.7 RUN NONLINEAR ANALYSIS
    NLFEA.runSolver = True #**optional**

```

```
    ###11. OUTPUTS FROM NONLINEAR ANALYSIS

    #-#11.1 Analysis output:
    NLFEA.output = {
        "DISPLA TOTAL TRANSL GLOBAL" : True,    #**Optional**
        "FORCE REACTI TRANSL GLOBAL" : True,    #**Optional**
        "STRAIN TOTAL GREEN GLOBAL" : True,     #**Optional**
        "STRAIN TOTAL GREEN PRINCI" : True,     #**Optional**
        "STRAIN CRKWDT GREEN GLOBAL" : True,    #**Optional**
        "STRAIN CRACK GREEN" : True,            #**Optional**
        "STRAIN CRKWDT GREEN PRINCI" : True,    #**Optional**
        "STRESS TOTAL CAUCHY GLOBAL" : True,    #**Optional**
        "STRESS TOTAL CAUCHY PRINCI" : True     #**Optional**
    }

    #-#11.2 Load-displacement graph (displacement in y-dir) **Optional**
    #Specify the coordinates of the point where the displacement will be retrived.
    LoadDispY_Graph.x_coord = beam.geometry.length/2 #[mm]
    LoadDispY_Graph.y_coord = 0 #[mm]

    ##The following options can be changed:
    Scalefactor.loadFactor_plot = 2#**optional**
    LoadDispY_Graph.ylabel = "Load [kN]"#**optional**

    #-#11.3 Load-displacement CSV (displacement in y-dir)**Optional**
    ##Specify the coordinates of the point for which the CSV-file will be generated:
    LoadDispY_CSV.x_coord = beam.geometry.length/2 #[mm]
    LoadDispY_CSV.y_coord = 0 #[mm]

    #The following options can be changed:
    Scalefactor.loadFactor_CSV =  2 #**optional*

    #-----------------------------------------------------------------------------------------------------------
    globals().update(locals()) #Allows main script to access all variables in function as globals
    return
```

# H Parametric study - Script B

```
def parametricStudy(valuesList):
    for val in valuesList:
        wd = sys.path[0] #Working directory of main.py
        exec(open(wd+"\\C_classesAndFunctions.py").read(), globals())
        LoadDispY_Graph.parametric = val
        LoadDispY_CSV.parametric = val
        exec(open(wd+"\\A_UserInput.py").read(), globals())
        functionOfUserInput(val)
        parametric_Study = True
        if not hasattr(Project, 'name'):
            Project.name = Project.modelName + "_" + date + "_" + Project.modelExtraInfo + "_" + str(val)
        else:
            Project.name = Project.name + "_" + str(val)
        exec(open(wd+"\\D_main.py").read())
    return locals()

##INPUT LIST OF PARAMETRIC VALUES HERE:
parametricStudy([3000,4000,5000,6000])
```

# I Parametric study - Analytical analysis

The critical value of resistance moment is evaluated with sectional analysis by assuming:

- plane sections remain plane,
- the strain in bonded reinforcement is the same as that in the surrounding concrete,
- the tensile strength of the concrete is ignored,
- the stresses in the concrete in compression are derived a parabola-rectangle relation,
- the stresses in the reinforcing steel are derived from the design curve in Figure 3.9,
- since the material properties have no uncertainties, the partial safety factors are set to $\gamma = 1$,
- the factor $\lambda = 0.8$ and $\eta = 1$.

The design bending moment resistance is calculated below. For further explanations of the following equations, please refer to [29].

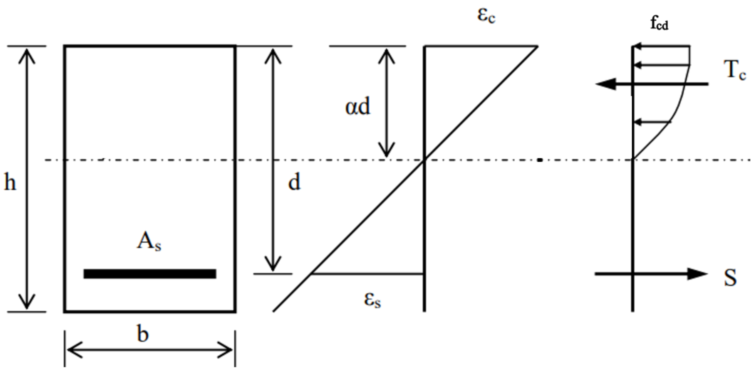



**Figure 1:** Stress block for determination of the design moment resistance [29]

**Design strength:**

$$f_{cd} = \alpha_{cc} \frac{f_{ck}}{\gamma_c} = 0.85 \cdot \frac{30}{1} = 25.5 \text{ MPa} \tag{1}$$

$$f_{yd} = \frac{f_{yk}}{\gamma_s} = \frac{500}{1} = 500 \text{ MPa} \tag{2}$$

**Yielding strain** of longitudinal reinforcement:

$$\varepsilon_{yd} = \frac{f_{yd}}{E_s} = \frac{500}{200000} = 0.0025 \tag{3}$$

Determining **Balanced section** [29]:

$$\alpha_b = \frac{\varepsilon_{cu}}{\varepsilon_{cu} + \varepsilon_{yd}} = \frac{0.0035}{0.0035 + 0.0025} = 0.583 \tag{4}$$

$$A_{s,b} = \lambda \cdot \eta \cdot \frac{f_{cd}}{f_{yd}} \cdot b \cdot d \cdot \alpha_b = 0.8 \cdot \frac{25.5}{500} \cdot 250 \cdot 365 \cdot 0.583 = 2172 \text{ mm}^2 \tag{5}$$

$$A_s = 3 \cdot 314 = 942 \text{ mm}^2 < A_{s,b} \tag{6}$$

The section is **underreinforced**, which gives:

$$\alpha = \frac{f_{yd} A_s}{\lambda \eta f_{cd} b d} = \frac{500 \cdot 942}{0,8 \cdot 25.5 \cdot 250 \cdot 365} = 0.253 \tag{7}$$

Control of reinforcement strain:

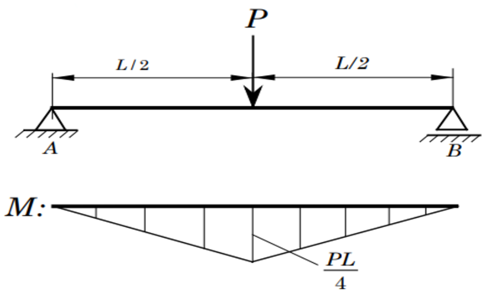
$$\varepsilon_s = \frac{1-\alpha}{\alpha} \cdot \varepsilon_{cu} = \frac{1-0.253}{0.253} \cdot 0.0035 = 0.0103 \rightarrow \text{steel yields} \tag{8}$$

$$\varepsilon_s < \varepsilon_{ud}$$

Design value of moment resistance [29]:

$$\mathrm{M}_{Rd} = \lambda\eta\alpha(1-0.5\lambda\alpha)f_{cd}bd^2 = 0.8 \cdot 0.253 \cdot (1-0.4 \cdot 0.253) \cdot 25.5 \cdot 250 \cdot 365^2 = \underline{\underline{155\text{kNm}}} \tag{9}$$

**Critical load** (dead weight is ignored) [21]:




$$\mathrm{P} = \frac{4M}{L} \tag{10}$$

The critical values of the point load for the different beam lengths are given in Table 1, and will be compared to the results of the NLFEA in Section 13.4.

**Table 1:** Critical loads

| L [m] | P [kN] |
|---|---|
| 3 | 206 |
| 4 | 129 |
| 5 | 103 |
| 6 | 86 |

Katja Hansen

Python/DIANA framework for robust NLFEA of RC beams

NTNU

Norwegian University of
Science and Technology