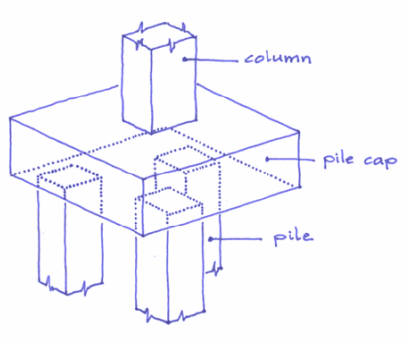
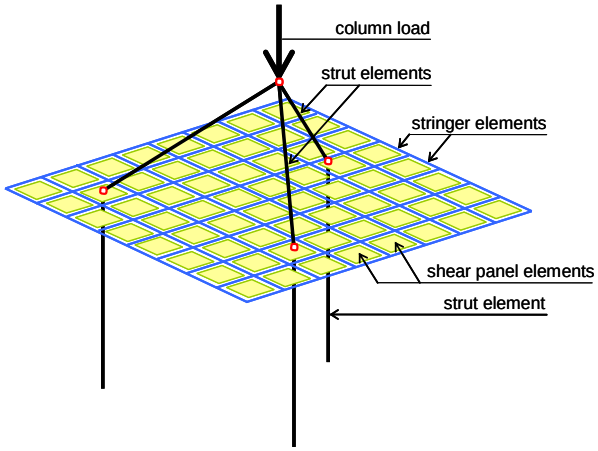
Structural design of reinforced concrete pile caps

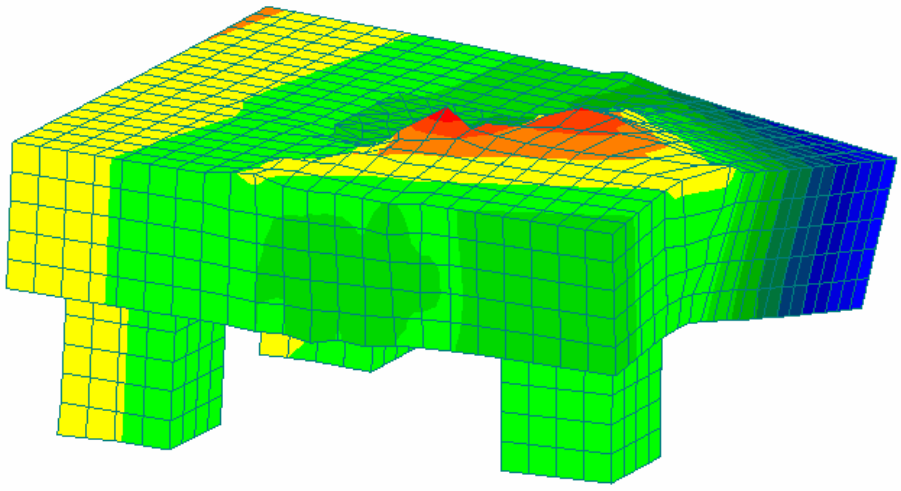
The strut-and-tie method extended with the stringer-panel method

December 2006

A.V. van de Graaf







TU Delft
Delft University of Technology

Faculty of Civil Engineering and Geosciences
Section Structural Mechanics

Structural design of reinforced concrete pile caps

The strut-and-tie method extended with the stringer-panel method

A.V. van de Graaf

Delft, December 2006

Delft University of Technology
Faculty of Civil Engineering and Geosciences
Section Structural Mechanics

Delft University of Technology

Personalia

STUDENT
Anne Vincent van de Graaf
1040626
annevandegraaf@yahoo.com
+ 31 (0)6 12 29 61 32

GRADUATION COMMITTEE
prof.dr.ir. J.G. Rots (supervisor graduation committee)
Delft University of Technology
Faculty of Civil Engineering and Geosciences – Section Structural Mechanics
j.g.rots@bk.tudelft.nl
+ 31 (0)15 278 44 90

dr.ir. P.C.J. Hoogenboom (daily supervisor)
Delft University of Technology
Faculty of Civil Engineering and Geosciences – Section Structural Mechanics
p.hoogenboom@citg.tudelft.nl
+ 31 (0)15 278 80 81

ir. W.J.M. Peperkamp
Delft University of Technology
Faculty of Civil Engineering and Geosciences – Section Concrete Structures
w.peperkamp@citg.tudelft.nl
+ 31 (0)15 278 45 76

ir. J.W. Welleman
Delft University of Technology
Faculty of Civil Engineering and Geosciences – Section Structural Mechanics
j.w.welleman@citg.tudelft.nl
+ 31 (0)15 278 48 56

ir. L.J.M. Houben (graduation coordinator)
Delft University of Technology
Faculty of Civil Engineering and Geosciences – Section Road & Railway Engineering
l.j.m.houben@tudelft.nl
+ 31 (0)15 278 49 17

Preface

This graduation report has been written within the framework of a Master of Science Project originally entitled ***WWW Design of Reinforced Concrete Pile Caps***. This project was put forward by the Structural Mechanics Section of the Faculty of Civil Engineering and Geosciences at Delft University of Technology.

Although I spent a lot of time in mastering the Java programming language and implementing the design model in an applet using Java SE Development Kit (JDK) [14], not much of this work can be found directly in this report. The same applies to the initial work that I have done in TurboPascal using Borland Delphi [13]. Therefore, this graduation report is rather brief. For those readers, who are interested in using the applet, please refer to the following web address: http://www.mechanics.citg.tudelft.nl/pca.

Hereby I would like to thank ir. H.J.A.M. Geers (Faculty of Electrical Engineering, Mathematics and Computer Science at Delft University of Technology) for his advice during the design and implementation of the applet. Many thanks also to ir. J.A. den Uijl for his contribution with Atena 3D. And last but not least, I would like to thank dr.ir. P.C.J. Hoogenboom for his support and suggestions during this project.

Delft, December 12, 2006

Anne van de Graaf

Table of contents

Summary

Many foundations in The Netherlands, mainly those in coastal areas, are on piles. These piles are often over 15 m long at distances of 1 to 4 m. If possible, these piles are driven into the soil at the positions of walls and columns of a building. The presence of piles of a previous building may hamper a free choice of the new pile positions. Removing the old piles is not a solution, because this leaves holes in deep clay layers through which saline groundwater may penetrate into the upper soil. Moreover, the old piles cannot be reused because their quality cannot be guaranteed. As a consequence, pile caps often have to cover piles that are positioned in an irregular pattern.

The objective of this Master of Science Project was to develop a design model for calculating the pile loading and reinforcement stresses for pile caps on irregularly positioned foundation piles. This model has been based on the strut-and-tie method, however, the ties have been replaced by another model consisting of stringer elements and shear panel elements. This model predicts vertical pile reactions, reinforcement stresses and shear stresses in concrete. For practical application, it has been implemented in a computer program called Pile Cap Applet (PCA). This applet was designed to be user-friendly, to require only a moderate amount of data and to execute fast.

PCA has been tested and validated in two ways. Firstly, it has been shown that the design model meets all equilibrium requirements. This has been tested for two pile caps. Both cases revealed that the design model complies with horizontal and vertical force equilibrium and moment equilibrium. From the theory of plasticity it then follows that this model gives a safe approximation of the ultimate load. Secondly, the ultimate load predicted by PCA has been compared to the ultimate load predicted by a non-linear finite element analysis. This comparison yielded several interesting conclusions whereof the most important ones are included in this summary.
The ultimate load predicted by PCA is very conservative. Clearly, the real structure can carry the load in more ways than an equilibrium system (PCA) assumes. Furthermore, for the considered pile cap the design model predicted another failure mechanism than the finite element analysis. PCA predicted that the considered pile cap 'collapsed' because of reaching the yield strength in one of the reinforcing bars. In the finite element analysis, the pile cap collapsed because of a shear failure. This failure mechanism cannot be predicted by PCA. For the considered pile cap the vertical pile reactions predicted by PCA are approximately equal to those predicted by the non-linear finite element analysis. However, the reinforcement stresses at serviceability load according to PCA are much higher than those determined by the finite element analysis. This implies that the stresses calculated by PCA are not useful for checking the maximum crack width.

List of symbols

Latin symbols

a	length of a shear panel element [mm]
b	width of a shear panel element [mm]
c	concrete cover [mm]
d_x	center-to-center distance of reinforcing bars in x -direction [mm]
d_y	center-to-center distance of reinforcing bars in y -direction [mm]
$E_{cap\ concrete}$	Young's modulus of the cap concrete [N/mm^2]
E_{compl}	complementary energy [Nmm]
$E_{pile\ concrete}$	Young's modulus of the pile concrete [N/mm^2]
E_{rebar}	Young's modulus of the reinforcement [N/mm^2]
EA	extensional stiffness [N]
F	external force [N]
$G_{cap\ concrete}$	shear modulus of cap concrete [N/mm^2]
h	depth of the pile cap [mm]
N	normal force [N]
S	shear force [N]
t	effective depth of the pile cap with regard to shear stresses [mm]
u_i	displacement in direction i [mm]

Greek symbols

$\gamma_{\underline{xy}}$	shear angle [rad]
ν	Poisson's ratio [-]
σ	normal stress [N/mm^2]
τ	shear stress [N/mm^2]
ϕ_x	reinforcing bar diameter in x -direction [mm]
ϕ_y	reinforcing bar diameter in y -direction [mm]

Remaining symbol

ℓ	length of a stringer element or strut element [mm]

1 Introduction

It is well-known that many buildings in The Netherlands, mainly those in coastal areas, are founded on piles. These piles can easily reach a length of over 15 m and are usually spaced at distances of 1 to 4 m. If possible, these piles are driven into the soil at the positions of walls and columns. Unfortunately, a structural designer is not always free in this choice, because piles of a previous building may be present. Removing these old piles is not a solution, since this leaves holes in deep clay layers through which saline groundwater may penetrate into the upper soil. Reusing the old piles is not an option either, because their quality cannot be guaranteed. These restrictions often result in irregular pile patterns, which makes calculation of pile caps by hand difficult if not impossible.

The objective of this Master of Science Project is to develop a design model for calculating the pile loading and reinforcement stresses for pile caps on irregularly positioned foundation piles. This design method is based on the strut-and-tie method extended with the stringer-panel method. The model is implemented in an applet and can be used for structural design.

The composition of this report is as follows. Chapter 2 gives a problem definition, discusses the model constitution and outlines the research. Chapter 3 considers the mathematical description of stringer elements, shear panel elements and strut elements. These are used as building blocks for the design model. In Chapter 4 it is explained how to assemble the system starting from the mathematical element descriptions given in the previous chapter. Furthermore, this chapter includes processing the boundary conditions and solving the obtained system of linear equations. Chapter 5 discusses the design of the applet and three important procedures, namely the preprocessor, the kernel and the postprocessor. In Chapter 6 the Java implementation is tested by checking equilibrium requirements in two specific cases. Chapter 7 compares the ultimate load predicted by the applet with a non-linear finite element analysis. Finally, Chapter 8 presents the conclusions and recommendations.

2 Design problem of the reinforced concrete pile cap

This chapter defines the design problem that was introduced in Chapter 1. Section 2.1 gives a description of the problem to be solved. Section 2.2 explains which elements are used and how these elements constitute the pile cap model. Section 2.3 gives an outline of the research area including aspects that are not taken into account. In the next chapter, Chapter 3, the elements which constitute the model presented in this chapter are mathematically described.

2.1 Problem description

The problem to be solved is to develop a design model for determining the pile loading and the reinforcement stresses for pile caps on irregularly positioned foundation piles in buildings (Figure 1). One way of calculating pile caps is to create a model in a 3D finite element package. An important disadvantage of this approach is that it is time-consuming. Creating the computer model as well as performing an advanced calculation requires a lot of time. Another method for solving this problem is to use rough models, which may be calculated by hand. But since these rough models introduce a lot of uncertainty, a large safety factor is required. Clearly, structural designers need a reliable and rational calculation method, which can be carried out easily.

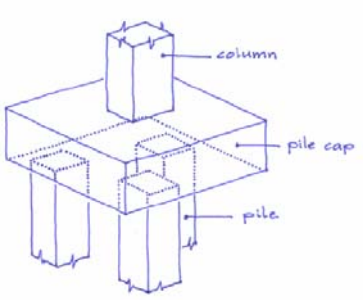


Figure 1 Example pile cap

2.2 Modeling the pile cap

For stocky structures loaded by concentrated forces, the strut-and-tie method is commonly adopted [10]. This method uses solely compression members (struts) and tension members (ties). In Figure 2 a strut-and-tie model has been drawn for the example pile cap given in Figure 1. Compression members have been drawn in green and tension members have been drawn in red. If reinforcing bars are put in the directions of the ties the result would be very impractical to make. Moreover, if a pile cap consists of more piles and columns the reinforcement patterns would be even more complicated and therefore labor-intensive and prone to error. Orthogonal reinforcement patterns with fixed center-to-center distances are far more practical. But then, the above mentioned strut-and-tie method is not convenient anymore. Therefore, the ties are replaced by another model (Figure 3), consisting of stringer elements and shear panel elements ([1], [2]). In this renewed model, the stringer elements represent the reinforcing bars, while the shear panel elements

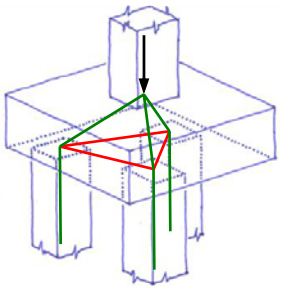


Figure 2 Strut-and-tie model for the example pile cap of Figure 1

represent the concrete in between. From Figure 3 it can be seen that the load is carried by strut elements that are hold in place by a combination of stringer elements and shear panel elements.

2.3 Research outline

Some restrictions need to be introduced to arrive at a practical design model.

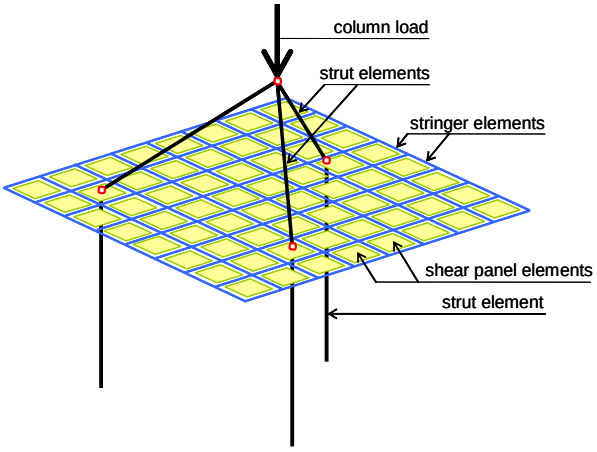


Figure 3 Strut-and-tie model extended with a stringer-panel model

The first restriction is that columns can only transfer normal (vertical) loads. A column load is represented by a concentrated force, which is applied at the center of gravity of the column (Figure 3). Therefore, moments in the columns cannot be included. Horizontal loads and bending moments are excluded from this research. Since the piles are modeled as strut elements, they can only transfer normal loads. Furthermore, it is assumed that the tip of the pile is restrained in all directions. The behavior of the soil in which the piles are embedded is not taken into consideration, which also means that no pile-soil interaction is taken into account. For the axial stiffness of the stringer elements, only the extensional stiffness of the reinforcing bars is taken into account. This means it is assumed that the concrete does not contribute to the transfer of tensile forces and that effects like tension-stiffening are not taken into consideration. Only main reinforcement is considered, which means that shear reinforcement and other kinds of reinforcement are excluded from the model. In both directions, only one layer of reinforcing bars is taken into account. Another restriction is that the dead weight of the pile cap is not taken into consideration. This is acceptable since the dead weight of the pile cap is only a fraction of the load that is carries.

The implementation of the design model in an applet also poses a few restrictions. To ensure an orderly Graphical User Interface (GUI) it is decided to limit the maximum number of columns to four and the maximum number of piles to six. The minimum number of piles is set to three to ensure a kinematical determinate system. The center-to-center distances of the reinforcing bars are equal per direction. Only one reinforcing bar diameter can be specified per direction.

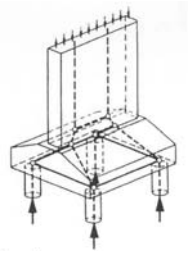


Figure 4 Pier on a pile cap

The design problem discussed in this graduation report is mainly aimed at pile caps used in buildings. But the general nature of the design model to be discussed makes its application also suitable for use in for example piers on pile caps (Figure 4).

3 Mathematical description of the used elements

In Chapter 2 it was explained that the model which represents the pile cap consists of three different elements, namely stringer elements, shear panel elements and strut elements. This chapter describes the structural behavior of these elements in a mathematical way. First, Section 3.1 gives the general agreements concerning local and global co-ordinate systems and notations. Then, in Section 3.2, the stiffness relation for a stringer element is derived, based on the graduation work of Hoogenboom (1993) [6]. In Section 3.3 the stiffness relation for a shear panel element is derived using the work of Blaauwendraad (2004) [4]. Finally, in Section 3.4 a description of the strut element is given, which has been based on the work of Nijenhuis (1973) [8] and Hartsuijker (2000) [5]. In the next chapter, Chapter 4, these descriptions are used to formulate the structural behavior of the pile cap.

3.1 Co-ordinate systems and notations

The global co-ordinate system xyz for the pile cap is indicated in Figure 5. In the next sections, local co-ordinate systems $\overline{xyz}$ are defined. In the case of stringer elements and shear panel elements, the orientation of the local co-ordinate axes is in the same direction as the global co-ordinate system (Figure 5). This implies that for these elements a rotation matrix is not needed. Because strut elements have a three dimensional orientation (Figure 3) and their local co-ordinate system is chosen according to the orientation of the element, a rotation matrix is necessary. Therefore, Section 3.4 is divided in two subsections. Subsection 3.4.1 gives the mathematical description of the strut element. In subsection 3.4.2 the rotation matrix is derived. In the next sections, the following (common) convention is used: scalars are not underlined, vectors are underlined and matrices are doubly underlined. The derivations in this chapter are valid for single elements only. To be formally correct a superscript (e) should be used, but for the sake of convenience this superscript is left out.

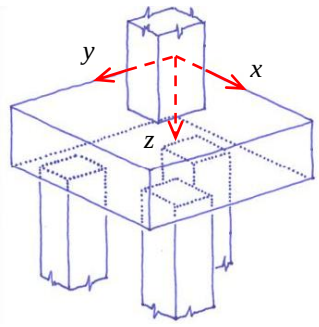


Figure 5 Global co-ordinate system

3.2 Stringer element

The stringer element consists of a bar with length ℓ and extensional stiffness EA and possesses three degrees of freedom (DOF): $u_{\bar{x}1}$, $u_{\bar{x}2}$ and $u_{\bar{x}3}$ (Figure 6). The DOF at the ends of the element are called $u_{\bar{x}1}$ and $u_{\bar{x}3}$ respectively. The intermediate DOF is named $u_{\bar{x}2}$. The element is loaded by two concentrated forces at the ends of the bar, which are called $F_{\bar{x}1}$ and $F_{\bar{x}3}$, and an evenly distributed shear force τt along the bar axis. This distributed shear force is a result of interaction with adjacent shear panel elements, which

are described in Section 3.3. The sum of the distributed shear force over the length ℓ is equal to $F_{\bar{x}2}$.

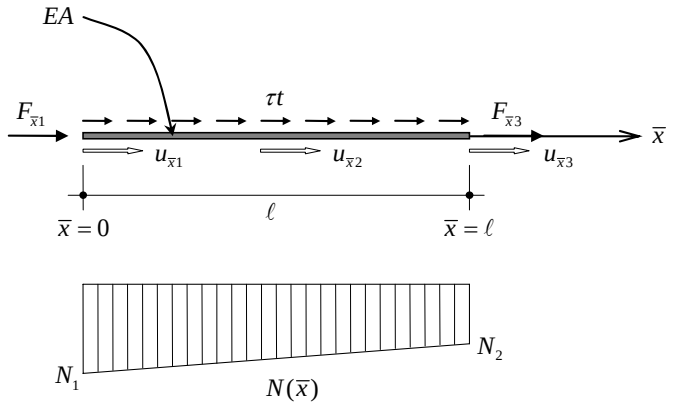


Figure 6 Stringer element in a local co-ordinate system $\overline{xyz}$ [figure taken from Hoogenboom [6]]

The normal force $N(\bar{x})$ in the bar can be described by

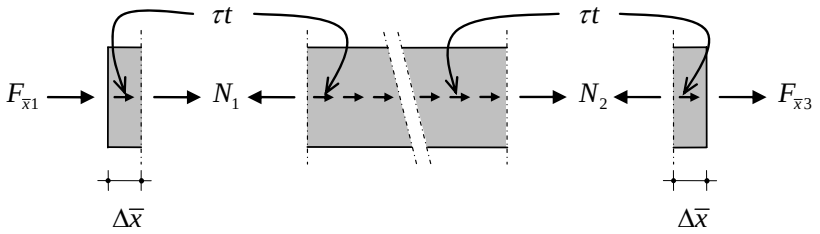
$$N(\bar{x}) = N_1 - \frac{\bar{x}}{\ell}(N_1 - N_2). \qquad (1)$$

From equilibrium of the bar ends (Figure 7) it may be concluded that

$$F_{\bar{x}1} = -N_1 \text{ and } F_{\bar{x}3} = N_2. \qquad (2)$$

$F_{\bar{x}2}$ can be expressed as

$$F_{\bar{x}2} = N_1 - N_2 = \tau t \ell \Leftrightarrow \tau t = \frac{N_1 - N_2}{\ell}. \qquad (3)$$



$$\lim_{\Delta\bar{x}\to 0}\left(F_{\bar{x}1} + N_1 + \tau t \Delta\bar{x}\right) = F_{\bar{x}1} + N_1 = 0 \qquad \lim_{\Delta\bar{x}\to 0}\left(-N_2 + F_{\bar{x}3} + \tau t \Delta\bar{x}\right) = -N_2 + F_{\bar{x}3} = 0$$

Figure 7 Equilibrium consideration of the end parts of the stringer element

The stiffness relation for the stringer element is derived using complementary energy. The expression for the complementary energy of the bar reads [6]

$$E_{\text{compl}} = \int_{\bar{x}=0}^{\ell} \tfrac{1}{2}\frac{N^2}{EA} d\bar{x} - F_{\bar{x}1}u_{\bar{x}1} - F_{\bar{x}3}u_{\bar{x}3} - \int_{\bar{x}=0}^{\ell} \tau t u_{\bar{x}}(\bar{x}) d\bar{x}. \qquad (4)$$

Substitution of equations (1), (2) and (3) in the expression for the complementary energy (4) gives

$$E_{\text{compl}} = \int_{\bar{x}=0}^{\ell} \frac{1}{2EA}\left(N_1 - \bar{x}\frac{N_1 - N_2}{\ell} \right)^2 d\bar{x} + N_1 u_{\bar{x}1} - N_2 u_{\bar{x}3} - \int_{\bar{x}=0}^{\ell} \frac{N_1 - N_2}{\ell} u_{\bar{x}}(\bar{x})\, d\bar{x} \,. \qquad (5)$$

The intermediate DOF $u_{\bar{x}2}$ is now defined as [6]

$$u_{\bar{x}2} = \frac{2}{\ell} \int_{\bar{x}=0}^{\ell} u_{\bar{x}}(\bar{x}) d\bar{x} \,, \qquad (6)$$

which may be interpreted as the mean displacement of the stringer element. Further elaboration of expression (5) using equation (6) leads to

$$\begin{aligned} E_{\text{compl}} &= \int_{\bar{x}=0}^{\ell} \frac{1}{2EA}\left(N_1^2 - 2\bar{x}\frac{N_1^2 - N_1 N_2}{\ell} + \bar{x}^2 \left(\frac{N_1 - N_2}{\ell} \right)^2 \right) d\bar{x} + N_1 u_{\bar{x}1} - N_2 u_{\bar{x}3} - \left(N_1 - N_2 \right) u_{\bar{x}2} \\ &= \frac{1}{2EA}\left[N_1^2 \bar{x} - \frac{N_1^2 - N_1 N_2}{\ell} \bar{x}^2 + \tfrac{1}{3} \bar{x}^3 \left(\frac{N_1 - N_2}{\ell} \right)^2 \right]_{\bar{x}=0}^{\ell} + N_1 u_{\bar{x}1} - N_2 u_{\bar{x}3} - N_1 u_{\bar{x}2} + N_2 u_{\bar{x}2} \\ &= \frac{1}{2EA}\left[N_1^2 \ell - (N_1^2 - N_1 N_2)\ell + \tfrac{1}{3}\ell (N_1^2 - 2N_1 N_2 + N_2^2) \right] + N_1 u_{\bar{x}1} - N_2 u_{\bar{x}3} - N_1 u_{\bar{x}2} + N_2 u_{\bar{x}2} \\ &= \frac{1}{2EA}\left[\tfrac{1}{3} N_1 N_2 \ell + \tfrac{1}{3} N_1^2 \ell + \tfrac{1}{3} N_2^2 \ell \right] + N_1 u_{\bar{x}1} - N_2 u_{\bar{x}3} - N_1 u_{\bar{x}2} + N_2 u_{\bar{x}2} \\ &= \frac{\ell}{6EA}\left[N_1^2 + N_1 N_2 + N_2^2 \right] + N_1 u_{\bar{x}1} - N_2 u_{\bar{x}3} - N_1 u_{\bar{x}2} + N_2 u_{\bar{x}2} . \end{aligned}$$

The complementary energy should be stationary in relation to variations of the stresses, meaning that the derivatives with respect to N_1 and N_2 need to be equal to zero [6]

$$\frac{\partial E_{\text{compl}}}{\partial N_1} = \frac{\ell}{6EA}\left(2N_1 + N_2 \right) + u_{\bar{x}1} - u_{\bar{x}2} = 0,$$
$$\frac{\partial E_{\text{compl}}}{\partial N_2} = \frac{\ell}{6EA}\left(N_1 + 2N_2 \right) - u_{\bar{x}3} + u_{\bar{x}2} = 0.$$

In matrix notation these equations read

$$\frac{\ell}{6EA} \cdot \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} N_1 \\ N_2 \end{bmatrix} = \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{\bar{x}1} \\ u_{\bar{x}2} \\ u_{\bar{x}3} \end{bmatrix}, \qquad (7)$$

where the dot implies matrix multiplication.
Pre-multiplication of equation (7) by the inverse of the left hand side matrix of equation (7), gives

$$\frac{EA}{\ell} \cdot \begin{bmatrix} 4 & -2 \\ -2 & 4 \end{bmatrix} \cdot \frac{\ell}{6EA} \cdot \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} N_1 \\ N_2 \end{bmatrix} = \frac{EA}{\ell} \cdot \begin{bmatrix} 4 & -2 \\ -2 & 4 \end{bmatrix} \cdot \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{\bar{x}1} \\ u_{\bar{x}2} \\ u_{\bar{x}3} \end{bmatrix} \Rightarrow$$

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} N_1 \\ N_2 \end{bmatrix} = \begin{bmatrix} N_1 \\ N_2 \end{bmatrix} = \frac{EA}{\ell} \cdot \begin{bmatrix} -4 & 6 & -2 \\ 2 & -6 & 4 \end{bmatrix} \cdot \begin{bmatrix} u_{\bar{x}1} \\ u_{\bar{x}2} \\ u_{\bar{x}3} \end{bmatrix}. \quad (8)$$

From equations (2) and (3) it follows that the relation between the internal forces N_1 and N_2 and external loads $F_{\bar{x}1}$, $F_{\bar{x}2}$ and $F_{\bar{x}3}$ can be described by

$$\begin{bmatrix} F_{\bar{x}1} \\ F_{\bar{x}2} \\ F_{\bar{x}3} \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} N_1 \\ N_2 \end{bmatrix}. \quad (9)$$

The final step in the derivation of the stiffness relation for a stringer element, is to substitute equation (8) into equation (9), which leads to

$$\begin{bmatrix} F_{\bar{x}1} \\ F_{\bar{x}2} \\ F_{\bar{x}3} \end{bmatrix} = \begin{bmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \cdot \frac{EA}{\ell} \cdot \begin{bmatrix} -4 & 6 & -2 \\ 2 & -6 & 4 \end{bmatrix} \cdot \begin{bmatrix} u_{\bar{x}1} \\ u_{\bar{x}2} \\ u_{\bar{x}3} \end{bmatrix} = \frac{EA}{\ell} \cdot \begin{bmatrix} 4 & -6 & 2 \\ -6 & 12 & -6 \\ 2 & -6 & 4 \end{bmatrix} \cdot \begin{bmatrix} u_{\bar{x}1} \\ u_{\bar{x}2} \\ u_{\bar{x}3} \end{bmatrix}.$$

As explained in Section 3.1, a rotation matrix is not needed. So the above relation also holds for the global co-ordinate system and reads

$$\begin{bmatrix} F_{x1} \\ F_{x2} \\ F_{x3} \end{bmatrix} = \frac{EA}{\ell} \cdot \begin{bmatrix} 4 & -6 & 2 \\ -6 & 12 & -6 \\ 2 & -6 & 4 \end{bmatrix} \cdot \begin{bmatrix} u_{x1} \\ u_{x2} \\ u_{x3} \end{bmatrix}. \quad (10)$$

As stated in Section 2.3, for the axial stiffness of the stringer elements, only the extensional stiffness of the reinforcing bars is taken into account. Therefore, the extensional stiffness EA in equation (10) can be calculated from the Young's modulus of the reinforcement E_{rebar} and the cross-sectional area of a reinforcing bar. The length ℓ in equation (10) is equal to the length or width of the adjacent shear panel element. Generating the stringer elements in the applet and the global numbering of the stringer element DOF is explained in Appendix A1. Once the displacements $u_{\bar{x}1}$, $u_{\bar{x}2}$ and $u_{\bar{x}3}$ are known, the normal forces N_1 and N_2 acting at the ends of the stringer element can be calculated by using equation (8).

3.3 Shear panel element

A shear panel element is a rectangular element that is meant for transmitting an evenly distributed shear force τt (Figure 8). At its edges this shear stress interacts with adjacent stringer elements. A shear panel element has a length a, a width b and an effective depth t. Determining this effective depth is explained at the end of this section. The shear panel element possesses a shear stiffness $G_{cap\ concrete}$, which can be calculated from the well-known expression

$$G_{cap\ concrete} = \frac{E_{cap\ concrete}}{2(1+\nu)},$$

where $E_{cap\ concrete}$ represents the Young's modulus of the cap concrete and ν represents Poisson's ratio.

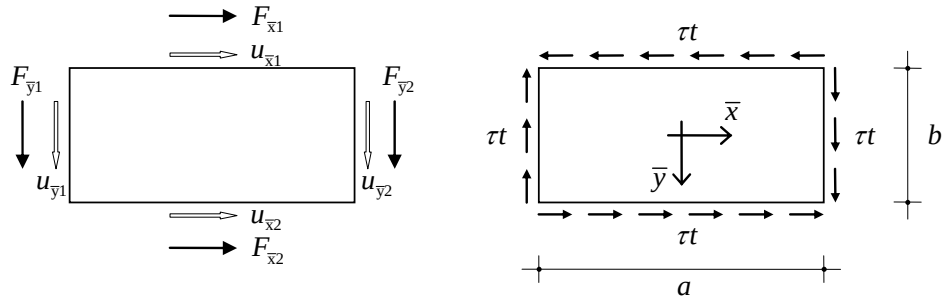


Figure 8 Shear panel element in a local co-ordinate system $\overline{xyz}$

Since the shear stress τt is constant, the shear angle $\gamma_{\bar{xy}}$ will also be constant. Moreover, the edges of the deformed shear panel element remain straight and do not elongate. Therefore, the deformation of the shear panel element can be described by four DOF: $u_{\bar{x}1}$, $u_{\bar{x}2}$, $u_{\bar{y}1}$ and $u_{\bar{y}2}$, which are chosen halfway each edge.
The resulting shear forces along the edges can be calculated as

$$\begin{aligned} &F_{\bar{x}1} = -\tau t a \text{ and } F_{\bar{x}2} = \tau t a, \\ &F_{\bar{y}1} = -\tau t b \text{ and } F_{\bar{y}2} = \tau t b. \end{aligned} \quad (11)$$

From the constitutive relation it is known that

$$\tau t = G t \gamma_{\bar{xy}}. \quad (12)$$

The shear angle $\gamma_{\bar{xy}}$ can be determined from Figure 9

$$\gamma_{\bar{xy}} = \frac{\Delta u_{\bar{x}}}{b} + \frac{\Delta u_{\bar{y}}}{a} = \frac{u_{\bar{x}2} - u_{\bar{x}1}}{b} + \frac{u_{\bar{y}2} - u_{\bar{y}1}}{a}. \quad (13)$$

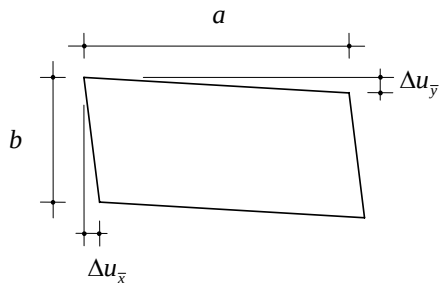


Figure 9 Deformed shear panel element

Substitution of equation (13) into equation (12) gives

$$\tau t = Gt\left(\frac{u_{\bar{x}2}}{b} - \frac{u_{\bar{x}1}}{b} + \frac{u_{\bar{y}2}}{a} - \frac{u_{\bar{y}1}}{a}\right). \qquad (14)$$

Substitution of equation (14) into equations (11) yields the following stiffness relation

$$\begin{aligned}
F_{\bar{x}1} &= -Gta\left(\frac{u_{\bar{x}2}}{b} - \frac{u_{\bar{x}1}}{b} + \frac{u_{\bar{y}2}}{a} - \frac{u_{\bar{y}1}}{a}\right) = Gt\left(-\frac{u_{\bar{x}2}a}{b} + \frac{u_{\bar{x}1}a}{b} - u_{\bar{y}2} + u_{\bar{y}1}\right),\\
F_{\bar{x}2} &= Gta\left(\frac{u_{\bar{x}2}}{b} - \frac{u_{\bar{x}1}}{b} + \frac{u_{\bar{y}2}}{a} - \frac{u_{\bar{y}1}}{a}\right) = Gt\left(\frac{u_{\bar{x}2}a}{b} - \frac{u_{\bar{x}1}a}{b} + u_{\bar{y}2} - u_{\bar{y}1}\right),\\
F_{\bar{y}1} &= -Gtb\left(\frac{u_{\bar{x}2}}{b} - \frac{u_{\bar{x}1}}{b} + \frac{u_{\bar{y}2}}{a} - \frac{u_{\bar{y}1}}{a}\right) = Gt\left(-u_{\bar{x}2} + u_{\bar{x}1} - \frac{u_{\bar{y}2}b}{a} + \frac{u_{\bar{y}1}b}{a}\right),\\
F_{\bar{y}2} &= Gtb\left(\frac{u_{\bar{x}2}}{b} - \frac{u_{\bar{x}1}}{b} + \frac{u_{\bar{y}2}}{a} - \frac{u_{\bar{y}1}}{a}\right) = Gt\left(u_{\bar{x}2} - u_{\bar{x}1} + \frac{u_{\bar{y}2}b}{a} - \frac{u_{\bar{y}1}b}{a}\right).
\end{aligned} \qquad (15)$$

For convenience, the following dimensionless parameters are defined

$$\alpha = \frac{a}{b} \text{ and } \beta = \alpha^{-1} = \frac{b}{a}. \qquad (16)$$

By using the dimensionless parameters α and β from (16) and by writing equations (15) in matrix form, the element stiffness relation is obtained

$$\begin{bmatrix} F_{\bar{x}1} \\ F_{\bar{x}2} \\ F_{\bar{y}1} \\ F_{\bar{y}2} \end{bmatrix} = Gt \cdot \begin{bmatrix} \alpha & -\alpha & 1 & -1 \\ -\alpha & \alpha & -1 & 1 \\ 1 & -1 & \beta & -\beta \\ -1 & 1 & -\beta & \beta \end{bmatrix} \cdot \begin{bmatrix} u_{\bar{x}1} \\ u_{\bar{x}2} \\ u_{\bar{y}1} \\ u_{\bar{y}2} \end{bmatrix}.$$

As explained in Section 3.1, a rotation matrix is not needed. Therefore, the above relation also holds in the global co-ordinate system

$$\begin{bmatrix} F_{x1} \\ F_{x2} \\ F_{y1} \\ F_{y2} \end{bmatrix} = Gt \cdot \begin{bmatrix} \alpha & -\alpha & 1 & -1 \\ -\alpha & \alpha & -1 & 1 \\ 1 & -1 & \beta & -\beta \\ -1 & 1 & -\beta & \beta \end{bmatrix} \cdot \begin{bmatrix} u_{x1} \\ u_{x2} \\ u_{y1} \\ u_{y2} \end{bmatrix}. \qquad (17)$$

The effective depth t from equation (17) can be determined from the concrete cover c, the center-to-center distance of the reinforcing bars in x-direction d_x, the center-to-center distance of the reinforcing bars in y-direction d_y and the reinforcing bar

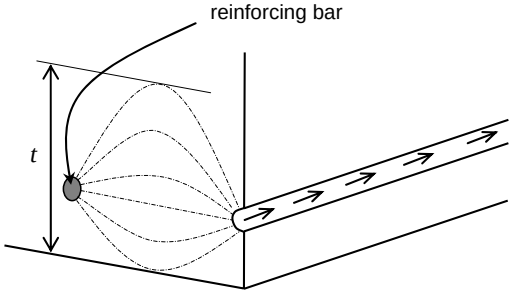


Figure 10 Shear stress trajectories between reinforcing bars

diameters ϕ_x and ϕ_y. For determining t the following scheme is used (which is based on the stress flow in Figure 10)

$$t = c + \frac{\left(\phi_x + \phi_y\right)}{4} + \frac{\left(d_x + d_y\right)}{4}.$$

But if this formula delivers a value for t which exceeds $\left(d_x + d_y\right)/2$ then the value for t is set to $\left(d_x + d_y\right)/2$. This only occurs in the rare case of a very large concrete cover. Of course, it also has to be checked that the effective depth t is not larger than the real pile cap depth h. Generating the shear panel elements in the applet and the global numbering of the shear panel element DOF is explained in Appendix A2. Once the displacements $u_{\bar{x}1}$, $u_{\bar{x}2}$, $u_{\bar{y}1}$ and $u_{\bar{y}2}$ are solved, the shear stress τ acting on the shear panel element can be determined by using equation (14).

It is noticed that in the stringer-panel method two slight incompatibilities occur. The normal force in a stringer element is assumed to be linear (Figure 6), which implies that the displacements of this element are quadratic. These displacements are not compatible with the displacements of a shear panel element. Moreover, the DOF of the shear panel element are situated halfway each side, while the intermediate DOF of the stringer element is interpreted as the mean displacement, which needs not to be necessarily halfway the stringer element. But as already mentioned, these are small incompatibilities.

One last remark is made concerning the structural behavior of the shear panel element after the first crack occurs. In this report it is assumed that cracking of the element does not influence its structural behavior, while in reality the stiffness of the element reduces. Although the shear panel element is used in the design model, an alternative can be used consisting of two diagonal elements (Figure 11). If in a diagonal element the tensile strength is exceeded, it should be left out in further analysis. This implies that the structural analysis then becomes an iterative process which ends when all diagonal elements are in compression or in tension but not cracked.

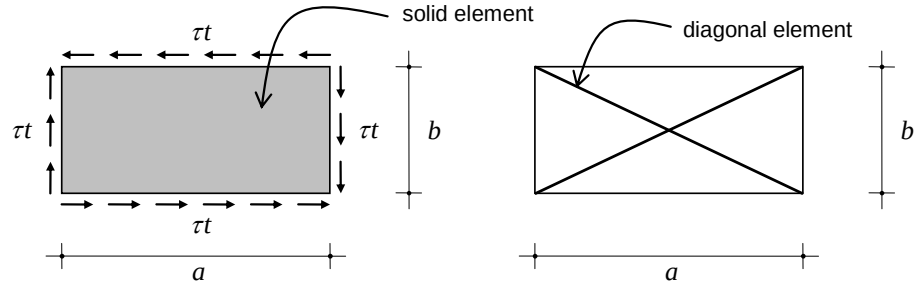


Figure 11 Shear panel element versus an element containing of two diagonal elements

3.4 Strut element

3.4.1 Element description

First, the mathematical description of the strut element is derived in a local co-ordinate system $\overline{xyz}$, in which the $\bar{x}$ -axis coincides with the centerline of the element. Later on, this description is transformed to the global co-ordinate system xyz.
A strut element consists of two nodes, called node 1 and node 2, which are connected through a straight bar. The length of a strut element is denoted by ℓ and its extensional stiffness is denoted by EA. Two concentrated forces, $F_{\bar{x}1}$ and $F_{\bar{x}2}$, are acting on nodes 1 and 2 respectively (Figure 12). The relation between forces and displacements is well-known

$$\underline{\bar{F}} = \underline{\underline{\bar{K}}} \cdot \underline{\bar{u}}\,, \tag{18}$$

where $\underline{\bar{F}}$ is the force vector, $\underline{\underline{\bar{K}}}$ is the element stiffness matrix and $\underline{\bar{u}}$ is the displacement vector.

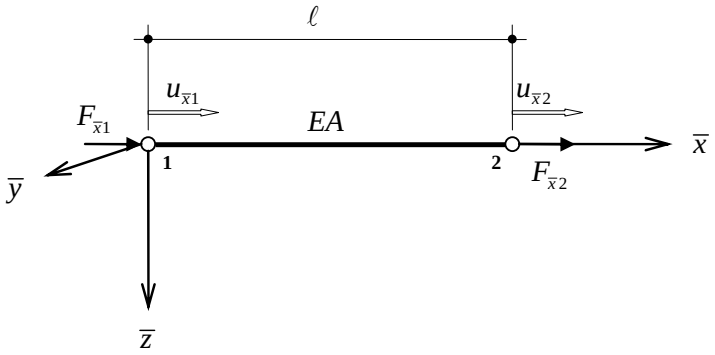


Figure 12 Strut element in a local co-ordinate system $\overline{xyz}$

This stiffness relation has been elaborated in many textbooks (Hartsuijker (2000) [5])

$$\begin{bmatrix} F_{\bar{x}1} \\ F_{\bar{x}2} \end{bmatrix} = \frac{EA}{\ell} \cdot \begin{bmatrix} +1 & -1 \\ -1 & +1 \end{bmatrix} \cdot \begin{bmatrix} u_{\bar{x}1} \\ u_{\bar{x}2} \end{bmatrix}, \tag{19}$$

where EA is the strut element axial stiffness and ℓ is the strut element length.

Since each node possesses three DOF, relation (19) needs to be expanded to include the $\bar{y}$ -direction and the $\bar{z}$ -direction

$$\begin{bmatrix} F_{\bar{x}1} \\ F_{\bar{y}1} \\ F_{\bar{z}1} \\ F_{\bar{x}2} \\ F_{\bar{y}2} \\ F_{\bar{z}2} \end{bmatrix} = \frac{EA}{\ell} \cdot \begin{bmatrix} +1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & +1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} u_{\bar{x}1} \\ u_{\bar{y}1} \\ u_{\bar{z}1} \\ u_{\bar{x}2} \\ u_{\bar{y}2} \\ u_{\bar{z}2} \end{bmatrix}.$$

The foregoing formulation has been derived, as already mentioned, in a local co-ordinate system. To transform this formulation to the global co-ordinate system xyz, the displacements as well as the forces have to be rotated.

3.4.2 Element rotation

First, the displacements are considered. The displacements $\underline{\bar{u}}_i$ of node i $(i=1,2)$ in the local co-ordinate system $\overline{xyz}$ is expressed in terms of the displacements $\underline{u}_i$ of node i $(i=1,2)$ in the global co-ordinate system xyz. Figure 13 shows the strut element, including the positive definitions of the displacements in the local co-ordinate system $\overline{xyz}$ and the global co-ordinate system xyz.

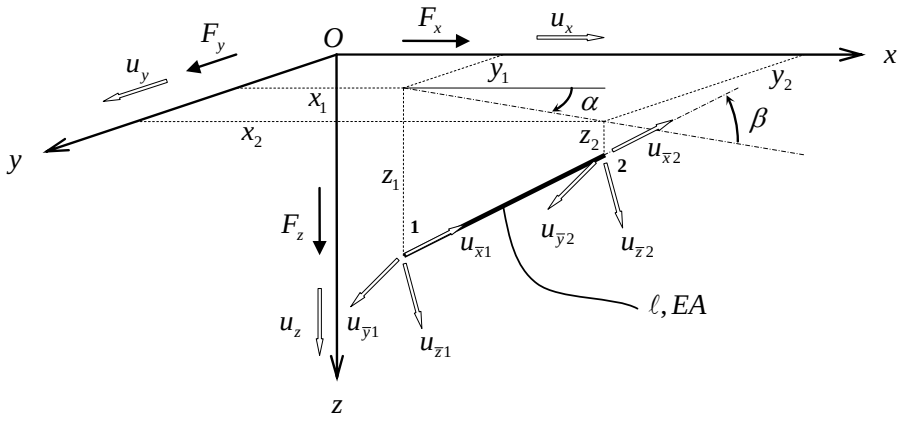


Figure 13 Strut element in a three dimensional space [figure based on Nijenhuis [8]]

The projection of the element on the Oxy-plane is at an angle α with the positive x-axis. The centerline of the strut element passes through the Oxy-plane at an angle β. The directions of the $\bar{y}$-axis and $\bar{z}$-axis in relation to the global co-ordinate system are of no importance for the stress and strain behavior of the element, since the strain only occurs in the $\bar{x}$-direction. Therefore, the $\bar{z}$-axis is chosen parallel to a vertical plane through the z-axis, and so, the two rotations over the angles α and β are sufficient. The relationship between the displacements in the local co-ordinate system $\overline{xyz}$ and the displacements in the global co-ordinate system xyz can be derived by performing these rotations consecutively. This is done by using an intermediate co-ordinate system.

First, rotation α is considered. Therefore, a new co-ordinate system $x_\alpha y_\alpha z_\alpha$ has been constructed, as can be seen from Figure 14. The x_α -axis is positioned in the Oxy -plane and points in the direction of the projection of the strut element onto the Oxy -plane. The y_α -axis is also situated in the Oxy -plane. From this, it follows that the z_α -axis coincides with the z -axis. The displacements $u_{x,i}$ and $u_{y,i}$ can be decomposed along the x_α -axis and y_α -axis

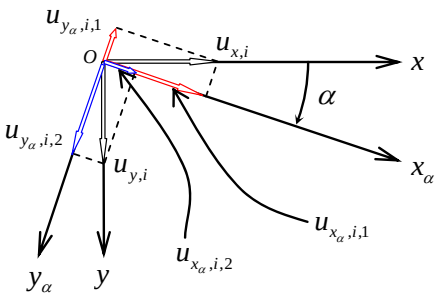


Figure 14 Decomposition of displacements in the global co-ordinate system in displacements in the $x_\alpha y_\alpha z_\alpha$ co-ordinate system [figure based on Nijenhuis [8]]

$$\begin{aligned} u_{x_\alpha,i} &= u_{x_\alpha,i,1} + u_{x_\alpha,i,2} = u_{x,i}\cos\alpha + u_{y,i}\sin\alpha, \\ u_{y_\alpha,i} &= u_{y_\alpha,i,1} + u_{y_\alpha,i,2} = -u_{x,i}\sin\alpha + u_{y,i}\cos\alpha. \end{aligned} \qquad (20)$$

Since the rotation takes place about the z -axis, it is concluded that

$$u_{z_\alpha} = u_z . \qquad (21)$$

Equations (20) and (21) may also be written in matrix notation

$$\underline{u}_{\alpha,i} = \begin{bmatrix} u_{x_\alpha,i} \\ u_{y_\alpha,i} \\ u_{z_\alpha,i} \end{bmatrix} = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} u_{x,i} \\ u_{y,i} \\ u_{z,i} \end{bmatrix} = \underline{\underline{R}}_{\alpha,i} \cdot \underline{u}_i .$$

Now, rotation β is introduced (Figure 15). The displacements $u_{x_\alpha,i}$ and $u_{z_\alpha,i}$ are decomposed along the $\bar{x}$ -axis and $\bar{z}$ -axis

$$\begin{aligned} u_{\bar{x},i} &= u_{\bar{x},i,1} + u_{\bar{x},i,2} = u_{x_\alpha,i}\cos\beta - u_{z_\alpha,i}\sin\beta, \\ u_{\bar{z},i} &= u_{\bar{z},i,1} + u_{\bar{z},i,2} = u_{x_\alpha,i}\sin\beta + u_{z_\alpha,i}\cos\beta. \end{aligned} \qquad (22)$$

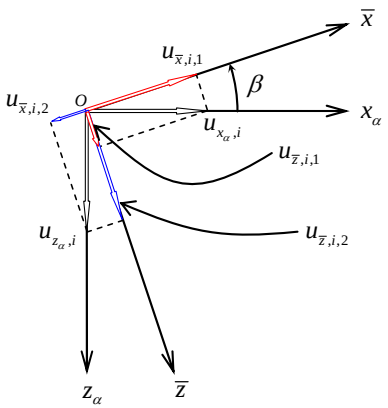


Figure 15 Decomposition of displacements in the $x_\alpha y_\alpha z_\alpha$ co-ordinate system in displacements in the local co-ordinate system [figure based on Nijenhuis [8]]

Since the rotation takes place about the y_α-axis, it is concluded that

$$u_{\bar{y},i} = u_{y_\alpha,i} . \qquad (23)$$

Equations (22) and (23) may also be written in matrix notation

$$\underline{\bar{u}}_i = \begin{bmatrix} u_{\bar{x},i} \\ u_{\bar{y},i} \\ u_{\bar{z},i} \end{bmatrix} = \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix} \cdot \begin{bmatrix} u_{x_\alpha,i} \\ u_{y_\alpha,i} \\ u_{z_\alpha,i} \end{bmatrix} = \underline{\underline{R}}_{\beta,i} \cdot \underline{u}_{\alpha,i} .$$

Now, the transformation from the global co-ordinate system to the local co-ordinate system can be written as

$$\underline{\bar{u}}_i = \underline{\underline{R}}_{\beta,i} \cdot \underline{u}_{\alpha,i} = \underline{\underline{R}}_{\beta,i} \cdot \left(\underline{\underline{R}}_{\alpha,i} \cdot \underline{u}_i \right) = \underline{\underline{R}}_{\beta,i} \cdot \underline{\underline{R}}_{\alpha,i} \cdot \underline{u}_i = \underline{\underline{R}}_i \cdot \underline{u}_i .$$

Elaboration of $\underline{\underline{R}}_i$ gives

$$\underline{\underline{R}}_i = \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix} \cdot \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\alpha\cos\beta & \sin\alpha\cos\beta & -\sin\beta \\ -\sin\alpha & \cos\alpha & 0 \\ \cos\alpha\sin\beta & \sin\alpha\sin\beta & \cos\beta \end{bmatrix} .$$

This is the rotation matrix for a single node. Since a strut element consists of two nodes, the above matrix has to be used twice

$$\underline{\bar{u}} = \begin{bmatrix} \underline{\bar{u}}_1 \\ \underline{\bar{u}}_2 \end{bmatrix} = \underline{\underline{R}} \cdot \underline{u} = \begin{bmatrix} \underline{\underline{R}}_1 & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{R}}_2 \end{bmatrix} \cdot \begin{bmatrix} \underline{u}_1 \\ \underline{u}_2 \end{bmatrix} . \qquad (24)$$

Since the angles α and β are the same for node 1 and node 2, $\underline{\underline{R}}_1$ should be equal to $\underline{\underline{R}}_2$. This gives the following rotation matrix $\underline{\underline{R}}$ for displacements of a strut element

$$\underline{\underline{R}} = \begin{bmatrix} \cos\alpha\cos\beta & \sin\alpha\cos\beta & -\sin\beta & 0 & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 & 0 & 0 \\ \cos\alpha\sin\beta & \sin\alpha\sin\beta & \cos\beta & 0 & 0 & 0 \\ 0 & 0 & 0 & \cos\alpha\cos\beta & \sin\alpha\cos\beta & -\sin\beta \\ 0 & 0 & 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & \cos\alpha\sin\beta & \sin\alpha\sin\beta & \cos\beta \end{bmatrix} .$$

The goniometric ratios $\sin\alpha$, $\cos\alpha$, $\sin\beta$ and $\cos\beta$ can be calculated from

$$\sin\alpha = \frac{y_2 - y_1}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} ,$$

$$\cos\alpha = \frac{x_2 - x_1}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}} ,$$

$$\sin\beta = \frac{z_1 - z_2}{\ell} ,$$

$$\cos\beta = \frac{\sqrt{\left(x_2 - x_1\right)^2 + \left(y_2 - y_1\right)^2}}{\ell},$$

where

$$\ell = \sqrt{\left(x_2 - x_1\right)^2 + \left(y_2 - y_1\right)^2 + \left(z_1 - z_2\right)^2}\ .$$

The next step is to derive in the same manner a rotation matrix for the forces acting on the strut element. The forces $\underline{F}_i$ acting in the global co-ordinate system xyz on node i $(i = 1,2)$ are expressed in terms of the forces $\underline{\overline{F}}_i$ acting in the local co-ordinate system $\overline{xyz}$ on node i $(i = 1,2)$. So, β is the first rotation to be considered (Figure 16)

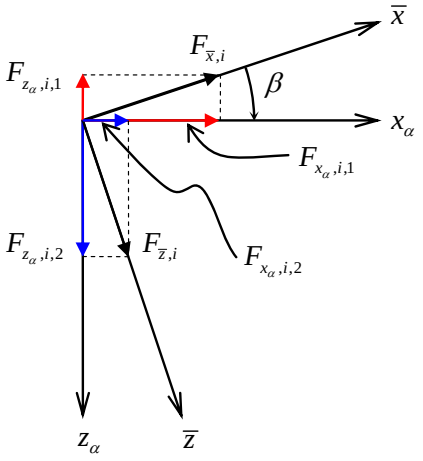


Figure 16 Decomposition of forces in the local co-ordinate system in forces in the co-ordinate system $x_\alpha y_\alpha z_\alpha$ [figure based on Nijenhuis [8]]

$$\begin{aligned} F_{x_\alpha,i} &= F_{\overline{x},i}\cos\beta + F_{\overline{z},i}\sin\beta, \\ F_{z_\alpha,i} &= -F_{\overline{x},i}\sin\beta + F_{\overline{z},i}\cos\beta. \end{aligned} \qquad (25)$$

Since the rotation takes place about the y_α-axis it is concluded that

$$F_{y_\alpha,i} = F_{\overline{y},i}\ . \qquad (26)$$

Equations (25) and (26) may also be written in matrix form

$$\underline{F}_{\alpha,i} = \begin{bmatrix} F_{x_\alpha,i} \\ F_{y_\alpha,i} \\ F_{z_\alpha,i} \end{bmatrix} = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \cdot \begin{bmatrix} F_{\overline{x},i} \\ F_{\overline{y},i} \\ F_{\overline{z},i} \end{bmatrix} = \underline{\underline{R}}^T_{\beta,i} \cdot \underline{\overline{F}}_i\ .$$

It can be checked that the formed matrix is the transposed of the matrix $\underline{\underline{R}}_{\beta,i}$, what is indicated by the superscript T.

Now, consider rotation α (Figure 17). This gives the following relations

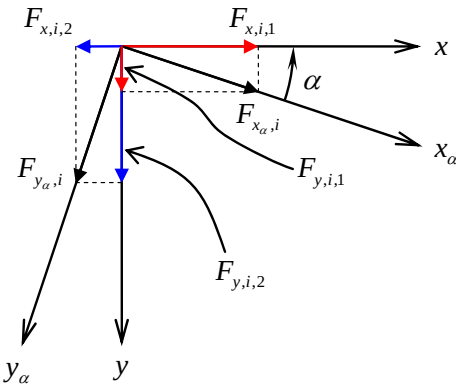


Figure 17 Decomposition of forces in the co-ordinate system $x_\alpha y_\alpha z_\alpha$ in forces in the global co-ordinate system [figure based on Nijenhuis [8]]

$$\begin{aligned} F_{x,i} &= F_{x_\alpha,i}\cos\alpha - F_{y_\alpha,i}\sin\alpha, \\ F_{y,i} &= F_{x_\alpha,i}\sin\alpha + F_{y_\alpha,i}\cos\alpha. \end{aligned} \qquad (27)$$

Since the rotation takes place about the z_α-axis, it is concluded that

$$F_{z,i} = F_{z_\alpha,i}. \qquad (28)$$

In matrix notation equations (27) and (28) read

$$\underline{F}_i = \begin{bmatrix} F_{x,i} \\ F_{y,i} \\ F_{z,i} \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} F_{x_\alpha,i} \\ F_{y_\alpha,i} \\ F_{z_\alpha,i} \end{bmatrix} = \underline{\underline{R}}^T_{\alpha,i} \cdot \underline{F}_{\alpha,i}.$$

Again, the formed matrix is the transposed of an earlier found matrix, namely $\underline{\underline{R}}_{\alpha,i}$.

Now, the transformation from the local co-ordinate system to the global co-ordinate system can be described by

$$\underline{F}_i = \underline{\underline{R}}^T_{\alpha,i} \cdot \underline{F}_{\alpha,i} = \underline{\underline{R}}^T_{\alpha,i} \cdot \left(\underline{\underline{R}}^T_{\beta,i} \cdot \underline{\bar{F}}_i \right) = \underline{\underline{R}}^T_{\alpha,i} \cdot \underline{\underline{R}}^T_{\beta,i} \cdot \underline{\bar{F}}_i = \underline{\underline{R}}^T_i \cdot \underline{\bar{F}}_i.$$

Elaboration of $\underline{\underline{R}}^T_i$ gives

$$\underline{\underline{R}}^T_i = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}$$

$$= \begin{bmatrix} \cos\alpha\cos\beta & -\sin\alpha & \cos\alpha\sin\beta \\ \sin\alpha\cos\beta & \cos\alpha & \sin\alpha\sin\beta \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}. \qquad (29)$$

As can be verified, this matrix is indeed the transposed of the earlier found rotation matrix $\underline{\underline{R}}_i$.

Just like in the case of the rotation matrix for displacements, this rotation matrix for forces has been derived for a single node. Since a strut element consists of two nodes, matrix (29) is used twice

$$\underline{F} = \begin{bmatrix} \underline{F}_1 \\ \underline{F}_2 \end{bmatrix} = \underline{\underline{R}}^T \cdot \underline{\bar{F}} = \begin{bmatrix} \underline{\underline{R}}^T_1 & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{R}}^T_2 \end{bmatrix} \cdot \begin{bmatrix} \underline{\bar{F}}_1 \\ \underline{\bar{F}}_2 \end{bmatrix}. \qquad (30)$$

Since the angles α and β are the same for node 1 and node 2, $\underline{\underline{R}}^T_1$ should be equal to $\underline{\underline{R}}^T_2$. This gives the following transposed transformation matrix $\underline{\underline{R}}^T$ for forces in a strut element

$$\underline{\underline{R}}^T = \begin{bmatrix} \cos\alpha\cos\beta & -\sin\alpha & \cos\alpha\sin\beta & 0 & 0 & 0 \\ \sin\alpha\cos\beta & \cos\alpha & \sin\alpha\sin\beta & 0 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 & 0 & 0 \\ 0 & 0 & 0 & \cos\alpha\cos\beta & -\sin\alpha & \cos\alpha\sin\beta \\ 0 & 0 & 0 & \sin\alpha\cos\beta & \cos\alpha & \sin\alpha\sin\beta \\ 0 & 0 & 0 & -\sin\beta & 0 & \cos\beta \end{bmatrix}.$$

The final step to arrive at the stiffness relation for the strut element in the global co-ordinate system is to combine equations (30), (18) and (24)

$$\underline{F} = \underline{\underline{R}}^T \cdot \underline{\bar{F}} = \underline{\underline{R}}^T \cdot \underline{\underline{\bar{K}}} \cdot \underline{\bar{u}} = \underline{\underline{R}}^T \cdot \underline{\underline{\bar{K}}} \cdot \underline{\underline{R}} \cdot \underline{u} = \underline{\underline{K}} \cdot \underline{u},$$

in which $\underline{\underline{K}} = \underline{\underline{R}}^T \cdot \underline{\underline{\bar{K}}} \cdot \underline{\underline{R}}$.

Fully written, this leads to

$$\begin{bmatrix} F_{x1} \\ F_{y1} \\ F_{z1} \\ F_{x2} \\ F_{y2} \\ F_{z2} \end{bmatrix} = \frac{EA}{\ell} \begin{bmatrix} \cos^2\alpha\cos^2\beta & \cos\alpha\cos^2\beta\sin\alpha & -\cos\alpha\cos\beta\sin\beta & -\cos^2\alpha\cos^2\beta & -\cos\alpha\cos^2\beta\sin\alpha & \cos\alpha\cos\beta\sin\beta \\ \cos\alpha\cos^2\beta\sin\alpha & \sin^2\alpha\cos^2\beta & -\sin\alpha\cos\beta\sin\beta & -\cos\alpha\cos^2\beta\sin\alpha & -\sin^2\alpha\cos^2\beta & \sin\alpha\cos\beta\sin\beta \\ -\cos\alpha\cos\beta\sin\beta & -\sin\alpha\cos\beta\sin\beta & \sin^2\beta & \cos\alpha\cos\beta\sin\beta & \sin\alpha\cos\beta\sin\beta & -\sin^2\beta \\ -\cos^2\alpha\cos^2\beta & -\cos\alpha\cos^2\beta\sin\alpha & \cos\alpha\cos\beta\sin\beta & \cos^2\alpha\cos^2\beta & \cos\alpha\cos^2\beta\sin\alpha & -\cos\alpha\cos\beta\sin\beta \\ -\cos\alpha\cos^2\beta\sin\alpha & -\sin^2\alpha\cos^2\beta & \sin\alpha\cos\beta\sin\beta & \cos\alpha\cos^2\beta\sin\alpha & \sin^2\alpha\cos^2\beta & -\sin\alpha\cos\beta\sin\beta \\ \cos\alpha\cos\beta\sin\beta & \sin\alpha\cos\beta\sin\beta & -\sin^2\beta & -\cos\alpha\cos\beta\sin\beta & -\sin\alpha\cos\beta\sin\beta & \sin^2\beta \end{bmatrix} \begin{bmatrix} u_{x1} \\ u_{y1} \\ u_{z1} \\ u_{x2} \\ u_{y2} \\ u_{z2} \end{bmatrix} \quad (31)$$

In the case of vertical strut elements, which is for example the case with foundation piles, angle α is not defined (Figure 13). This means that equation (31) cannot be used. Instead, the element stiffness matrix is redefined, based on equation (19). The difference is that in this case the strut element is not oriented along the $\bar{x}$ -axis, but along the $\bar{z}$ -axis. Therefore, the stiffness relation for a vertical strut element reads

$$\frac{EA}{\ell} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & +1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & +1 \end{bmatrix} \begin{bmatrix} u_{\bar{x}1} \\ u_{\bar{y}1} \\ u_{\bar{z}1} \\ u_{\bar{x}2} \\ u_{\bar{y}2} \\ u_{\bar{z}2} \end{bmatrix} = \begin{bmatrix} F_{\bar{x}1} \\ F_{\bar{y}1} \\ F_{\bar{z}1} \\ F_{\bar{x}2} \\ F_{\bar{y}2} \\ F_{\bar{z}2} \end{bmatrix}.$$

Global numbering of strut elements and their DOF and generating strut elements in the applet, is discussed in Appendix A3. Once the displacements $u_{\bar{x}1}$ and $u_{\bar{x}2}$ are known, the normal force in the strut element can be calculated from

$$N = -F_{\bar{x}1} = -\frac{EA}{\ell}\left(u_{\bar{x}1} - u_{\bar{x}2}\right),$$

which has been derived from equation (19). The displacements $u_{\bar{x}1}$ and $u_{\bar{x}2}$ can be calculated from the earlier derived rotation matrix $\underline{\underline{R}}$ and the displacements u_{x1}, u_{x2}, u_{y1}, u_{y2}, u_{z1} and u_{z2}. If this is done, the normal force in the strut element can be calculated from

$$N = -\frac{EA}{\ell}\left(\cos\alpha\cos\beta\left(u_{x1}-u_{x2}\right)+\sin\alpha\cos\beta\left(u_{y1}-u_{y2}\right)-\sin\beta\left(u_{z1}-u_{z2}\right)\right).$$

For vertical strut elements, calculating the normal force is easier. Since only displacements in $\bar{z}$ -direction are needed and this direction coincides with the z -direction, the normal force can be calculated from

$$N = -F_{\bar{z}1} = -\frac{EA}{\ell}\left(u_{\bar{z}1}-u_{\bar{z}2}\right) = -\frac{EA}{\ell}\left(u_{z1}-u_{z2}\right).$$

4 Assembling the model and solving the system

In Chapter 3 the behavior of the elements to be used was described. In this chapter it is explained how to describe the behavior of the structure, which is composed of these elements and how to solve the resulting system of linear equations. In Section 4.1 it is explained how to assemble the system stiffness matrix starting from the element stiffness matrices. In Section 4.2 it is discussed how to process the imposed forces. Section 4.3 explains how to make use of so-called tying. Section 4.4 discusses processing the supports, which may be regarded as imposed displacements. Finally, Section 4.5 contains a brief description of how to solve the obtained system of linear equations by using the method of LU decomposition.

4.1 Assembling the system stiffness matrix

Assume that the considered system has n degrees of freedom. The system stiffness matrix will then have dimensions of $n \times n$. The most important part in assembling the system stiffness matrix is to find the corresponding local and global degrees of freedom (DOF). In this way, the entries of the element stiffness matrices are added to the correct entries of the system stiffness matrix. This procedure has been visualized for a stringer element in Figure 18. Consider an arbitrary stringer element with element number i. From Section 3.1 it is known that a stringer element possesses three DOF. Locally, these are called 1, 2 and 3, but globally these may be called j, k and l. If the corresponding entries have been found, a summation of the entry of the element stiffness matrix and the entry of the system stiffness matrix takes place.

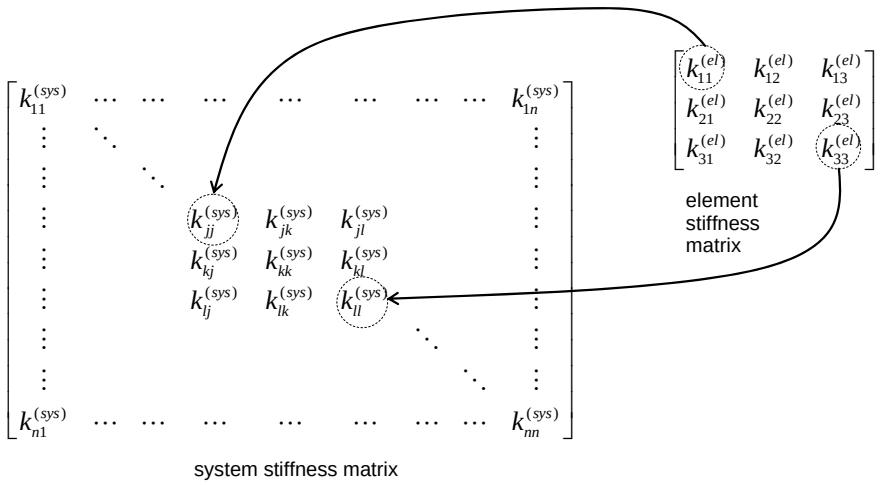


Figure 18 Assembling the system stiffness matrix

The same procedure may be applied to strut elements and shear panel elements, with the difference that the element stiffness matrices have different sizes. In Appendix B1, the source code for this procedure is given.

4.2 Processing imposed forces

Processing the imposed forces comes down to nothing more than assigning the column normal forces to the correct entries of the force vector. No more work has to be performed in this step, since the column normal forces are the only loads applied to the pile cap. The implementation of this procedure is given in Appendix B2.

4.3 Processing tying

Since strut elements are attached to the interior of shear panel elements, so-called tying is needed. This means that the normal forces in the stringer elements adjacent to these shear panel elements are not independent, but related to the horizontal components of the strut element normal forces. In a similar way it can be said that the horizontal displacements of a strut element end are not independent, but related to the displacements of the shear panel elements. These displacements are equal to the displacements of the adjacent stringer elements.

Consider a shear panel element to which a strut element is attached (Figure 19).

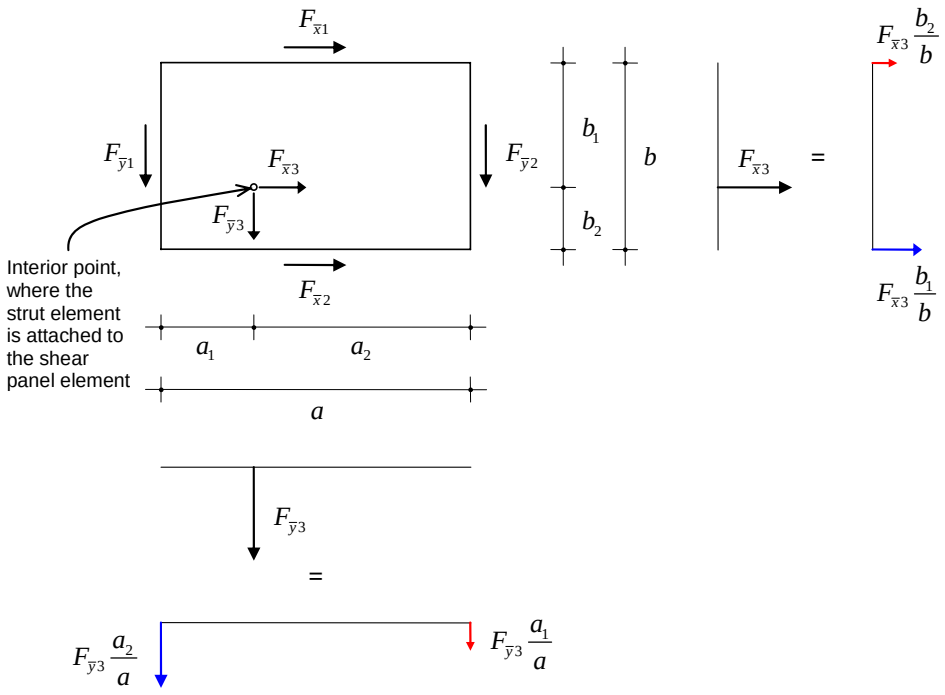


Figure 19 Relations between horizontal force components of the strut element normal force and the shear forces acting on the edges of the shear panel element

Since strut elements can only transfer normal forces (Section 3.4) and since the piles have been modeled as strut elements, it is clear that the piles can only accommodate the vertical components of the strut element normal forces. In the general case, a strut element normal force is composed of one vertical force component and two horizontal force components. Since these horizontal force components cannot be transferred to the piles,

these need to be transferred to the stringer elements adjacent to the considered shear panel element. Since it is not unambiguously established how the horizontal force components of the strut element normal force are distributed over the adjacent stringer elements, linear relations are assumed. For the forces in x-direction these relations read

$$F_{\bar{x}1} = \frac{b_2}{b} \cdot F_{\bar{x}3} \text{ and } F_{\bar{x}2} = \frac{b_1}{b} \cdot F_{\bar{x}3}.$$

For the forces in y-direction these relations read

$$F_{\bar{y}1} = \frac{a_2}{a} \cdot F_{\bar{y}3} \text{ and } F_{\bar{y}2} = \frac{a_1}{a} \cdot F_{\bar{y}3}.$$

The introduction of these force components into the stringer elements leads to a certain displacement field for the stringer elements and the shear panel elements. Since the strut elements are attached to the interior of the shear panel elements, this indicates that the horizontal displacements of the strut element ends are related to the displacements of the shear panel element edges. As stated earlier in this section, the displacements of the shear panel elements are equal to the displacements of the adjacent stringer elements. Again, since it is not unambiguously established how the displacements of the shear panel element are related to the horizontal displacements of the strut element end, linear relations are assumed (Figure 20).

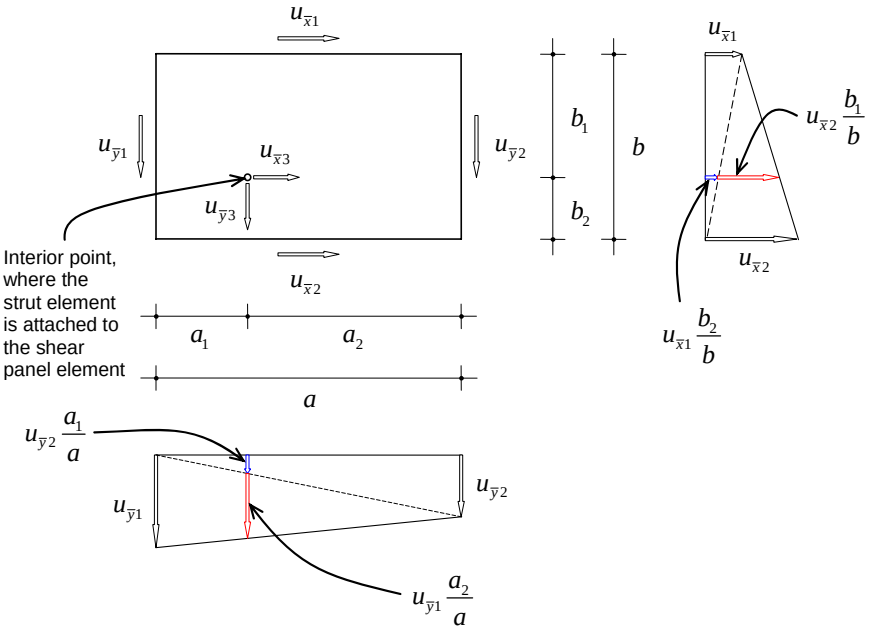


Figure 20 Relations between the displacements of a strut element end and the displacements of a shear panel element

For the displacements in x-direction, this relation holds

$$u_{\bar{x}3} = u_{\bar{x}1}\frac{b_2}{b} + u_{\bar{x}2}\frac{b_1}{b}.$$

For the displacements in y-direction, this relation holds

$$u_{\bar{y}3} = u_{\bar{y}2}\frac{a_1}{a} + u_{\bar{y}1}\frac{a_2}{a}.$$

Now suppose that the following system stiffness relation was derived after performing the steps described in Sections 4.1 and 4.2

$$\begin{bmatrix} F_{x1} \\ F_{x2} \\ F_{x3} \\ F_{x4} \\ \vdots \\ F_{x5} \end{bmatrix} = \begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} & \cdots & k_{1n} \\ k_{21} & k_{22} & k_{23} & k_{24} & \cdots & k_{2n} \\ k_{31} & k_{32} & k_{33} & k_{34} & \cdots & k_{3n} \\ k_{41} & k_{42} & k_{43} & k_{44} & \cdots & k_{4n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ k_{n1} & k_{n2} & k_{n3} & k_{n4} & \cdots & k_{nn} \end{bmatrix} \begin{bmatrix} u_{x1} \\ u_{x2} \\ u_{x3} \\ u_{x4} \\ \vdots \\ u_{xn} \end{bmatrix}. \qquad (32)$$

To demonstrate in a simple way how tying is processed, only forces and displacements in x-direction are considered. It is obvious that this procedure can be extended easily to include forces and displacements in y-direction as well. Fully written, matrix equation (32) reads:

$$\begin{aligned} F_{x1} &= k_{11}u_{x1} + k_{12}u_{x2} + k_{13}u_{x3} + k_{14}u_{x4} + \ldots + k_{1n}u_{xn}, \\ F_{x2} &= k_{21}u_{x1} + k_{22}u_{x2} + k_{23}u_{x3} + k_{24}u_{x4} + \ldots + k_{2n}u_{xn}, \\ F_{x3} &= k_{31}u_{x1} + k_{32}u_{x2} + k_{33}u_{x3} + k_{34}u_{x4} + \ldots + k_{3n}u_{xn}, \\ F_{x4} &= k_{41}u_{x1} + k_{42}u_{x2} + k_{43}u_{x3} + k_{44}u_{x4} + \ldots + k_{4n}u_{xn}, \\ &\vdots \\ F_{xn} &= k_{n1}u_{x1} + k_{n2}u_{x2} + k_{n3}u_{x3} + k_{n4}u_{x4} + \ldots + k_{nn}u_{xn}. \end{aligned} \qquad (33)$$

Now suppose that for a certain shear panel element the following relations hold (Figure 21):

$$F_{x1} = \beta_1 F_{x3}, \qquad (34)$$

$$F_{x2} = \beta_2 F_{x3}, \qquad (35)$$

$$u_{x3} = \beta_1 u_{x1} + \beta_2 u_{x2}, \qquad (36)$$

where $\beta_1 = \frac{b_2}{b}$ and $\beta_2 = \frac{b_1}{b}$.

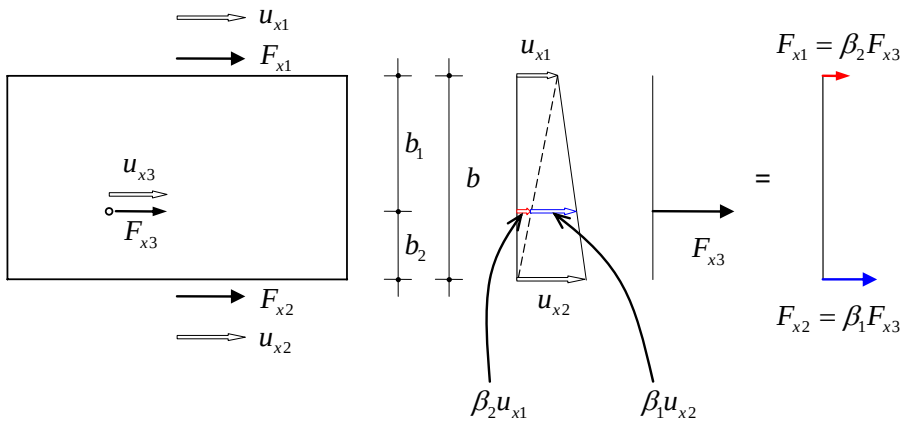


Figure 21 Tying in x -direction

Equation (34) states that F_{x1} is equal to F_{x3} times a factor β_1. The equation for F_{x3} can be found in the third row of the system of equations (33). So multiplication of this third row with factor β_1 and addition of these terms to row one, inserts equation (34) into the system of equations. In the same manner it can be shown that inserting equation (35) into the system of equations results in the addition of factor β_2 times the third row to the second row. The last tying equation (36) can be inserted into the system of equations by rearranging the terms in this equation

$$u_{x3} = \beta_1 u_{x1} + \beta_2 u_{x2} \Rightarrow \beta_1 u_{x1} + \beta_2 u_{x2} - u_{x3} = 0 .$$

If these operations are carried out, then the following system of equations results

$$\begin{bmatrix} F_{x1} + \beta_1 F_{x3} \\ F_{x2} + \beta_2 F_{x3} \\ 0 \\ F_{x4} \\ \vdots \\ F_{xn} \end{bmatrix} = \begin{bmatrix} k_{11} + \beta_1 k_{31} & k_{12} + \beta_1 k_{32} & k_{13} + \beta_1 k_{33} & k_{14} + \beta_1 k_{34} & \cdots & k_{1n} + \beta_1 k_{3n} \\ k_{21} + \beta_2 k_{31} & k_{22} + \beta_2 k_{32} & k_{23} + \beta_2 k_{33} & k_{24} + \beta_2 k_{34} & \cdots & k_{2n} + \beta_2 k_{3n} \\ \beta_1 & \beta_2 & -1 & 0 & \cdots & 0 \\ k_{41} & k_{42} & k_{43} & k_{44} & \cdots & k_{4n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ k_{n1} & k_{n2} & k_{n3} & k_{n4} & \cdots & k_{nn} \end{bmatrix} \begin{bmatrix} u_{x1} \\ u_{x2} \\ u_{x3} \\ u_{x4} \\ \vdots \\ u_{xn} \end{bmatrix} .$$

It has to be noted that a slight incompatibility occurs. Silently it is assumed that for a stringer element the mean displacement, which corresponds to the displacement of the intermediate DOF, is equal to the displacement in the middle of the stringer element. Appendix B3 gives the source code for processing tying.

4.4 Processing imposed displacements

The text in this section is based on Blaauwendraad (2000) [3]. Consider the following system stiffness relation for an n DOF pile cap after performing the steps described in Sections 4.1, 4.2 and 4.3

$$\begin{bmatrix} k_{11} & k_{12} & \cdots & k_{1i} & \cdots & k_{1n} \\ k_{21} & k_{22} & \cdots & k_{2i} & \cdots & k_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ k_{i1} & k_{i2} & \cdots & k_{ii} & \cdots & k_{in} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ k_{n1} & k_{n2} & \cdots & k_{ni} & \cdots & k_{nn} \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_i^0 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_i + F_i^0 \\ \vdots \\ F_n \end{bmatrix}. \qquad (37)$$

A certain number of DOF has to be prescribed, such that rigid body movements are restrained. Otherwise the system stiffness matrix $\underline{\underline{K}}$ would be singular, which means that the system of equations cannot be solved. Assume that the i-th DOF u_i has been set to a value u_i^0. If this value is equal to zero, this means that the pile cap is supported at this point in the direction of this DOF. A value u_i^0 unequal to zero occurs for example in the case of support settlement. In equation (37) F_i represents the load applied to the i-th DOF. F_i^0 represents the support reaction at the i-th DOF, which is unknown beforehand and therefore has to be calculated.

From equation (37) it follows that the terms in the i-th column of the system stiffness matrix are multiplied by a known value u_i^0. This implies that in each row of the system a known term will occur. In the first row this term is equal to $k_{1i}u_i^0$, in the second row this term is equal to $k_{2i}u_i^0$, and so forth. Each term can be removed from the system stiffness matrix and added to the right hand side vector. This means that the first row of this vector reads $F_1 - k_{1i}u_i^0$, the second row reads $F_2 - k_{2i}u_i^0$, and so forth. This implies that the i-th column fills up with zeros. Because F_i^0 in the right hand side vector is still unknown, the terms in the i-th row cannot be used in this stage and are therefore replaced by zeros. Only the diagonal term is set to 1, which leads to the identity $1 \times u_i^0 = u_i^0$. The system of equations now reads

$$\begin{bmatrix} k_{11} & k_{12} & \cdots & 0 & \cdots & k_{1n} \\ k_{21} & k_{22} & \cdots & 0 & \cdots & k_{2n} \\ \vdots & \vdots & \ddots & 0 & \ddots & \vdots \\ 0 & 0 & 0 & 1 & 0 & 0 \\ \vdots & \vdots & \cdots & 0 & \ddots & \vdots \\ k_{n1} & k_{n2} & \cdots & 0 & \cdots & k_{nn} \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_i^0 \\ \vdots \\ u_n \end{bmatrix} = \begin{bmatrix} F_1 - k_{1i}u_i^0 \\ F_2 - k_{2i}u_i^0 \\ \vdots \\ u_i^0 \\ \vdots \\ F_n - k_{ni}u_i^0 \end{bmatrix}.$$

Once the displacements $\underline{u}$ are known, the support reactions F_i^0 may be calculated. For this, the earlier removed entries of the i-th row are needed again. Therefore, it is necessary that these entries are stored elsewhere before they are replaced by zeros. Appendix B4 gives the source code for processing imposed displacements.

4.5 Solving the obtained system of linear equations

For solving the obtained system of linear equations, LU decomposition is used. The brief description in this section gives a general approach. Therefore, the considered equation reads $\underline{\underline{A}} \cdot \underline{x} = \underline{b}$ instead of $\underline{\underline{K}} \cdot \underline{u} = \underline{F}$. But obviously, the described method is in its entirety applicable to the system of equations described in the previous sections. Solving for $\underline{x}$ involves the following three steps:

1. Decompose the obtained matrix $\underline{\underline{A}}$ into a matrix product $\underline{\underline{L}} \cdot \underline{\underline{U}}$, where $\underline{\underline{L}}$ is a lower triangular matrix and $\underline{\underline{U}}$ is an upper triangular matrix. A lower triangular matrix is a matrix which has elements unequal to zero only on the main diagonal and below. An upper triangular matrix is a matrix which has elements unequal to zero only on the main diagonal and above. The obtained matrix equation now reads $\left(\underline{\underline{L}} \cdot \underline{\underline{U}}\right) \cdot \underline{x} = \underline{b}$.
2. Rewrite the above stated equation to $\underline{\underline{L}} \cdot \left(\underline{\underline{U}} \cdot \underline{x}\right) = \underline{b}$ and define a vector $\underline{y}$ such that $\underline{\underline{L}} \cdot \underline{y} = \underline{b}$ and solve for this vector by performing a forward substitution.
3. Solve for vector $\underline{x}$ by using the definition $\underline{\underline{U}} \cdot \underline{x} = \underline{y}$. This can be done by performing a backward substitution.

These steps including the implementation of LU decomposition and testing this separate module are elaborated in Appendix B5.

5 Applet design

In Chapter 4 the setup of the calculation model was discussed. This chapter describes how the applet is designed. Section 5.1 gives a short general introduction to applets and explains the program setup, which is based on suggestions given at Sun's Java tutorial website [16]. Section 5.2 goes into the first main procedure: the preprocessor. Section 5.3 discusses the second main procedure, which is the calculation part of the program: the kernel. In the last section, Section 5.4, it is explained how the results calculated by the kernel are processed and presented in an user-friendly way. This third and last main procedure is called the postprocessor. In the next chapter, Chapter 6, the Java implementation is tested.

5.1 Applet setup and Java basics

According to Sun's web pages on applets [15] an applet can be defined as: '… a program written in the Java programming language that can be included in an HTML page, …'. The advantages of using an applet over a standalone application are threefold. The first advantage is that no installation is required. A simple computer with an Internet connection and a Java technology-enabled browser to view a page that contains an applet, is sufficient. Secondly, an applet which is made available through the Internet is accessible from every computer which has an Internet connection at its disposal. The third advantage is that applets cannot harm a client's system, since they have very limited access to system resources and system information.

The program consists of several classes and nested classes. In the Java programming language, a program is built starting from classes. According to Sun's lesson on Object-Oriented Programming Concepts [17], a class is defined as: '…a blueprint or prototype from which objects are created.'. An object is defined as: '…a software bundle of related state and behavior'. It is possible to define a class within another class. Such classes are called nested classes. Two advantages of nested classes over normal classes are that 'it is a way of logically grouping classes that are only used in one place' and that 'nested classes can lead to more readable and maintainable code'. The program-specific class `PileCapApplet` and its nested classes have been grouped in a package called `pilecap`. A package is a namespace for organizing classes (and interfaces) in a logical manner. The remaining classes

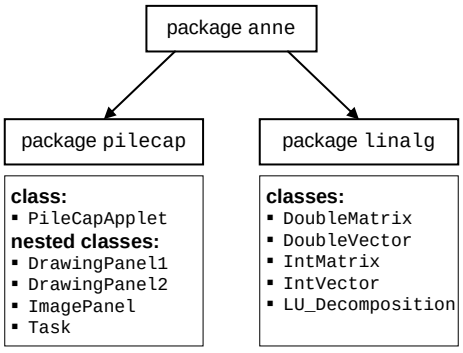


Figure 22 Applet setup in terms of packages and (nested) classes

have been grouped in a package which holds classes for linear algebra, called `linalg`. These two packages are contained in another package, called `anne` (Figure 22). The folders which contain the `.class`-files are also organized in this way. The source code of the classes `DoubleMatrix`, `DoubleVector`, `IntMatrix` and `IntVector` is given in Appendix C1.

Inside a class, procedures are defined, which may be called by other procedures or events fired by user actions. The applet possesses three main procedures: the preprocessor, the kernel and the postprocessor. These procedures are examined in the following sections.

The applet is a graphical application, so that the user does not have to know anything about the Java programming language. The Graphical User Interface (GUI) on startup is shown in Figure 23.

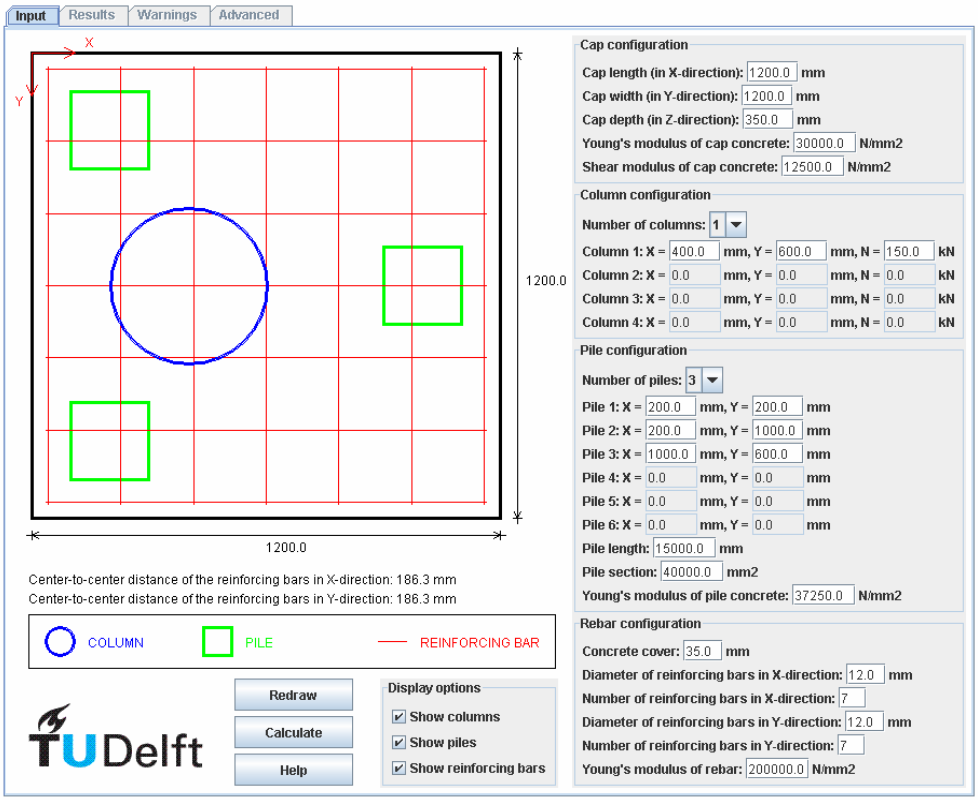


Figure 23 Input screen of PCA at startup

The input screen has been divided in two parts: one textual part which consists of text fields and combo boxes in which the user can insert data, and one graphical part which displays the pile cap plan view. By pressing the 'Redraw' button, the user can verify graphically that the geometrical parameters are correct. By pressing the 'Calculate' button the preprocessor is started which reads the entered data and checks the validity of the data. If all data has been specified correctly, the kernel is started, which sets up the system of equations and solves this system. If the results returned by the kernel are valid, the

postprocessor is started, which calculates all internal stresses and vertical pile reactions and displays them graphically.
If during the analysis an error occurs, the user is warned by a message which shows the cause of the error. Processing is then stopped and the user returns to the input screen to adjust one or more parameters. If the analysis was successful, the user is automatically taken to the results screen, which displays the obtained results in an orderly manner. If the user needs more detailed information on the results, use can be made of the 'Advanced' tab. If any warnings appeared during the analysis, these are collected in the 'Warnings' tab and the user is notified of these warnings by displaying a message.

5.2 Preprocessor

The preprocessor reads all data entered by the user if the 'Calculate' button (Figure 23) is clicked. Then it prepares these data for calculation. This means that all data is checked and if specified in a valid interval, the values are assigned to the appropriate variables. Then the model is generated, which means that all elements and their DOF are numbered. If all goes well, the `startCalculation` flag has the value `true` and a calculation task is created. The purpose of creating a separate task is to keep the GUI responsive. If the kernel was called directly from the preprocessor, the calculation would be performed on the so-called Event Dispatching Thread, which results in a 'frozen' GUI for long tasks. Therefore, calculations are performed on a separate thread. After the calculation has been started, the 'Calculate' button cannot be clicked anymore and the cursor is set to a 'wait' cursor. After completing the calculation procedure, these actions are undone. To make the source code readable and maintainable, the preprocessor calls other procedures to fulfil the described tasks. The names of these procedures are self-explaining. The source code of the procedure `preprocessor ()` reads

```
private void preprocessor ()
{

    startCalculation = true;
    showErrorMessage = true;
    setCapLength ();
    setCapWidth ();
    setCapDepth ();
    setE_CapConcrete ();
    setG_CapConcrete ();
    setColumnXY ();
    setColumnNormalForces ();
    setPileXY ();
    setPileLength ();
    setPileSection ();
    setE_PileConcrete ();
    setConcreteCover ();
    setRebarDiameterX ();
    setNrOfRebarsX ();
    setCtcDistanceOfRebarsX ();
    setRebarDiameterY ();
    setNrOfRebarsY ();
    setCtcDistanceOfRebarsY ();
    setE_Rebars ();
    generateStringers ();
    generateShearPanels ();
    generateStruts ();
```

```
        generateImposedForces ();
        generateTyings ();
        generateImposedDisplacements ();


        // if all input has been specified correctly then start the calculation

        if (startCalculation)
        {

            task = new Task ();
            task.execute ();
            calculateButton.setEnabled (false);
            setCursor (Cursor.getPredefinedCursor (Cursor.WAIT_CURSOR));

        }

    }
```

5.3 Kernel

The kernel is started by an object called `task`. In the kernel, the system stiffness matrix is generated, the boundary conditions are processed and the resulting system of linear equations is solved. After solving the displacement field and before starting the postprocessor, the displacements are checked on their magnitude. If one or more displacements exceed 100 mm, this is an indication for a kinematical indeterminate structure. Then further processing is aborted, because the flag `showResults` is set to `false`. For the same reasons given in Section 5.2, the kernel itself only calls other procedures to execute these tasks. Again, the names of these procedures have been chosen such that these are self-explaining. The source code of the procedure `kernel ()` reads

```
private void kernel ()
{

    showResults = true;
    initialiseSystemStiffnessMatrix ();
    assembleStringers ();
    assembleShearPanels ();
    assembleStruts ();
    processImposedForces ();
    processTyings ();
    processImposedDisplacements ();
    solveSystem ();


    // if the pile cap appears to be stable then start postprocessing

    if (showResults)
    {

        postprocessor ();

    }

}
```

5.4 Postprocessor

If the postprocessor is started, internal stresses and pile reactions are calculated. Moreover, the (absolute) maximum values of the stresses are determined, which are of interest for the structural designer. Also the stress results are graphically displayed on the screen (Figure 24).

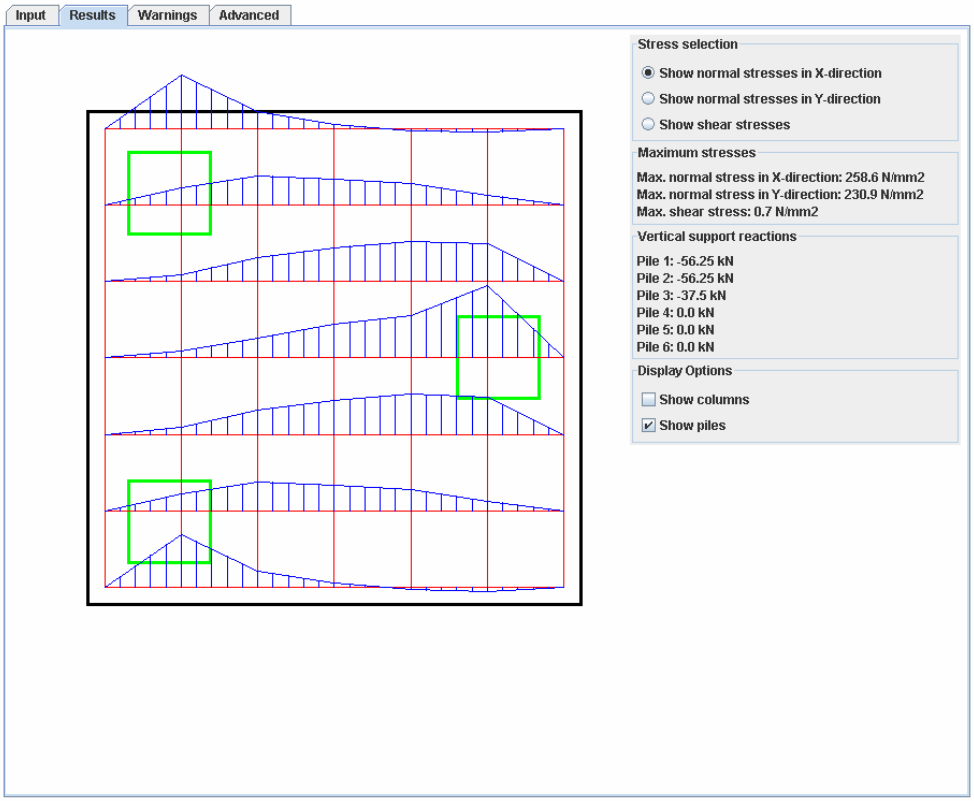


Figure 24 Screenshot of the 'Results' tab

Again, this main procedure makes primarily use of procedures that are defined elsewhere in the source file. This makes the source code of the postprocessor readable and maintainable. The source code of the procedure `postprocessor ()` reads

```
private void postprocessor ()
{

    tabbedPane.setEnabledAt (1, true);    // make results tab accessible
    tabbedPane.setEnabledAt (2, true);    // make warnings tab accessible
    tabbedPane.setEnabledAt (3, true);    // make advanced tab accessible
    tabbedPane.setSelectedIndex (1);      // set results tab as selected tab
    calculateStringerStresses ();
    calculateShearPanelStresses ();
    calculateStrutForces ();
    calculateSupportReactions ();
    determineMaxValues ();


    // if this is the first calculation then create the results GUI,
    // otherwise refresh the results GUI using the new results
```

```
        if (firstCalculation)
        {

            createResultsGUI ();


            // create advanced tab

            textArea = new JTextArea ();
            textArea.setEditable (false);
            JScrollPane scrollPane = new JScrollPane (textArea);
            scrollPane.setPreferredSize (new Dimension (880, 700));
            panel1c.add (scrollPane);


            // create warnings tab

            textArea2 = new JTextArea ();
            textArea2.setEditable (false);
            JScrollPane scrollPane2 = new JScrollPane (textArea2);
            scrollPane2.setPreferredSize (new Dimension (880, 700));
            panel1d.add (scrollPane2);


            calculateButton.setText ("Recalculate");
            firstCalculation = false;

        }
        else
        {

            updateLabels ();
            stressXX_RadioButton.setSelected (true);
            panel1b.repaint ();
            textArea.setText ("");
            textArea2.setText ("");

        }


        // add detailed information to advanced tab

        showParameters ();
        showStringerInfo ();
        showShearPanelInfo ();
        showStrutInfo ();
        showImposedForcesInfo ();
        showTyingInfo ();
        showImposedDisplacements ();
        showSupportReactions ();


        // show warning information

        createWarningInfo ();

    }
```

6 Equilibrium considerations

The previous chapter discussed the design of the applet. In this chapter the Java implementation is tested by checking equilibrium requirements. Section 6.1 considers a symmetrical pile cap which consists of one column and three piles. Section 6.1.1 focuses on the equilibrium of the whole structure. This includes vertical equilibrium of forces, horizontal equilibrium of forces and moment equilibrium. In Section 6.1.2 two separate cuts are made, after which the equilibrium of forces is considered for one part of the structure. Section 6.2 involves an asymmetrical pile cap which consists of two columns and six piles. Here only the force equilibrium of the whole structure is considered. The next chapter discusses checking the ultimate load.

6.1 Case 1: Symmetrical pile cap consisting of three piles and one column

Input parameters

The input parameters which have been used for the calculation are given in Table 1. A screenshot of the pile cap plan view is provided in Figure 25.

	Cap configuration		
Cap length (in x-direction):	1600 mm		
Cap width (in y-direction):	1400 mm		
Cap depth (in z-direction):	400 mm		
Young's modulus of cap concrete:	9000 N/mm^2		
Shear modulus of cap concrete:	4500 N/mm^2		
	Column configuration		
Number of columns:	1		
Column 1:	$x = 600$ mm	$y = 700$ mm	$N = 200$ kN
	Pile configuration		
Number of piles:	3		
Pile 1:	$x = 250$ mm	$y = 250$ mm	
Pile 2:	$x = 250$ mm	$y = 1150$ mm	
Pile 3:	$x = 1350$ mm	$y = 700$ mm	
Pile length:	15000 mm		
Pile section:	40000 mm^2		
Young's modulus of pile concrete:	12000 N/mm^2		
	Reinforcing bar configuration		
Concrete cover:	40 mm		
Diameter of reinforcing bars in x-direction:	16 mm		
Number of reinforcing bars in x-direction:	7		
Diameter of reinforcing bars in y-direction:	16 mm		
Number of reinforcing bars in y-direction:	8		
Young's modulus of reinforcing bars:	200000 N/mm^2		

Table 1 Pile cap parameters for Case 1

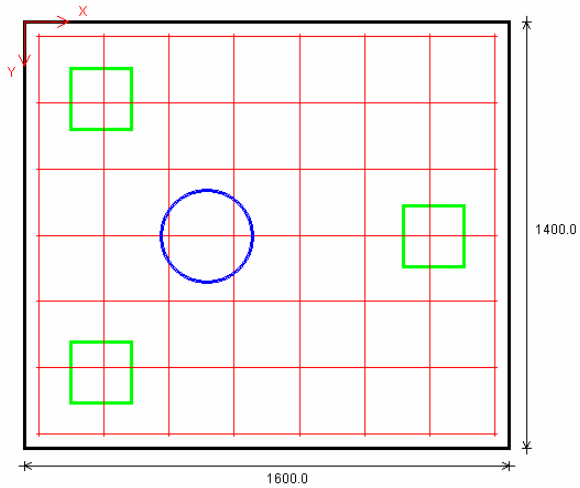


Figure 25 Screenshot of the pile cap geometry in Case 1

Pile 1:	-68.18 kN
Pile 2:	-68.18 kN
Pile 3:	-63.64 kN
Total:	-200.0 kN

Table 3 Vertical pile reactions in Case 1

6.1.1 Equilibrium consideration of the whole structure

Vertical forces

The vertical support reactions of the piles can be read from the Results tab after performing the calculation. These values have been collected in Table 3. The minus sign indicates that the piles are in compression. It is clear that the sum of the vertical pile reactions in equilibrium is with the column load.

Horizontal forces

The horizontal support reactions in x-direction can be read from the Advanced tab after performing the calculation. These values have been collected in Table 2. Since no horizontal load is applied in x-direction, the sum of the horizontal support reactions in x-direction has to be zero, which is true accepting a round-off error. Furthermore, the stresses at the ends of the stringer elements near the edges of the pile cap have to be equal to zero. This follows from the equilibrium of such a stringer element end. By plotting the reinforcement stresses in x-direction, it can be verified that Case 1 complies with this requirement (Figure 26). It is important to note that in practice the reinforcement stresses at these positions are <u>not</u> equal to zero, and therefore the reinforcing bars have to be anchored in some way. Usually this is done by applying hooks.

Support reaction 1:	$-6.895 \cdot 10^{-11}$ kN
Support reaction 3:	$-5.190 \cdot 10^{-11}$ kN
Support reaction 4:	0.0 kN
Support reaction 7:	0.0 kN
Support reaction 10:	0.0 kN

Table 2 Horizontal support reactions (in x-direction) in Case 1

The horizontal support reactions in y-direction can also be read from the 'Advanced' tab. These values have been collected in Table 4. Since no horizontal load is applied in y-direction, the sum of the horizontal support reactions in y-direction has to be zero, which is true accepting a round-off error.

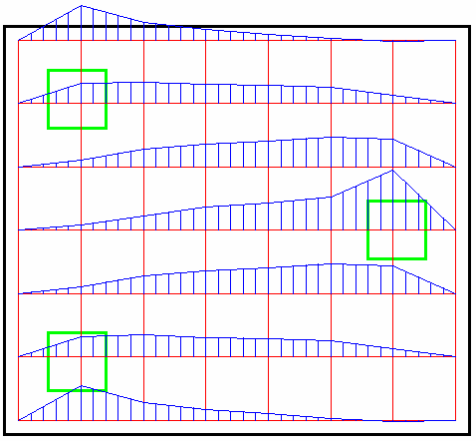


Figure 26 Reinforcement stresses in x-direction

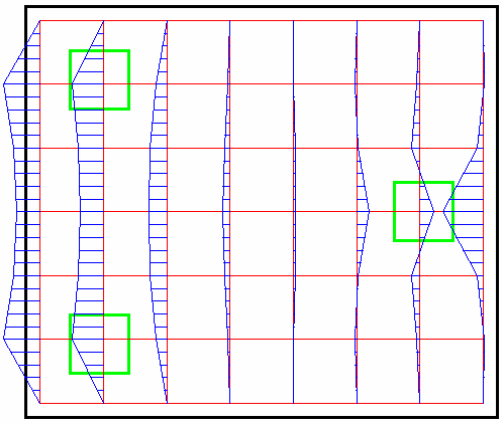


Figure 27 Reinforcement stresses in y-direction

Furthermore, the stresses at the ends of the stringer elements near the edges of the pile cap have to be equal to zero. This follows from the equilibrium of such a stringer end. By plotting the reinforcement stresses in y-direction, it can be verified that Case 1 complies with this requirement, see Figure 27. Again, it is important to notice that in practice the reinforcement stresses at these positions are not equal to zero.

Support reaction 2:	$2{,}604 \cdot 10^{-11}$ kN
Support reaction 5:	0,0 kN
Support reaction 8:	0,0 kN
Support reaction 11:	0,0 kN

Table 4 Horizontal support reactions (in y-direction) in Case 1

Symmetry

From the input parameters in Table 1 and the screenshot in Figure 25 it is clear that the pile cap is symmetrical about the line y = 700 mm. The stresses in the reinforcing bars in x-direction that are symmetrically positioned about this line are identical (Figure 26). The stresses in the reinforcing bars in y-direction are also symmetrical about the line y = 700 mm (Figure 27). Also the shear stresses are symmetrical about this line (Figure 28). From Table 3 it follows that the vertical pile reactions are also symmetrical.

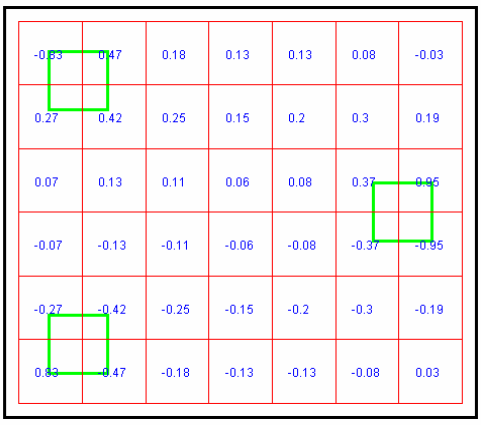


Figure 28 Shear stresses (in N/mm^2)

Moment equilibrium

The third and last equilibrium requirement is moment equilibrium of the whole structure. First the moment equilibrium about the line x = 250 mm is checked

$$200 \text{ kN} \times (0.600 - 0.250) \text{ m} - 63.64 \text{ kN} \times (1.35 - 0.250) \text{ m} \cong 0.00 \text{ kNm}.$$

Secondly, the moment equilibrium about the line $x = 1350$ mm is checked

$$200 \text{ kN} \times (1.35 - 0.600) \text{ m} + 2 \times (-68.18 \text{ kN}) \times (1.35 - 0.250) \text{ m} \cong 0.00 \text{ kNm}.$$

From these two calculations it is concluded that the moment equilibrium requirements are fulfilled.

6.1.2 Equilibrium consideration of a part of the structure

First a cut in x-direction is made, exactly halfway the second row of shear panel elements (Figure 29). This cut is called cut A-A. By performing this cut, not only stringer elements and shear panel elements are cut, but also one strut element (Strut 1). The normal forces and shear forces acting at the "upper" part of the structure, which is considered in the remaining of this section, have to be in equilibrium, since no external load is applied to this part.

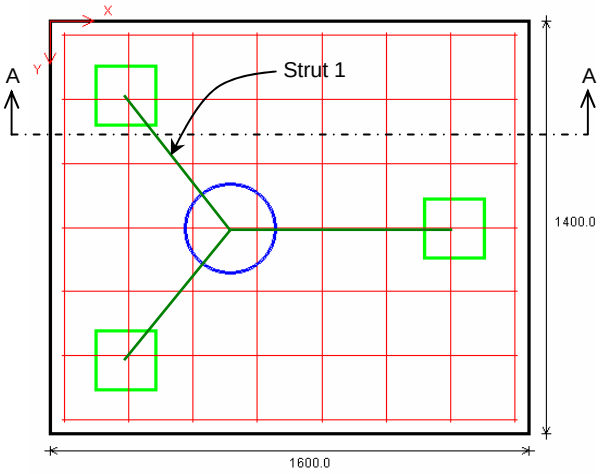


Figure 29 Cut A-A for Case 1

Horizontal force equilibrium in cut A-A

Now consider the normal forces acting at cut A-A (Figure 30). The magnitude of these normal forces in the stringer elements is calculated in Table 5. The values for the reinforcement stresses at the ends of the concerning stringer elements can be read from the 'Advanced' tab.

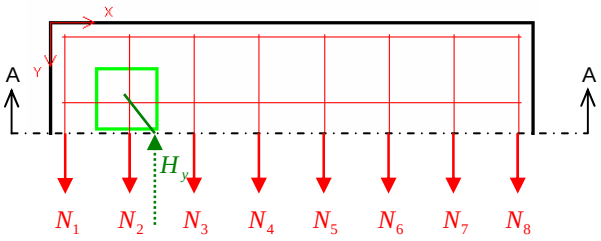


Figure 30 Normal forces in y-direction acting at cut A-A

The sum of these normal forces is equal to

$$\sum_{i=1}^{8} N_i = 28.29 + 25.40 + 12.50 + 3.539 + (-0.9359) + 0.2474 + 5.553 + 2.102 = 76.70 \text{ kN}.$$

For the calculation of H_y, see Figure 32 and Table 6.

Stringer element 51:	$\sigma_1 = 163.3\ \text{N/mm}^2$ $\sigma_2 = 118.1\ \text{N/mm}^2$	$\sigma_{average} = 140.7\ \text{N/mm}^2$	$N_1 = 28.29\ \text{kN}$
Stringer element 57:	$\sigma_1 = 139.1\ \text{N/mm}^2$ $\sigma_2 = 113.6\ \text{N/mm}^2$	$\sigma_{average} = 126.4\ \text{N/mm}^2$	$N_2 = 25.40\ \text{kN}$
Stringer element 63:	$\sigma_1 = 48.18\ \text{N/mm}^2$ $\sigma_2 = 76.14\ \text{N/mm}^2$	$\sigma_{average} = 62.16\ \text{N/mm}^2$	$N_3 = 12.50\ \text{kN}$
Stringer element 69:	$\sigma_1 = 9.031\ \text{N/mm}^2$ $\sigma_2 = 26.17\ \text{N/mm}^2$	$\sigma_{average} = 17.60\ \text{N/mm}^2$	$N_4 = 3.539\ \text{kN}$
Stringer element 75:	$\sigma_1 = -0.8935\ \text{N/mm}^2$ $\sigma_2 = -8.416\ \text{N/mm}^2$	$\sigma_{average} = -4.655\ \text{N/mm}^2$	$N_5 = -0.9359\ \text{kN}$
Stringer element 81:	$\sigma_1 = 10.08\ \text{N/mm}^2$ $\sigma_2 = -7.619\ \text{N/mm}^2$	$\sigma_{average} = 1.231\ \text{N/mm}^2$	$N_6 = 0.2474\ \text{kN}$
Stringer element 87:	$\sigma_1 = 18.19\ \text{N/mm}^2$ $\sigma_2 = 37.05\ \text{N/mm}^2$	$\sigma_{average} = 27.62\ \text{N/mm}^2$	$N_7 = 5.553\ \text{kN}$
Stringer element 93:	$\sigma_1 = -5.519\ \text{N/mm}^2$ $\sigma_2 = 26.43\ \text{N/mm}^2$	$\sigma_{average} = 10.46\ \text{N/mm}^2$	$N_8 = 2.102\ \text{kN}$

Table 5 Normal forces acting at cut A-A

Figure 32 Similarity between force components and geometry components for a strut element

$V =$	-68.18 kN	$\Delta z =$	400 mm
$H =$	-97.17 kN	$\ell_{projected} = \sqrt{(\Delta x)^2 + (\Delta y)^2} =$	570,1 mm
$H_x =$	-59.66 kN	$\Delta x =$	350 mm
$H_y =$	-76.70 kN	$\Delta y =$	450 mm

Table 6 Strut force components and geometric components

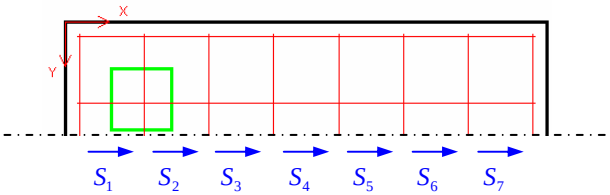


Figure 31 Shear forces acting at cut A-A

Now consider the shear forces acting at cut A-A (Figure 31).

The values for the shear stresses at the edges of the concerning shear panel elements can be read again from the 'Advanced' tab. The length a of a shear panel element is equal to the center-to-center distance of the reinforcing bars in y -direction

$$a = \frac{1600 - 2 \times 40 - 16}{7} = 214.9 \text{ mm.}$$

The effective depth t can be determined from scheme given in Section 3.3. First calculate the width b of a shear panel element

$$b = \frac{1400 - 2 \times 40 - 16}{6} = 217.3 \text{ mm.}$$

The decisive value for the effective depth t is equal to

$$t = 40 + \frac{(16+16)}{4} + \frac{(217.3+214.9)}{4} = 156.1 \text{ mm.}$$

From these data the resulting shear forces can be calculated (Table 7).

Shear panel element 8:	$\tau = 0.2681$ N/mm^2	$S_1 = 8.994$ kN
Shear panel element 9:	$\tau = 0.4191$ N/mm^2	$S_2 = 14.06$ kN
Shear panel element 10:	$\tau = 0.2534$ N/mm^2	$S_3 = 8.501$ kN
Shear panel element 11:	$\tau = 0.1518$ N/mm^2	$S_4 = 5.092$ kN
Shear panel element 12:	$\tau = 0.1964$ N/mm^2	$S_5 = 6.588$ kN
Shear panel element 13:	$\tau = 0.3013$ N/mm^2	$S_6 = 10.11$ kN
Shear panel element 14:	$\tau = 0.1894$ N/mm^2	$S_7 = 6.354$ kN

Table 7 Shear forces acting at cut A-A

The sum of these shear forces is equal to

$$\sum_{i=1}^{7} S_i = 8.994 + 14.06 + 8.501 + 5.092 + 6.588 + 10.11 + 6.354 = 59.70 \text{ kN.}$$

From Table 6 it can be seen that H_x is equal to -59.66 kN. If also the support reaction at DOF 1 is taken into account, the horizontal force equilibrium can be checked

$$\sum H = 59.70 - 59.66 - 6.895 \times 10^{-11} = 0.04000 \text{ kN.}$$

Round-off errors during calculation cause this sum not to be exactly equal to zero.

Horizontal force equilibrium in x -direction in cut B-B

Now a cut in y -direction is made (Figure 34). This cut B-B is made exactly halfway the second column of shear panel elements. Again, this cut delivers normal forces where stringer elements are cut and shear forces where shear panel elements are cut. The left part of the structure is taken into consideration. Since no horizontal external load is applied to this part of the structure, the normal forces and shear forces have to be in equilibrium with each other.

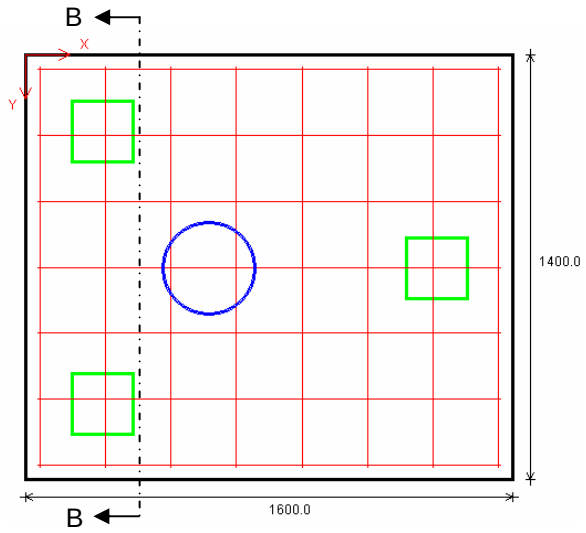


Figure 34 Cut B-B in Case 1

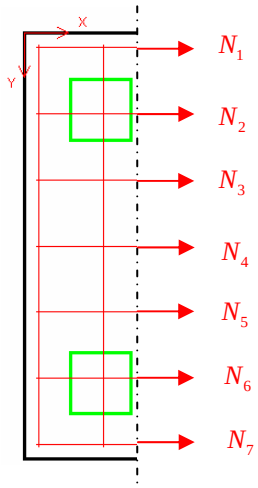


Figure 33 Normal forces acting at cut B-B

Stringer element 2:	$\sigma_1 = 159.8$ N/mm^2 $\sigma_2 = 81.67$ N/mm^2	$\sigma_{average} = 120.7$ N/mm^2	$N_1 = 24.28$ kN
Stringer element 9:	$\sigma_1 = 92.18$ N/mm^2 $\sigma_2 = 100.4$ N/mm^2	$\sigma_{average} = 96.29$ N/mm^2	$N_2 = 19.36$ kN
Stringer element 16:	$\sigma_1 = 33.29$ N/mm^2 $\sigma_2 = 80.84$ N/mm^2	$\sigma_{average} = 57.07$ N/mm^2	$N_3 = 11.47$ kN
Stringer element 23:	$\sigma_1 = 22.83$ N/mm^2 $\sigma_2 = 67.51$ N/mm^2	$\sigma_{average} = 45.17$ N/mm^2	$N_4 = 9.082$ kN
Stringer element 30:	$\sigma_1 = 33.29$ N/mm^2 $\sigma_2 = 80.84$ N/mm^2	$\sigma_{average} = 57.07$ N/mm^2	$N_5 = 11.47$ kN
Stringer element 37:	$\sigma_1 = 92.18$ N/mm^2 $\sigma_2 = 100.4$ N/mm^2	$\sigma_{average} = 96.29$ N/mm^2	$N_6 = 19.36$ kN
Stringer element 44:	$\sigma_1 = 159.8$ N/mm^2 $\sigma_2 = 81.67$ N/mm^2	$\sigma_{average} = 120.7$ N/mm^2	$N_6 = 24.28$ kN

Table 8 Normal forces acting at cut B-B

Consider the normal forces acting perpendicular to cut B-B (Figure 33). The stresses in the stringer elements can be read from the 'Advanced' tab. The resulting normal forces are calculated in Table 8. From this table the symmetry is clear again. The sum of the calculated normal forces is equal to

$$\sum_{i=1}^{7} N_i = 24.28 + 19.36 + 11.47 + 9.082 + 11.47 + 19.36 + 24.28 = 119.3 \text{ kN}.$$

This sum of normal forces has to be in equilibrium with the horizontal force components of the cut strut elements and the horizontal support reactions in x-direction. The horizontal force component of the cut strut elements can be read from Table 6. The horizontal support reactions in x-direction can be read from Table 2. The horizontal equilibrium then reads

$$\sum H_x = 119.3 + 2\times(-59.66) - 6.895\cdot10^{-11} - 5.190\cdot10^{-11} \cong 0.00 \text{ kN},$$

which is correct.

Horizontal force equilibrium in y -direction in cut B-B

Again, the values for the shear stresses at the edges of the concerning shear panel elements can be read from the 'Advanced' tab. Notice that this time the width b of a shear panel element has to be used, in stead of the length a of a shear panel element. The shear forces have been calculated in Table 9.

Shear panel element 2:	$\tau = 0.4687$ N/mm^2	$S_1 = 15.90$ kN
Shear panel element 9:	$\tau = 0.4191$ N/mm^2	$S_2 = 14.22$ kN
Shear panel element 16:	$\tau = 0.1340$ N/mm^2	$S_3 = 4.545$ kN
Shear panel element 23:	$\tau = -0.1340$ N/mm^2	$S_4 = -4.545$ kN
Shear panel element 30:	$\tau = -0.4191$ N/mm^2	$S_5 = -14.22$ kN
Shear panel element 37:	$\tau = -0.4687$ N/mm^2	$S_6 = -15.90$ kN

Table 9 Shear forces acting at cut B-B

Again, the symmetry can be noticed. The horizontal force components originating from the struts point in opposite direction. The magnitude of these forces can be read from Table 6. The horizontal support reactions in y -direction can be found in Table 4.
The resultant force in y -direction is

$$\sum H_y = 15.90 + 14.22 + 4.545 + (-4.545) + (-14.22) + (-15.90) + (-76.70) - (-76.70) + 2,604\cdot10^{-11}$$
$$\cong 0.00 \text{ kN}.$$

6.2 Case 2: Asymmetrical pile cap consisting of six piles and two columns

Pile cap parameters

The input parameters which have been used for the calculation are given in Table 11. A screenshot of the pile cap plan view is provided in Figure 35.

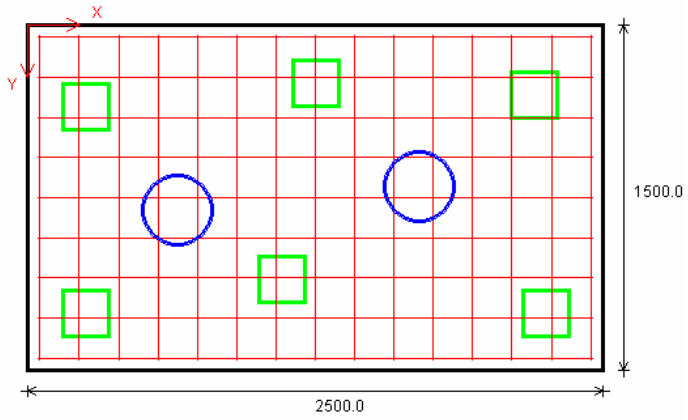


Figure 35 Screenshot of the pile cap geometry in Case 2

	Cap configuration		
Cap length (in x-direction):	2500 mm		
Cap width (in y-direction):	1500 mm		
Cap depth (in z-direction):	450 mm		
Young's modulus of cap concrete:	9500 N/mm^2		
Shear modulus of cap concrete:	4750 N/mm^2		
	Column configuration		
Number of columns:	2		
Column 1:	$x = 650$ mm	$y = 800$ mm	$N = 215$ kN
Column 2:	$x = 1700$ mm	$y = 700$ mm	$N = 180$ kN
	Pile configuration		
Number of piles:	6		
Pile 1:	$x = 250$ mm	$y = 350$ mm	
Pile 2:	$x = 1250$ mm	$y = 250$ mm	
Pile 3:	$x = 2200$ mm	$y = 300$ mm	
Pile 4:	$x = 250$ mm	$y = 1250$ mm	
Pile 5:	$x = 1100$ mm	$y = 1100$ mm	
Pile 6:	$x = 2250$ mm	$y = 1250$ mm	
Pile length:	17000 mm		
Pile section:	40000 mm^2		
Young's modulus of pile concrete:	12000 N/mm^2		
	Reinforcing bar configuration		
Concrete cover:	45 mm		
Diameter of reinforcing bars in x-direction:	16 mm		
Number of reinforcing bars in x-direction:	9		
Diameter of reinforcing bars in y-direction:	16 mm		
Number of reinforcing bars in y-direction:	15		
Young's modulus of reinforcing bars:	200000 N/mm^2		

Table 11 Pile cap parameters for Case 2

Vertical equilibrium of forces

The vertical support reactions of the piles can be read from the 'Results' tab after performing the calculation. These values are collected in Table 10. The minus sign indicates that the piles are in compression. It is clear that the sum of the vertical pile reactions in equilibrium is with the sum of the column loads: $215+180 = 395$ kN.

Pile 1:	-65.91 kN
Pile 2:	-75.76 kN
Pile 3:	-50.48 kN
Pile 4:	-67.73 kN
Pile 5:	-84.72 kN
Pile 6:	-50.39 kN +
Total:	-395.0

Table 10 Vertical pile reactions

Horizontal equilibrium of forces

The horizontal support reactions in x-direction can be read from the 'Advanced' tab after performing the calculation. These values are collected in Table 12. Since no horizontal load is applied in x-direction, the sum of the horizontal support reactions in x-direction has to be zero, which is true accepting a round-off error. Furthermore, the stresses at the ends of the stringer elements near the edges of the pile cap have to be equal to zero. This follows from the equilibrium of such a stringer element end. By plotting the reinforcement

stresses in x-direction, it can be verified that Case 2 complies with this requirement (Figure 36).
The horizontal support reactions in y-direction can also be read from the 'Advanced' tab. These values have been collected in Table 12. Since no horizontal load is applied in this direction, the sum of the horizontal support reactions in y-direction has to be zero, which is true accepting a round-off error. Furthermore, the reinforcement stresses at the ends of the stringer elements near the edges of the pile cap have to be equal to zero again. By plotting the normal stresses in y-direction, it can be verified that Case 2 complies with this requirement (Figure 37).

Support reaction 1:	$1.153 \cdot 10^{-8}$ kN
Support reaction 3:	$-1.003 \cdot 10^{-8}$ kN
Support reaction 4:	0.0 kN
Support reaction 7:	0.0 kN
Support reaction 10:	0.0 kN
Support reaction 13:	0.0 kN
Support reaction 16:	0.0 kN
Support reaction 19:	0.0 kN

Table 12 Horizontal support reactions (in x-direction)

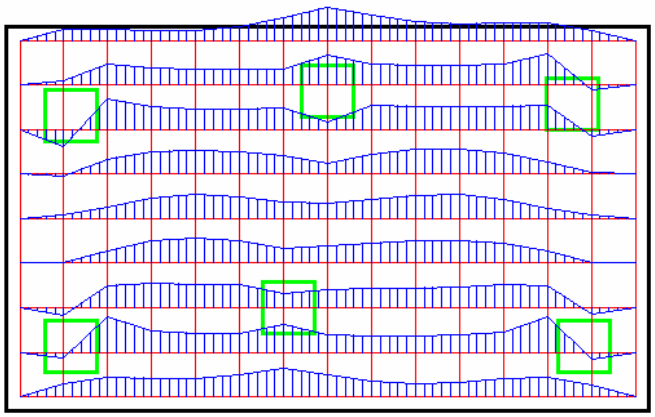


Figure 36 Reinforcement stresses in x-direction

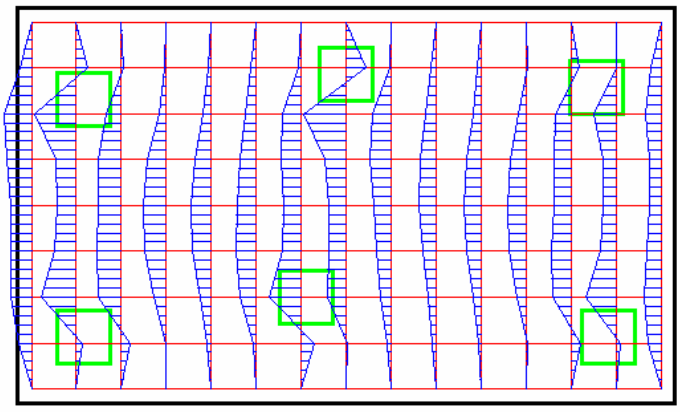


Figure 37 Reinforcement stresses in y-direction

Considering these two cases, it may be concluded that all equilibrium requirement are fulfilled. From the theory of plasticity (Vrouwenvelder, 2003) [11] it is known that a system which is in equilibrium gives a safe approximation of the ultimate load.

One more interesting conclusion can be drawn by looking at the reinforcement stresses. Often the reinforcing bars at the edges of pile caps have higher stresses than other reinforcing bars.

7 Non-linear finite element analysis

This chapter presents the second part of the verification process, which entails a comparison of the ultimate load predicted by the applet with the ultimate load predicted by the finite element package Atena 3D [12]. Section 7.1 gives the pile cap geometry and the material parameters used in this case study. Section 7.2 reveals the ultimate load according to the applet. In Section 0 the same pile cap is tested with Atena 3D. In this section the results of both analyses are compared and conclusions are drawn.

7.1 Geometry of the considered pile cap and material parameters

The considered (asymmetric) pile cap consists of three piles and one column (Figure 38). The cap length is 2200 mm, the cap width is 2000 mm and the cap depth is 500 mm. The strength class of the cap concrete is chosen as C30/37. For an analysis with the applet, the dimensions of the column are of no importance, since the column load is represented as a point load, which is applied to a certain point on the concrete surface. But for a finite element analysis, applying a point load directly onto the concrete surface is not sensible, since this introduces very large if not infinitely large stresses directly under this point load. Therefore, it is decided to use a thick steel plate for the finite element analysis to simulate the normal force transfer of the column in a better way. On top of this steel plate a point load is applied. This gives a disturbed picture of the stresses inside the steel plate. But since these stresses are of no importance for the analysis, this is good way of applying the column load. The steel plate has sides of 500 mm by 500 mm. The foundation piles are chosen to be 16 m long and they are made of C50/60 concrete. The reinforcement in x-direction as well as in y-direction consists of reinforcing bars with a diameter of 16 mm. The concrete cover is chosen as 45 mm. In x-direction eleven reinforcing bars are applied, which means that the center-to-center distance of these reinforcing bars is equal to 189.4 mm. In y-direction twelve reinforcing bars are applied, which means that the center-to-center distance of these reinforcing bars is equal to 190.4 mm. The strength class of the reinforcement is FeB500. The positions of

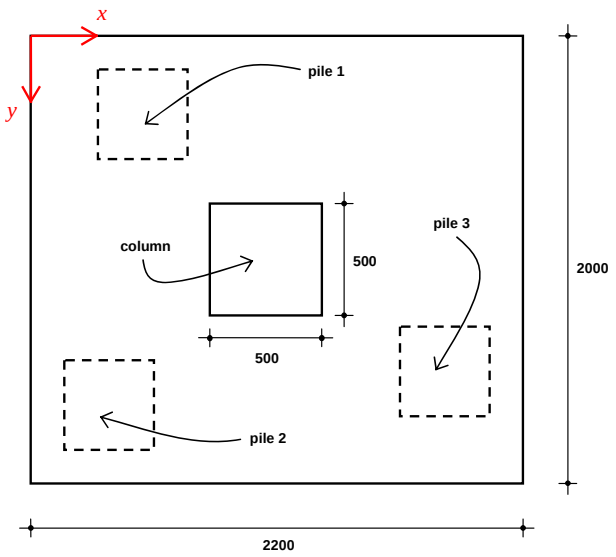


Figure 38 Plan view of the considered pile cap

the foundation piles have been chosen in an irregular pattern with a minimum edge distance of 150 mm. All parameters that are needed to perform the structural analysis can be found in Table 13.

	Cap configuration
Cap length (in x-direction):	2200 mm
Cap width (in y-direction):	2000 mm
Cap depth (in z-direction):	500 mm
Cap concrete:	C30/37
	Column configuration
Number of columns:	1
Column 1:	$x = 1050$ mm $y = 1000$ mm
	Pile configuration
Number of piles:	3
Pile 1:	$x = 500$ mm $y = 350$ mm
Pile 2:	$x = 350$ mm $y = 1650$ mm
Pile 3:	$x = 1850$ mm $y = 1500$ mm
Pile length:	16000 mm
Pile section:	$400 \times 400 = 160000$ mm^2
Pile concrete:	C50/60
	Reinforcement configuration
Concrete cover:	45 mm
Diameter of reinforcing bars in x-direction:	16 mm
Number of reinforcing bars in x-direction:	11
Diameter of reinforcing bars in y-direction:	16 mm
Number of reinforcing bars in y-direction:	12
Young's modulus of rebar:	$2{,}00 \cdot 10^5$ N/mm^2

Table 13 Pile cap parameters for the considered case

7.2 Ultimate load predicted by Pile Cap Applet (PCA)

The ultimate load is calculated by assuming that failure of the pile cap occurs when the yield stress is reached in one of the reinforcing bars. The material parameters that have not been specified explicitly in Section 7.1, but which are needed for applet calculation, have been collected in Table 14. The values for $E_{cap\ concrete}$ and $E_{pile\ concrete}$ have been taken from Table 3.1 in Eurocode 2 (EN 1992-1-1:2004). These values are also used by Atena 3D. The value

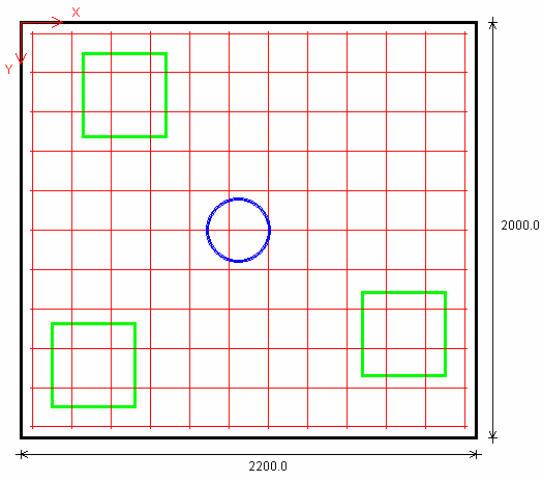


Figure 39 Screenshot of pile cap plan view

for ν has been taken from Atena 3D. A screenshot of the pile cap plan view is given in Figure 39.

After several iterations it is found that a load of 404 kN produces a maximum reinforcement stress in x-direction of 434.1 N/mm^2. A load of 405 kN produces a maximum normal stress in x-direction of 435.2 N/mm^2, which is larger than the design yield strength. Therefore, it is decided that 404 kN is the ultimate load according to Pile Cap Applet (PCA). At this load the pile reactions are -182.35 kN for Pile 1, -51.35 kN for Pile 2 and -170.3 kN for Pile 3. The reinforcement stresses σ_{xx} at this load level are given in Figure 40. Figure 41 gives the reinforcement stresses σ_{yy} for the same load level.

To make sure that punching of the column or one of the piles is not the decisive failure mechanism, an extra check is performed using Section 6.4 'Punching' of Eurocode 2 (EN 1992-1-1:2004).

$E_{\text{cap concrete}}$	=	33000 N/mm^2
ν	=	0.2
$G_{\text{cap concrete}}$	=	13750 N/mm^2
$E_{\text{pile concrete}}$	=	37000 N/mm^2

Table 14 Material parameters for applet calculation

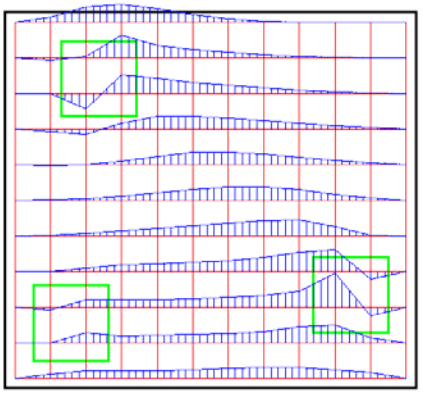


Figure 40 Reinforcement stresses σ_{xx} for the considered pile cap at a load level of 404 kN

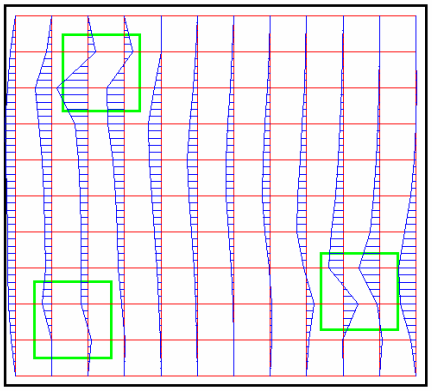


Figure 41 Normal stresses σ_{yy} for the considered pile cap at a load level of 404 kN

Punching of the column

Punch is checked by using the Dutch code NEN 6720. Since no shear reinforcement is applied, only the concrete contributes to the shear resistance

$$\tau_1 = 0.8 f_b k_d \sqrt[3]{\omega_0} > 0.8 f_b ,$$

in which f_b represents the concrete tensile strength. For convenience, the lower bound is used first, which means that τ_1 is calculated from $0.8 f_b$. f_b can be determined from the characteristic cubic compression strength $f_{ck}^{'}$

$$f_{b,rep} = 0.7\left(1.05 + 0.05 f_{ck}^{'}\right) = 0.7\left(1.05 + 0.05 \times 30\right) = 1.785 \text{ N/mm}^2 ,$$

$$f_b = \frac{f_{b,rep}}{1.4} = \frac{1.785}{1.4} = 1.275 \text{ N/mm}^2 .$$

Now τ_1 can be calculated

$$\tau_1 = 0.8 \times 1.275 = 1.02 \text{ N/mm}^2 .$$

The effective thickness of the pile cap can be calculated from

$d = 0.5 \times (d_1 + d_2)$.

$$d_1 = 500 - 45 - 16 - \frac{16}{2} = 431 \text{ mm},$$

$$d_2 = 500 - 45 - \frac{16}{2} = 447 \text{ mm},$$

$$d = 0.5(431 + 447) = 439 \text{ mm},$$

$$a = \frac{2}{\pi}(500 + 500) = 636.6 \text{ mm},$$

$$p = \pi(d + a) = \pi(439 + 636.6) = 3379 \text{ mm},$$

$$\tau_d = \frac{F}{pd} = \frac{404 \times 10^3}{3379 \times 439} = 0.27 \text{ N/mm}^2 < \tau_1 = 1.02 \text{ N/mm}^2.$$

From the above calculations it may be concluded that punching of the column is not decisive.

7.3 Ultimate load predicted by non-linear finite element analysis

For material input, the appropriate concrete strength classes can be selected from the in-built catalogue. In this case design values have been used. The reinforcement is specified via a direct definition. The material type is 'reinforcement', which has a bilinear stress-strain relation, with a Young's modulus of $2{,}00 \cdot 10^5$ N/mm² and a design yield strength of 435 N/mm².

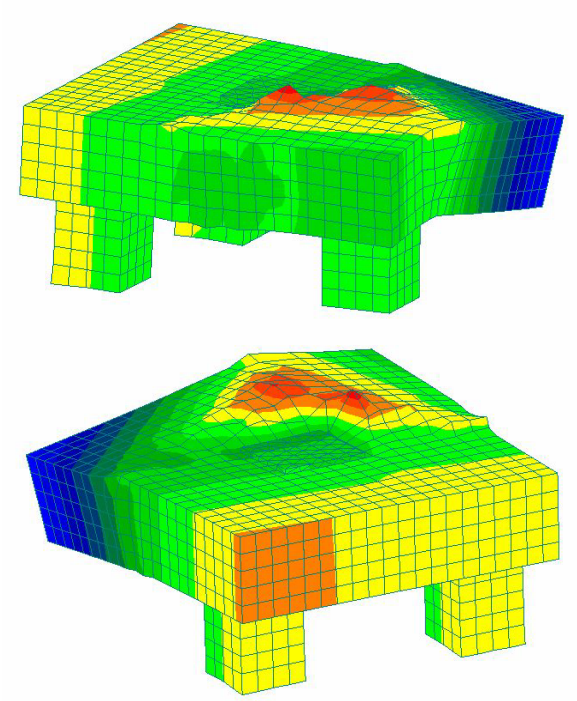


Figure 42 Vertical displacements of the pile cap after failure

From Figure 42 it can be concluded that the pile cap collapsed because of a shear failure around pile 3. This is a completely different failure mechanism than PCA predicted.

Figure 43 shows the load-displacement graph obtained with Atena 3D in blue and the ultimate load obtained with the applet in red. It is surprising that the ultimate load obtained with Atena 3D is almost a factor 3 larger than the ultimate load obtained with PCA. From this it can be concluded that PCA is very conservative.

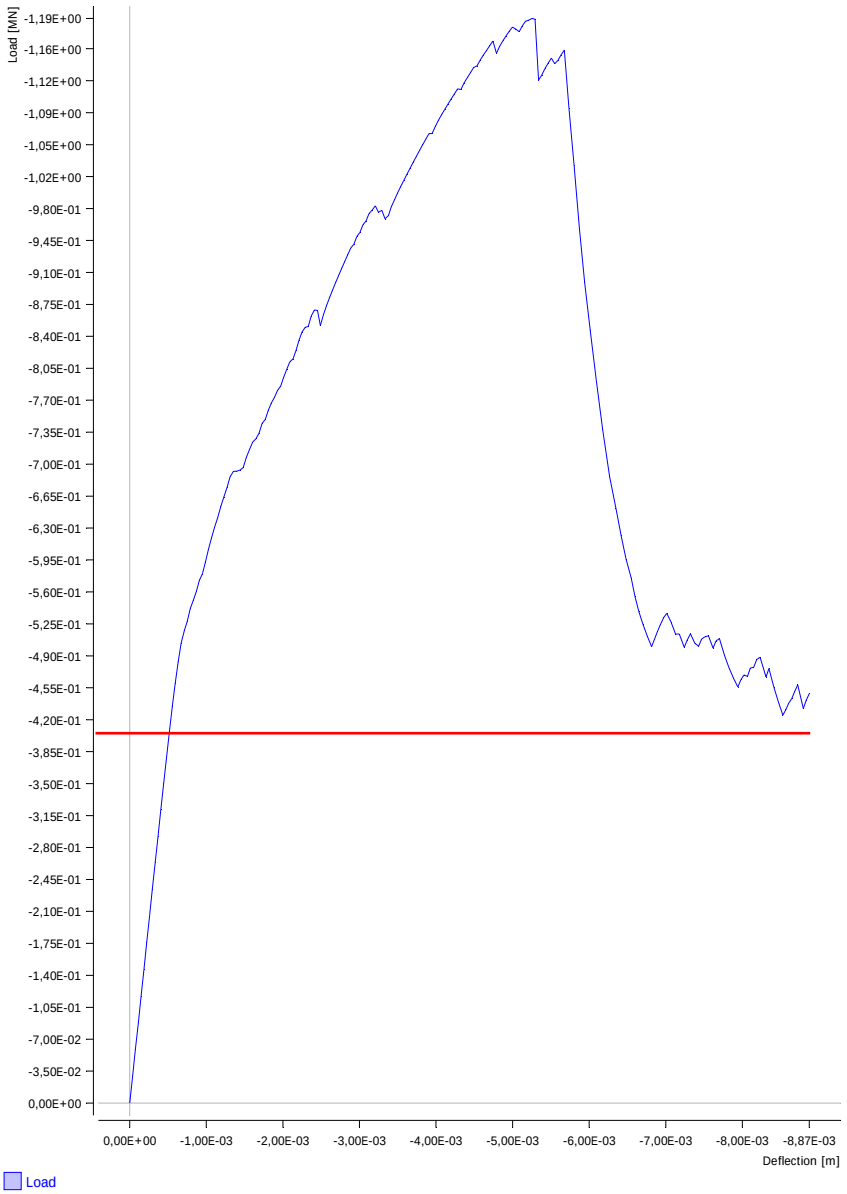


Figure 43 Load-displacement graph according to Atena 3D (in blue) and the ultimate load according to PCA (in red)

Figure 45 shows the reinforcement stresses at a load level of approximately 400 kN. According to PCA in one of the reinforcing bars the yield stress is reached. But the finite element calculation shows that the maximum stress is not larger than 8.66 N/mm^2, which is a factor 50 smaller. Therefore, it can be concluded that reinforcement stresses at serviceability load according to PCA are much higher than those determined by the finite element analysis. This implies that the stresses calculated by PCA are not useful for checking the maximum crack width.

Another point of interest is that the reinforcements stresses near the edge of the pile cap are almost equal to zero (Figure 45 and Figure 44). This implies that hooks may not be necessary for all reinforcing bars.

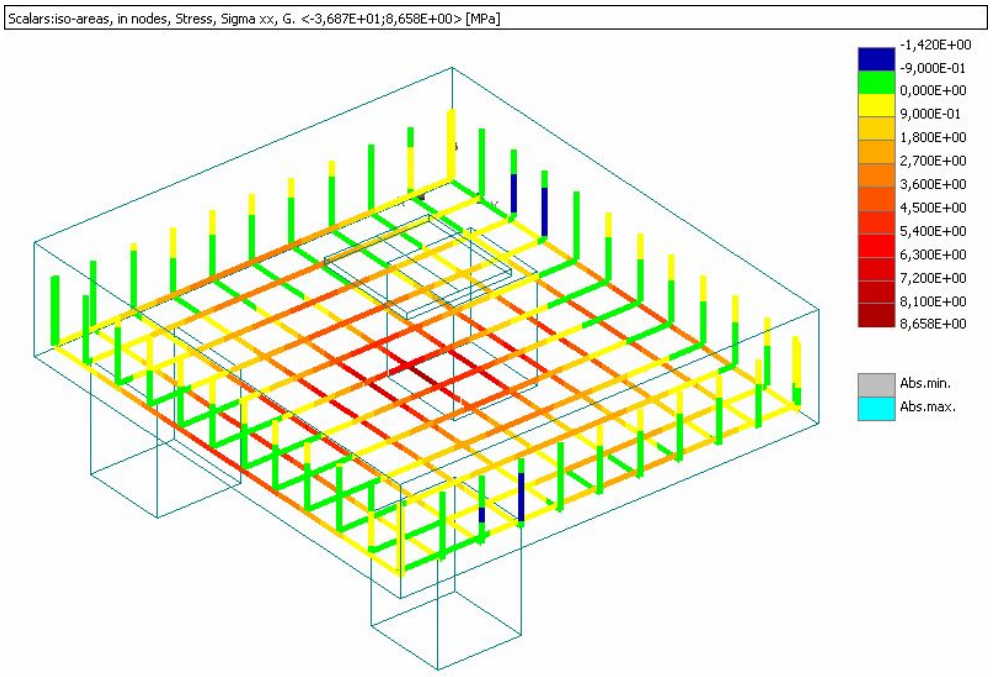


Figure 45 Reinforcement stresses at a load level of approximately 400 kN (iso-areas)

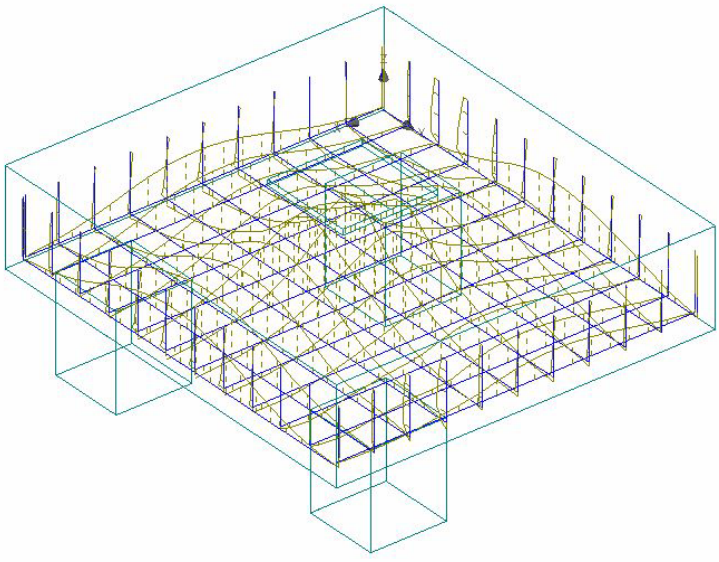


Figure 44 Reinforcement stresses at a load level of approximately 400 kN (diagrams)

To make sure that the serviceability limit state is not normative in the case of PCA, the maximum crack width at this load level had to be checked. Since PCA does not give

information on crack widths, this information is gathered from the non-linear finite element analysis. From the previous section it is known that PCA predicted an ultimate load of 404 kN. In practice load factors on dead weight and variable load are applied. Since pile caps usually bear a lot of dead weight, the overall load factor is estimated at 1.35. This means that the maximum crack width at load level $404/1.35 \cong 300$ kN has to be considered. According to Atena, the maximum crack width is then approximately equal to 1.295×10^{-4} mm, which is far below the limits given in the codes. Based on this single value it can be said that the serviceability limit state probably is not normative when pile caps are designed using PCA.

Also the serviceability limit state in Atena has to be checked. The ultimate load was determined to be 1.19 MN. Considering a load factor of 1.35, this means a service load of approximately $\left(1.19 \times 10^3\right)/1.35 \cong 880$ kN. At this load level the maximum crack width according to Atena is equal to 0.8909 mm which is far more than allowed in codes. An impression of the crack pattern at this stage is given in Figure 46.

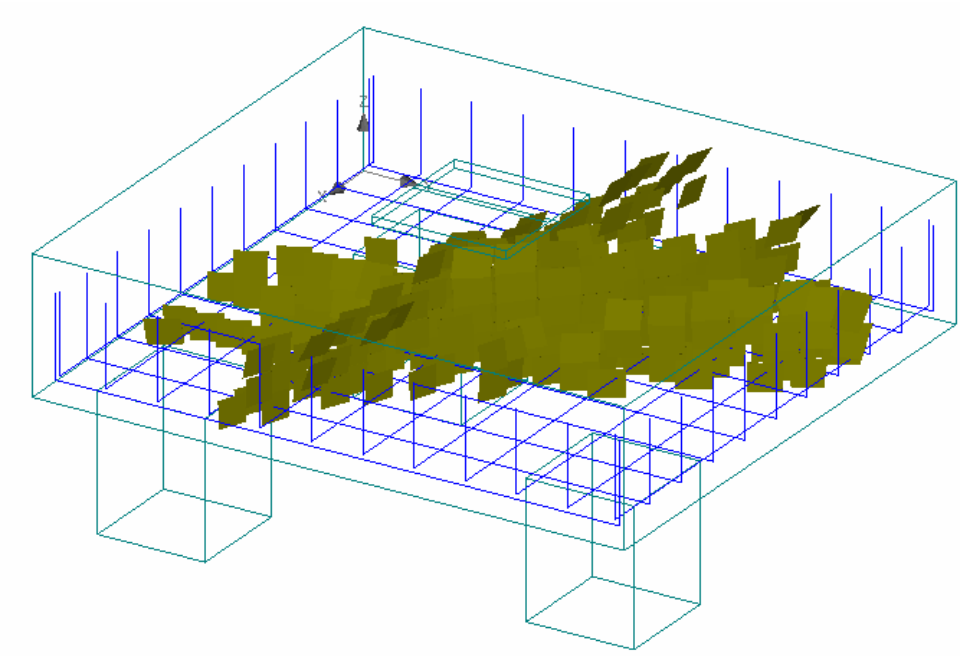


Figure 46 Crack pattern at a load level of approximately 880 kN

Now consider the pile cap at a load level of 1.19 MN, which is the ultimate load according to Atena 3D. The stresses in the reinforcing bars are now considerably higher (Figure 47). The largest tensile stress is 381 N/mm², which is still below the yield strength of 435 N/mm². By considering these reinforcement stresses, it may be concluded that the reinforcement stresses at ultimate load are approximately the same for PCA (404 kN ultimate load) and finite element analysis (1.19 MN ultimate load)

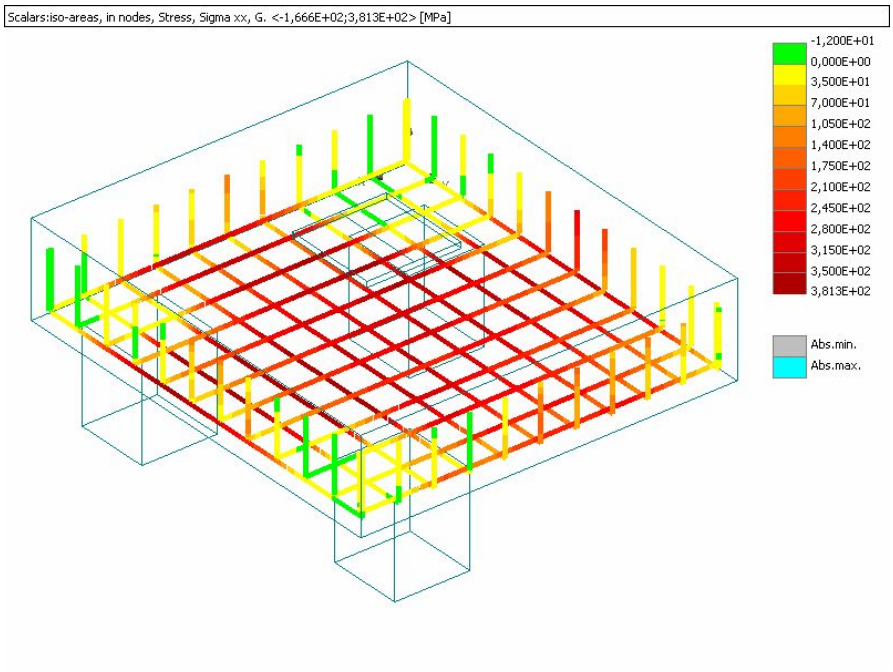


Figure 47 Reinforcement stresses at a load level of approximately 1.19 MN

Now consider the normal strain ε_{xx} in the reinforcing bars after failure (Figure 48). It is interesting to notice that only in a few areas plastic strains occurs, i.e. strains larger than 2.175×10^{-3}.

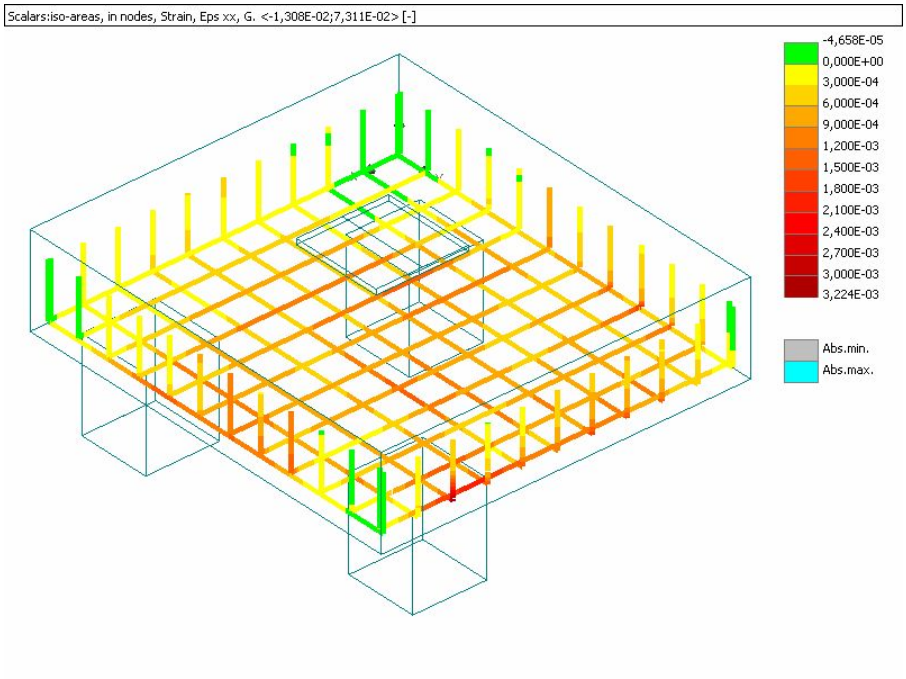


Figure 48 Normal strain in the reinforcing bars at the last load step

One more check can be performed and this is the support reactions of the piles. The piles have been modeled as short concrete blocks under the pile cap. To simulate the extensional behavior of the piles, these concrete blocks are supported by surface springs. The stresses in these surface springs at a load level of approximately 404 kN are shown in Figure 49. By estimating the average stress in the surface springs per pile and multiplying this stress by the pile section, an estimation for the pile reaction is found

Pile 1: $1.15 \times 400 \times 400 \times 10^{-3} = 184$ kN (182.35 kN was found earlier),

Pile 2: $0.35 \times 400 \times 400 \times 10^{-3} = 56$ kN (51.35 kN was found earlier),

Pile 3: $1.10 \times 400 \times 400 \times 10^{-3} = 176$ kN (170.3 kN was found earlier).

From this it may be concluded that the vertical pile reactions at a load level of 404 kN predicted by PCA are approximately equal to those predicted by the non-linear finite element analysis. This result is not trivial because the finite element model was externally statically indeterminate.

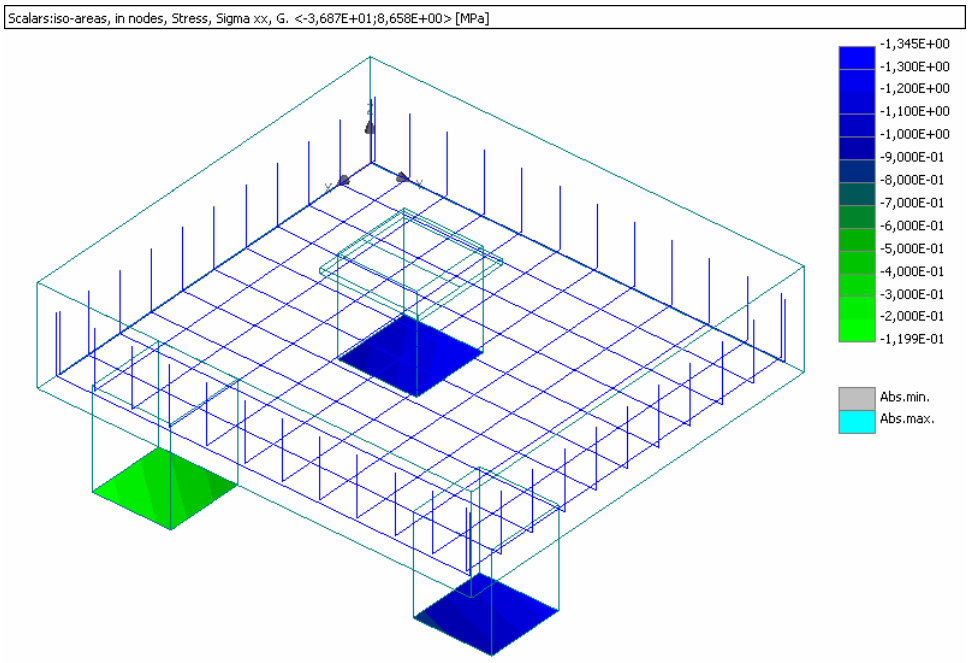


Figure 49 Stresses in the surface springs under the piles

8 Conclusions and recommendations

Conclusions

- A model for the design of reinforced concrete pile caps on irregularly positioned foundation piles has been established based on the use of stringer elements, shear panel elements and strut elements. This model predicts vertical pile reactions, reinforcement stresses and shear stresses in concrete. For practical application, it has been implemented in a computer program called Pile Cap Applet (PCA). This applet is user-friendly, requires a moderate amount of data and takes only a few seconds to execute.

- The design model meets all equilibrium requirements. This has been tested for two specific pile caps: one symmetrical pile cap which was founded on three piles and loaded by one column, and one asymmetrical pile cap which was founded on six piles and loaded by two columns. In the case of the symmetrical pile cap, the results also correctly showed to be symmetrical.

- Comparison of the ultimate load predicted by PCA with a non-linear finite element analysis showed that the ultimate load predicted by the design model is very conservative. In this specific case, an asymmetrical pile cap consisting of three piles and one column was tested. PCA determined that this specific pile cap collapsed at a load level of 404 kN, while the same pile cap analyzed with a non-linear finite element package showed that 1.19 MN was the ultimate load. Clearly, the real structure can carry the load in more ways than an equilibrium system (PCA) assumes.

- For the considered pile cap the design model predicted another failure mechanism than the finite element analysis. In the case of PCA, the pile cap 'collapsed' because the yield strength was reached in one of the reinforcing bars. In the finite element analysis, the pile cap collapsed because of a shear failure. This failure mechanism cannot be predicted by PCA.

- The vertical pile reactions at a load level of 404 kN predicted by PCA are approximately equal to those predicted by the non-linear finite element analysis. This result is not trivial because the finite element model was externally statically indeterminate.

- The reinforcement stresses at serviceability load according to PCA are much higher than those determined by the finite element analysis. This implies that the stresses calculated by PCA are not useful for checking the maximum crack width.

- The reinforcement stresses at ultimate load are approximately the same for PCA (404 kN ultimate load) and finite element analysis (1.19 MN ultimate load).

- Often the reinforcing bars at the edges of pile caps have higher stresses than other reinforcing bars.

Recommendations

- As a future improvement, it is recommended to replace cracked shear panel elements by diagonal elements. However, the consequence of this is that the structural analysis has to be performed several times and the analysis time increases.

- Alternatively, all shear panel elements can be replaced by diagonal strut elements. In a number of analyses the correct (compression) direction of the strut elements need to be determined. This might reduce peak stresses in the reinforcing bars. It would also show the need for hooks at the reinforcing bar ends and the number of DOFs would be reduced.

- Since the design model does not predict the correct failure mechanism, a more refined model for the design problem may be considered. For example, a design model based on volume elements instead of stringer elements, shear panel elements and strut elements. This model would be similar to a three dimensional finite element model. In the near future (say within five years), this model probably can be executed in a few seconds too.

- For rational calculation of crack widths a better model needs to be developed.

References

Literature

[1] Blaauwendraad, J. and P.C.J. Hoogenboom, 'Stringer Panel Model for Structural Concrete Design', *ACI Structural Journal*, Vol. 93, No. 3 (1996), pp. 295-305.

[2] Blaauwendraad, J. and P.C.J. Hoogenboom, 'Stringer Panel Model for Structural Concrete Design', *ACI Structural Journal*, Vol. 94, No. 3 (1997), pp. 336-338.

[3] Blaauwendraad, J., *Eindige-ElementenMethode voor Staafconstructies* (Dutch). 2nd edition. Schoonhoven: Academic Service, 2000.

[4] Blaauwendraad, J., *Plates and Slabs, Volume 2, Numerical Methods*. Delft: 2004.

[5] Hartsuijker, C., and C. F. Vrijman, *Mechanica van constructies 2a, statisch onbepaalde constructies* (Dutch). Delft: 2000.

[6] Hoogenboom, P.C.J., *Het staaf-paneel-model* (Dutch). Delft: 1993.

[7] Lay, D.C., *Linear algebra and its applications*. 2nd edition. Addison Wesley Longman, Inc., 2000

[8] Nijenhuis, W., *De verplaatsingsmethode, toegepast voor de berekening van (staaf)constructies* (Dutch). Amsterdam/Brussel: Agon Elsevier, 1973.

[9] Press, W.H., et al, *Numerical Recipes in C, The Art of Scientific Computing*. Cambridge: Cambridge University Press, 1988.

[10] Schlaich, J., K. Schäfer and M. Jennewein, 'Toward a Consistent Design of Structural Concrete'. *PCI Journal, Special Report*, Vol. 32, No. 3 (1987).

[11] Vrouwenvelder, A.C.W.M., *Plasticity, The plastic behaviour and the calculation of beams and frames subjected to bending*. Delft: 2003.

Software

[12] *Atena 3D*, Version: Exe 1.4.0.21, Pre: 1.4.0.29, Run: 1.4.0.20, Post: 1.4.0.16. Cervenka Consulting.

[13] *Borland Delphi Enterprise*, Version 7.0 (Build 4.453). Borland Software Corporation.

[14] *Java ™ Platform, Standard Edition 6*, Version: 1.6.0-rc (build 1.6.0-rc-b104). Sun Microsystems, Inc.

Websites

[15] http://java.sun.com/applets/

[16] http://java.sun.com/docs/books/tutorial/

[17] http://java.sun.com/docs/books/tutorial/java/concepts/index.html

[18] http://www.cs.princeton.edu/introcs/95linear/Matrix.java.html

[19] http://www.math.gatech.edu/~bourbaki/math2601/Web-notes/2num.pdf

Appendix A1: Numbering and generating stringer elements

This appendix discusses numbering and generating stringer elements in the applet. First, it is explained how the degrees of freedom (DOF) are numbered. Then the source code of the procedure `generateStringers ()` is given.

Numbering stringer elements and their DOF

First, stringer elements in x-direction are numbered. Obviously, the number of stringer elements that fits in x-direction is equal to the number of shear panel elements that fits in x-direction, which is called n_x. The total number of stringer elements in x-direction is equal to the product of n_x and the number of reinforcing bars in x-direction. Numbering stringer elements is explained on the basis of Figure 50. The reinforcing bars are drawn in red, the stringer elements are drawn as grey bar elements. Numbering starts from the origin, which has been indicated in blue. The horizontal axis is the x-axis, the vertical axis is the y-axis. Continue numbering from left to right and from top to bottom.

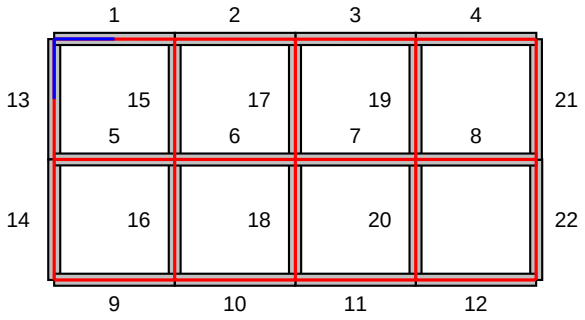


Figure 50 Numbering stringer elements

Then, stringer elements in y-direction are numbered. Obviously, numbering has to continue from the last stringer element in x-direction. The number of stringer elements that fits in y-direction is equal to the number of shear panel elements in y-direction, which is called n_y. The total number of stringer elements in y-direction can be calculated as the product of n_y and the number of reinforcing bars in y-direction.

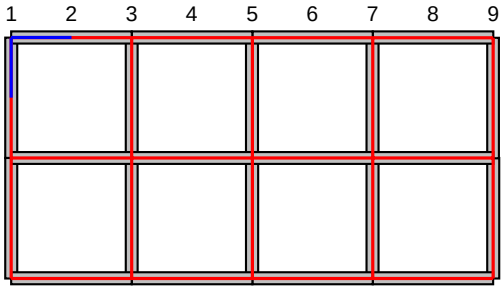


Figure 51 Numbering stringer element DOF for the first row

While stringers elements are numbered, also their DOF have to be numbered (Figure 51). From Section 3.2 it is known that a stringer element is a 3 DOF element. Moreover, the DOF at the ends of an element coincide with those of another stringer element, since they are mutually connected (except for elements ending at the pile cap edge). For stringer elements in x-direction the global number of the intermediate DOF is first determined, based on stringer element row and column. The global numbers of the two remaining DOF can be calculated by subtracting

respectively adding 1 to this number. For the stringer elements in y-direction, first the last assigned DOF number has to be calculated. Then numbering can continue, again based on the stringer element row and column.

Generating stringer elements

The global DOF numbers per stringer element are stored in a matrix called `stringerDOF`. Each row corresponds to one stringer element and has three columns. The first column holds the global DOF numbers per stringer element corresponding to local DOF number 1. Similarly, columns 2 and 3 hold the global DOF numbers per stringer element corresponding to local DOF numbers 2 respectively 3. While the DOF numbers are assigned, also the extensional stiffness EA and the stringer element length ℓ are determined and stored in vectors: `stringerEA` respectively `stringerLength`. The source code of the procedure `generateStringers ()` reads:

```
private void generateStringers ()
{

    // variable declaration

    int nx = nrOfRebarsY - 1;  // number of shear panels in X-direction
    int ny = nrOfRebarsX - 1;  // number of shear panels in Y-direction


    // determine the number of stringers used in the model

    nrOfStringers = nx * nrOfRebarsX + ny * nrOfRebarsY;


    // create a matrix which stores the DOF per stringer

    stringerDOF = new IntMatrix (nrOfStringers, 3);


    // create a vector which stores the extensional stiffness per stringer
    // and one for the length per stringer

    stringerEA = new DoubleVector (nrOfStringers);
    stringerLength = new DoubleVector (nrOfStringers);


    // calculate the extensional stiffness for the stringers in X-direction

    double stringerEA_X = E_Rebars * Math.PI / 4 * rebarDiameterX *
                          rebarDiameterX;


    // number the DOF of the stringers in X-direction

    for (int j = 1; j <= nrOfRebarsX; j++)
    {

        for (int i = 1; i <= nx; i++)
        {

            // determine the stringer number, starting from 0

            int stringerNr = (j - 1) * nx + i - 1;


            // determine the number of the middle DOF of the stringer

            int n = (j - 1) * (2 * nx + 1) + 2 * i;
```

```
                // set the entries of the stringerDOF matrix, the stringerEA
                // vector and the stringerLength vector

                stringerDOF.setEntry (stringerNr, 0, (n - 1));
                stringerDOF.setEntry (stringerNr, 1, n);
                stringerDOF.setEntry (stringerNr, 2, (n + 1));
                stringerEA.setEntry (stringerNr, stringerEA_X);
                stringerLength.setEntry (stringerNr, ctcDistanceOfRebarsY);

            }

        }


        // calculate the extensional stiffness for the stringers in Y-direction

        double stringerEA_Y = E_Rebars * Math.PI / 4 * rebarDiameterY *
                              rebarDiameterY;


        // number the DOF of the stringers in Y-direction

        for (int i = 1; i <= nrOfRebarsY; i++)
        {

            for (int j = 1; j <= ny; j++)
            {

                // determine the stringer number, starting from
                // the last number assigned in X-direction

                int stringerNr = nx * nrOfRebarsX + (i - 1) * ny + j - 1;


                // determine the number of the middle DOF of the stringer

                int n = (2 * nx + 1) * nrOfRebarsX + (i - 1) * (2 * ny + 1) + 2
                        * j;


                // set the entries of the stringerDOF matrix, the stringerEA
                // vector and the stringerLength vector

                stringerDOF.setEntry (stringerNr, 0, (n - 1));
                stringerDOF.setEntry (stringerNr, 1, n);
                stringerDOF.setEntry (stringerNr, 2, (n + 1));
                stringerEA.setEntry (stringerNr, stringerEA_Y);
                stringerLength.setEntry (stringerNr, ctcDistanceOfRebarsX);

            }

        }

    }
```

Appendix A2: Numbering and generating shear panel elements

This appendix discusses numbering and generating shear panel elements. First, it is explained how the degrees of freedom (DOF) are numbered. Then the source code of the procedure `generateShearPanels ()` is given.

Numbering shear panel elements and their DOF

Since the stringer elements and their DOF already have been numbered (Appendix A1), no "new" DOF numbers need to be assigned. It is sufficient to find the correct intermediate DOF numbers of the adjacent stringer elements, because these DOF coincide (Section 3.3). Numbering shear panel elements is analogous to numbering stringer elements (Appendix A1): start from the shear panel element closest to the origin, from left to right and from top to bottom. This is demonstrated in Figure 53. The origin is indicated in blue. The horizontal axis is the x-axis, the vertical axis is the y-axis. While the shear panel elements are numbered, their DOF have to be numbered as well (Figure 52). The number of shear panel elements that fits in x-direction is called n_x, the number of shear panel elements that fits in y-direction is called n_y. These numbers are determined by the number of reinforcing bars specified by the user. The global DOF number for local DOF 1 can be determined based on the shear panel element row and column. Obviously, the global DOF number for local DOF 2 is determined by adding $2 \times n_x + 1$ (the number of DOF per stringer element row) to this number. The same applies to determining the global DOF numbers for local DOF 3 and 4. Once the global DOF number for local DOF 3 has been determined, which can be done based on the shear panel element row and column and the number of DOF already assigned in x-direction, the global DOF number for local DOF 4 is determined by adding $2 \times n_y + 1$ to this number.

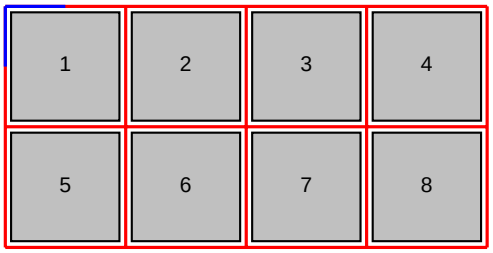


Figure 53 Numbering shear panel elements

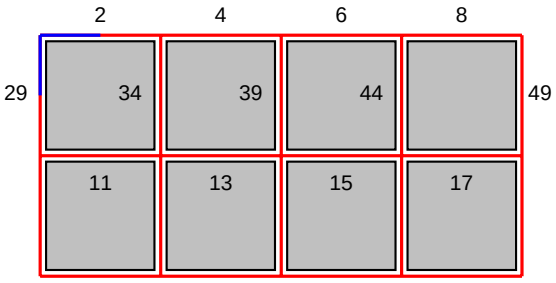


Figure 52 Numbering shear panel element DOF for the first row

Generating shear panel elements

The global DOF numbers per shear panel element are stored in a matrix called `shearPanelDOF`. Each row corresponds to one shear panel element and has four columns. The first column holds the global DOF numbers per shear panel element corresponding to local DOF number 1. Similarly, columns 2, 3 and 4 hold the global DOF numbers per shear panel element corresponding to local DOF numbers 2 and 3 respectively 4. The effective depth and the corresponding shear stiffness is determined in this procedure as well. The source code of the procedure `generateShearPanels ()` reads:

```
private void generateShearPanels ()
{

    // variable declaration

    int nx = nrOfRebarsY - 1;  // number of shear panels in X-direction
    int ny = nrOfRebarsX - 1;  // number of shear panels in Y-direction
    double t;


    // determine the number of shear panels used in the model

    nrOfShearPanels = nx * ny;


    // create a matrix which stores the DOF per shear panel

    shearPanelDOF = new IntMatrix (nrOfShearPanels, 4);


    // determine the shear stiffness of a shear panel

    t = concreteCover + (rebarDiameterX + rebarDiameterY) / 4 +
        (ctcDistanceOfRebarsX + ctcDistanceOfRebarsY) / 4;


    if (t > ((ctcDistanceOfRebarsX + ctcDistanceOfRebarsY) / 2))
    {

        t = (ctcDistanceOfRebarsX + ctcDistanceOfRebarsY) / 2;

    }


    if (t > capDepth)
    {

        t = capDepth;

    }


    Gt = G_CapConcrete * t;


    // number the DOF per shear panel

    for (int j = 1; j <= ny; j++)
    {

        for (int i = 1; i <= nx; i++)
        {

            // determine the shear panel number, starting from 0

            int shearPanelNr = (j - 1) * nx + i - 1;
```

```
                int n = (j - 1) * (2 * nx + 1) + 2 * i;
                shearPanelDOF.setEntry (shearPanelNr, 0, n);
                n = j * (2 * nx + 1) + 2 * i;
                shearPanelDOF.setEntry (shearPanelNr, 1, n);
                n = (2 * nx + 1) * nrOfRebarsX + (i - 1) * (2 * ny + 1) + 2 * j;
                shearPanelDOF.setEntry (shearPanelNr, 2, n);
                n = (2 * nx + 1) * nrOfRebarsX + i * (2 * ny + 1) + 2 * j;
                shearPanelDOF.setEntry (shearPanelNr, 3, n);

            }

        }

    }
```

Appendix A3: Numbering and generating strut elements

This appendix discusses numbering and generating strut elements. First, it is explained how strut elements and their degrees of freedom (DOF) are numbered. Then the source code of the procedure `generateStruts ()` is given.

Numbering strut elements and their DOF

First, the 'internal' strut elements are numbered. The strut element from column 1 to pile 1 receives strut element 1, the strut element from column 1 to pile 2 is called strut element 2, and so on (Figure 55). This process is repeated for all piles and columns. Then the piles, which are also strut elements, are numbered in the pile number order starting from the strut element number which was assigned last to an 'internal' strut element. Simultaneously, the global DOF numbers are assigned (Figure 54). For the 'internal' strut elements, first the DOF at the pile side are numbered. In Figure 54 the concerning strut element ends are numbered 1, 2 and 3. Then the DOF at the column side of these strut element are numbered (4 in Figure 54). For the piles, first the DOF at the shear panel element side are numbered (1, 2 and 3 in Figure 54), then the DOF at the column tip side (5, 6 and 7 in Figure 54). Note that the DOF of the 'internal' struts at the shear panel element side coincide with the DOF of the corresponding piles at the same side. Per strut element end, 3 DOF need to be numbered. First the DOF pointing in x-direction is

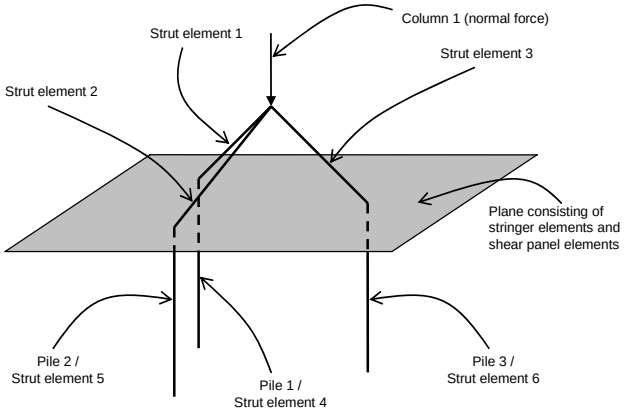


Figure 55 Numbering strut elements

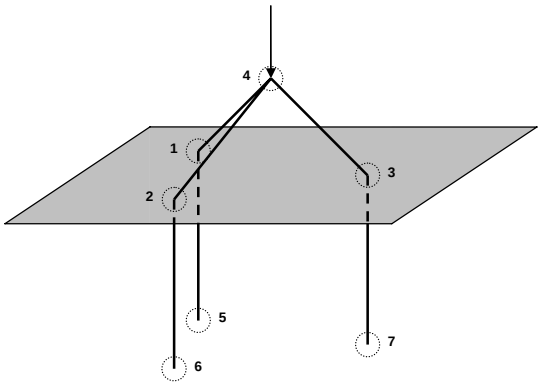


Figure 54 Order of numbering strut element DOF

numbered, then the DOF pointing in y-direction and finally the DOF pointing in z-direction.

Generating strut elements

The global DOF numbers per strut element are stored in a matrix called `strutDOF`. Each row corresponds to one strut element and has six columns. The first column holds the global DOF numbers per strut element corresponding to local DOF number 1. Similarly, columns 2, 3, 4, 5 and 6 hold the global DOF numbers per strut element corresponding to local DOF numbers 2, 3, 4, 5 respectively 6. While the DOF numbers are assigned, also the extensional stiffness EA and the strut element length ℓ are determined and stored in vectors: `strutEA` respectively `strutLength`. Another matrix, `strutGonio`, holds the values for $\sin\alpha$, $\cos\alpha$, $\sin\beta$, $\cos\beta$ and β. For the definition of angles α and β, refer to Section 3.4. The source code of the procedure `generateStruts ()` reads:

```
private void generateStruts ()
{

    // variable declaration

    int nx = nrOfRebarsY - 1;  // number of shear panels in X-direction
    int ny = nrOfRebarsX - 1;  // number of shear panels in Y-direction
    int k = (2 * nx + 1) * nrOfRebarsX + (2 * ny + 1) * nrOfRebarsY;
    int l = k + 3 * nrOfPiles + 3 * nrOfColumns;


    // determine the number of struts used in the model

    nrOfStruts = (nrOfColumns + 1) * nrOfPiles;


    // create a matrix which stores the DOF per strut

    strutDOF = new IntMatrix (nrOfStruts, 6);


    // create a vector which stores the extensional stiffness per strut
    // and one for the length per strut

    strutEA = new DoubleVector (nrOfStruts);
    strutLength = new DoubleVector (nrOfStruts);


    // create a matrix which stores sin(Alpha), cos(Alpha),
    // sin(Beta), cos(Beta) and Beta per strut

    strutGonio = new DoubleMatrix (nrOfStruts, 5);


    // calculate the extensional stiffness of a pile

    pileEA = E_PileConcrete * pileSection;


    // number the DOF of the internal struts

    for (int i = 1; i <= nrOfColumns; i++)
    {

        for (int j = 1; j <= nrOfPiles; j++)
        {

            // determine the strut number, starting from 0
```

```
int strutNr = (i - 1) * nrOfPiles + j - 1;


int n = k + 3 * j - 2;
strutDOF.setEntry (strutNr, 0, n);
n = k + 3 * j - 1;
strutDOF.setEntry (strutNr, 1, n);
n = k + 3 * j;
strutDOF.setEntry (strutNr, 2, n);
n = k + 3 * nrOfPiles + 3 * i - 2;
strutDOF.setEntry (strutNr, 3, n);
n = k + 3 * nrOfPiles + 3 * i - 1;
strutDOF.setEntry (strutNr, 4, n);
n = k + 3 * nrOfPiles + 3 * i;
strutDOF.setEntry (strutNr, 5, n);


// get column co-ordinates and pile co-ordinates

double columnX = columnXY.getEntry ((i - 1), 0);
double columnY = columnXY.getEntry ((i - 1), 1);
double pileX = pileXY.getEntry ((j - 1), 0);
double pileY = pileXY.getEntry ((j - 1), 1);


// determine the projected strut length, the strut length and
// the extensional stiffness of the strut

double deltaX = columnX - pileX;
double deltaY = columnY - pileY;
double deltaZ = capDepth;
double projectedStrutLength = Math.sqrt (deltaX * deltaX +
                              deltaY * deltaY);
double strutL = Math.sqrt (deltaZ * deltaZ +
                projectedStrutLength * projectedStrutLength);
strutLength.setEntry (strutNr, strutL);
strutEA.setEntry (strutNr, pileEA);


// determine sin(Alpha), cos(Alpha), sin(Beta), cos(Beta) and
// Beta and store these values in strutGonio

if ((columnX == pileX) && (columnY == pileY))
{

    // the strut has a vertical orientation, which means that
    // Alpha is not defined
    // in this case the goniometric values are not important,
    // so fill the first four entries with zeros

    strutGonio.setEntry (strutNr, 0, 0.0);
    strutGonio.setEntry (strutNr, 1, 0.0);
    strutGonio.setEntry (strutNr, 2, 0.0);
    strutGonio.setEntry (strutNr, 3, 0.0);
    strutGonio.setEntry (strutNr, 4, 90.0);

}
else
{

    double sinAlpha = deltaY / projectedStrutLength;
    strutGonio.setEntry (strutNr, 0, sinAlpha);
    double cosAlpha = deltaX / projectedStrutLength;
    strutGonio.setEntry (strutNr, 1, cosAlpha);
    double sinBeta = deltaZ / strutL;
    strutGonio.setEntry (strutNr, 2, sinBeta);
    double cosBeta = projectedStrutLength / strutL;
    strutGonio.setEntry (strutNr, 3, cosBeta);
```

```
                double Beta = Math.toDegrees (Math.asin (deltaZ / strutL));
                strutGonio.setEntry (strutNr, 4, Beta);

            }

        }

    }


    // number the DOF of the piles

    for (int i = 1; i <= nrOfPiles; i++)
    {

        // determine the strut number, starting from the last number
        // assigned

        int strutNr = nrOfPiles * nrOfColumns + i - 1;


        int n = k + 3 * i - 2;
        strutDOF.setEntry (strutNr, 0, n);
        n = k + 3 * i - 1;
        strutDOF.setEntry (strutNr, 1, n);
        n = k + 3 * i;
        strutDOF.setEntry (strutNr, 2, n);
        n = l + 3 * i - 2;
        strutDOF.setEntry (strutNr, 3, n);
        n = l + 3 * i - 1;
        strutDOF.setEntry (strutNr, 4, n);
        n = l + 3 * i;
        strutDOF.setEntry (strutNr, 5, n);


        // determine and store the strut extensional stiffness and length

        strutEA.setEntry (strutNr, pileEA);
        strutLength.setEntry (strutNr, pileLength);


        // set the pile gonio

        strutGonio.setEntry (strutNr, 0, 0.0);
        strutGonio.setEntry (strutNr, 1, 0.0);
        strutGonio.setEntry (strutNr, 2, 0.0);
        strutGonio.setEntry (strutNr, 3, 0.0);
        strutGonio.setEntry (strutNr, 4, 90.0);

    }

}
```

Appendix B1: Assembling the elements

This appendix gives the source code for assembling the three different elements.

Assembling stringer elements

```
private void assembleStringers ()
{

    // create an element stiffness matrix

    DoubleMatrix elementStiffnessMatrix = new DoubleMatrix (3, 3);


    // set the entries of the element stiffness matrix

    elementStiffnessMatrix.setEntry (0, 0, 4.0);
    elementStiffnessMatrix.setEntry (0, 1, -6.0);
    elementStiffnessMatrix.setEntry (0, 2, 2.0);
    elementStiffnessMatrix.setEntry (1, 0, -6.0);
    elementStiffnessMatrix.setEntry (1, 1, 12.0);
    elementStiffnessMatrix.setEntry (1, 2, -6.0);
    elementStiffnessMatrix.setEntry (2, 0, 2.0);
    elementStiffnessMatrix.setEntry (2, 1, -6.0);
    elementStiffnessMatrix.setEntry (2, 2, 4.0);


    // assemble the stringers into the system stiffness matrix

    for (int stringerNr = 0; stringerNr < nrOfStringers; stringerNr++)
    {

        double k = stringerEA.getEntry (stringerNr) /
stringerLength.getEntry (stringerNr);


        for (int ii = 0; ii < 3; ii++)
        {

            int i = stringerDOF.getEntry (stringerNr, ii) - 1;


            for (int jj = 0; jj < 3; jj++)
            {

                int j = stringerDOF.getEntry (stringerNr, jj) - 1;
                double temp = systemStiffnessMatrix.getEntry (i, j)
                              + k * elementStiffnessMatrix.getEntry (ii,
jj);
                systemStiffnessMatrix.setEntry (i, j, temp);

            }

        }

    }

}
```

Assembling shear panel elements

```
private void assembleShearPanels ()
{

    // create an element stiffness matrix

    DoubleMatrix elementStiffnessMatrix = new DoubleMatrix (4, 4);


    // calculate length-to-width ratio and width-to-length ratio

    double alpha = ctcDistanceOfRebarsY / ctcDistanceOfRebarsX;
    double beta = ctcDistanceOfRebarsX / ctcDistanceOfRebarsY;


    // fill the entries of the element stiffness matrix

    elementStiffnessMatrix.setEntry (0, 0, alpha);
    elementStiffnessMatrix.setEntry (0, 1, -alpha);
    elementStiffnessMatrix.setEntry (0, 2, 1.0);
    elementStiffnessMatrix.setEntry (0, 3, -1.0);
    elementStiffnessMatrix.setEntry (1, 0, -alpha);
    elementStiffnessMatrix.setEntry (1, 1, alpha);
    elementStiffnessMatrix.setEntry (1, 2, -1.0);
    elementStiffnessMatrix.setEntry (1, 3, 1.0);
    elementStiffnessMatrix.setEntry (2, 0, 1.0);
    elementStiffnessMatrix.setEntry (2, 1, -1.0);
    elementStiffnessMatrix.setEntry (2, 2, beta);
    elementStiffnessMatrix.setEntry (2, 3, -beta);
    elementStiffnessMatrix.setEntry (3, 0, -1.0);
    elementStiffnessMatrix.setEntry (3, 1, 1.0);
    elementStiffnessMatrix.setEntry (3, 2, -beta);
    elementStiffnessMatrix.setEntry (3, 3, beta);


    // assemble the shear panels into the system stiffness matrix

    for (int shearPanelNr = 0; shearPanelNr < nrOfShearPanels;
shearPanelNr++)
    {

        for (int ii = 0; ii < 4; ii++)
        {

            int i = shearPanelDOF.getEntry (shearPanelNr, ii) - 1;


            for (int jj = 0; jj < 4; jj++)
            {

                int j = shearPanelDOF.getEntry (shearPanelNr, jj) - 1;
                double temp = systemStiffnessMatrix.getEntry (i, j)
                             + Gt * elementStiffnessMatrix.getEntry (ii,
jj);
                systemStiffnessMatrix.setEntry (i, j, temp);

            }

        }

    }

}
```

Assembling strut elements

```
private void assembleStruts ()
{

    // create an element stiffness matrix

    DoubleMatrix elementStiffnessMatrix = new DoubleMatrix (6, 6);


    // process the internal struts first

    for (int columnNr = 0; columnNr < nrOfColumns; columnNr++)
    {

        for (int pileNr = 0; pileNr < nrOfPiles; pileNr++)
        {

            int strutNr = columnNr * nrOfPiles + (pileNr + 1) - 1;
            double k = strutEA.getEntry (strutNr) / strutLength.getEntry
(strutNr);
            double columnX = columnXY.getEntry (columnNr, 0);
            double columnY = columnXY.getEntry (columnNr, 1);
            double pileX = pileXY.getEntry (pileNr, 0);
            double pileY = pileXY.getEntry (pileNr, 1);


            // if the strut has a vertical orientation then use the same
            // element stiffness matrix as for the piles

            if ((columnX == pileX) && (columnY == pileY))
            {

                elementStiffnessMatrix.setEntry (0, 0, 0.0);
                elementStiffnessMatrix.setEntry (0, 1, 0.0);
                elementStiffnessMatrix.setEntry (0, 2, 0.0);
                elementStiffnessMatrix.setEntry (0, 3, 0.0);
                elementStiffnessMatrix.setEntry (0, 4, 0.0);
                elementStiffnessMatrix.setEntry (0, 5, 0.0);
                elementStiffnessMatrix.setEntry (1, 0, 0.0);
                elementStiffnessMatrix.setEntry (1, 1, 0.0);
                elementStiffnessMatrix.setEntry (1, 2, 0.0);
                elementStiffnessMatrix.setEntry (1, 3, 0.0);
                elementStiffnessMatrix.setEntry (1, 4, 0.0);
                elementStiffnessMatrix.setEntry (1, 5, 0.0);
                elementStiffnessMatrix.setEntry (2, 0, 0.0);
                elementStiffnessMatrix.setEntry (2, 1, 0.0);
                elementStiffnessMatrix.setEntry (2, 2, 1.0);
                elementStiffnessMatrix.setEntry (2, 3, 0.0);
                elementStiffnessMatrix.setEntry (2, 4, 0.0);
                elementStiffnessMatrix.setEntry (2, 5, -1.0);
                elementStiffnessMatrix.setEntry (3, 0, 0.0);
                elementStiffnessMatrix.setEntry (3, 1, 0.0);
                elementStiffnessMatrix.setEntry (3, 2, 0.0);
                elementStiffnessMatrix.setEntry (3, 3, 0.0);
                elementStiffnessMatrix.setEntry (3, 4, 0.0);
                elementStiffnessMatrix.setEntry (3, 5, 0.0);
                elementStiffnessMatrix.setEntry (4, 0, 0.0);
                elementStiffnessMatrix.setEntry (4, 1, 0.0);
                elementStiffnessMatrix.setEntry (4, 2, 0.0);
                elementStiffnessMatrix.setEntry (4, 3, 0.0);
                elementStiffnessMatrix.setEntry (4, 4, 0.0);
                elementStiffnessMatrix.setEntry (4, 5, 0.0);
                elementStiffnessMatrix.setEntry (5, 0, 0.0);
                elementStiffnessMatrix.setEntry (5, 1, 0.0);
                elementStiffnessMatrix.setEntry (5, 2, -1.0);
                elementStiffnessMatrix.setEntry (5, 3, 0.0);
                elementStiffnessMatrix.setEntry (5, 4, 0.0);
```

```
                elementStiffnessMatrix.setEntry (5, 5, 1.0);

            }
            else
            {

                double sinAlpha = strutGonio.getEntry (strutNr, 0);
                double cosAlpha = strutGonio.getEntry (strutNr, 1);
                double sinBeta = strutGonio.getEntry (strutNr, 2);
                double cosBeta = strutGonio.getEntry (strutNr, 3);


                elementStiffnessMatrix.setEntry (0, 0, (cosAlpha * cosAlpha
* cosBeta * cosBeta));
                elementStiffnessMatrix.setEntry (0, 1, (cosAlpha * cosBeta *
cosBeta * sinAlpha));
                elementStiffnessMatrix.setEntry (0, 2, (-cosAlpha * cosBeta
* sinBeta));
                elementStiffnessMatrix.setEntry (0, 3, (-cosAlpha * cosAlpha
* cosBeta * cosBeta));
                elementStiffnessMatrix.setEntry (0, 4, (-cosAlpha * cosBeta
* cosBeta * sinAlpha));
                elementStiffnessMatrix.setEntry (0, 5, (cosAlpha * cosBeta *
sinBeta));
                elementStiffnessMatrix.setEntry (1, 0, (cosAlpha * cosBeta *
cosBeta * sinAlpha));
                elementStiffnessMatrix.setEntry (1, 1, (sinAlpha * sinAlpha
* cosBeta * cosBeta));
                elementStiffnessMatrix.setEntry (1, 2, (-sinAlpha * cosBeta
* sinBeta));
                elementStiffnessMatrix.setEntry (1, 3, (-cosAlpha * cosBeta
* cosBeta * sinAlpha));
                elementStiffnessMatrix.setEntry (1, 4, (-sinAlpha * sinAlpha
* cosBeta * cosBeta));
                elementStiffnessMatrix.setEntry (1, 5, (sinAlpha * cosBeta *
sinBeta));
                elementStiffnessMatrix.setEntry (2, 0, (-cosAlpha * cosBeta
* sinBeta));
                elementStiffnessMatrix.setEntry (2, 1, (-sinAlpha * cosBeta
* sinBeta));
                elementStiffnessMatrix.setEntry (2, 2, (sinBeta * sinBeta));
                elementStiffnessMatrix.setEntry (2, 3, (cosAlpha * cosBeta *
sinBeta));
                elementStiffnessMatrix.setEntry (2, 4, (sinAlpha * cosBeta *
sinBeta));
                elementStiffnessMatrix.setEntry (2, 5, (-sinBeta *
sinBeta));
                elementStiffnessMatrix.setEntry (3, 0, (-cosAlpha * cosAlpha
* cosBeta * cosBeta));
                elementStiffnessMatrix.setEntry (3, 1, (-cosAlpha * cosBeta
* cosBeta * sinAlpha));
                elementStiffnessMatrix.setEntry (3, 2, (cosAlpha * cosBeta *
sinBeta));
                elementStiffnessMatrix.setEntry (3, 3, (cosAlpha * cosAlpha
* cosBeta * cosBeta));
                elementStiffnessMatrix.setEntry (3, 4, (cosAlpha * cosBeta *
cosBeta * sinAlpha));
                elementStiffnessMatrix.setEntry (3, 5, (-cosAlpha * cosBeta
* sinBeta));
                elementStiffnessMatrix.setEntry (4, 0, (-cosAlpha * cosBeta
* cosBeta * sinAlpha));
                elementStiffnessMatrix.setEntry (4, 1, (-sinAlpha * sinAlpha
* cosBeta * cosBeta));
                elementStiffnessMatrix.setEntry (4, 2, (sinAlpha * cosBeta *
sinBeta));
                elementStiffnessMatrix.setEntry (4, 3, (cosAlpha * cosBeta *
cosBeta * sinAlpha));
                elementStiffnessMatrix.setEntry (4, 4, (sinAlpha * sinAlpha
* cosBeta * cosBeta));
                elementStiffnessMatrix.setEntry (4, 5, (-sinAlpha * cosBeta
* sinBeta));
```

```
                elementStiffnessMatrix.setEntry (5, 0, (cosAlpha * cosBeta *
sinBeta));
                elementStiffnessMatrix.setEntry (5, 1, (sinAlpha * cosBeta *
sinBeta));
                elementStiffnessMatrix.setEntry (5, 2, (-sinBeta *
sinBeta));
                elementStiffnessMatrix.setEntry (5, 3, (-cosAlpha * cosBeta
* sinBeta));
                elementStiffnessMatrix.setEntry (5, 4, (-sinAlpha * cosBeta
* sinBeta));
                elementStiffnessMatrix.setEntry (5, 5, (sinBeta * sinBeta));

            }


            for (int ii = 0; ii < 6; ii++)
            {

                int i = strutDOF.getEntry (strutNr, ii) - 1;


                for (int jj = 0; jj < 6; jj++)
                {

                    int j = strutDOF.getEntry (strutNr, jj) - 1;
                    double temp = systemStiffnessMatrix.getEntry (i, j)
                                + k * elementStiffnessMatrix.getEntry (ii,
jj);
                    systemStiffnessMatrix.setEntry (i, j, temp);

                }

            }

        }

    }


    // process the piles

    elementStiffnessMatrix.setEntry (0, 0, 0.0);
    elementStiffnessMatrix.setEntry (0, 1, 0.0);
    elementStiffnessMatrix.setEntry (0, 2, 0.0);
    elementStiffnessMatrix.setEntry (0, 3, 0.0);
    elementStiffnessMatrix.setEntry (0, 4, 0.0);
    elementStiffnessMatrix.setEntry (0, 5, 0.0);
    elementStiffnessMatrix.setEntry (1, 0, 0.0);
    elementStiffnessMatrix.setEntry (1, 1, 0.0);
    elementStiffnessMatrix.setEntry (1, 2, 0.0);
    elementStiffnessMatrix.setEntry (1, 3, 0.0);
    elementStiffnessMatrix.setEntry (1, 4, 0.0);
    elementStiffnessMatrix.setEntry (1, 5, 0.0);
    elementStiffnessMatrix.setEntry (2, 0, 0.0);
    elementStiffnessMatrix.setEntry (2, 1, 0.0);
    elementStiffnessMatrix.setEntry (2, 2, 1.0);
    elementStiffnessMatrix.setEntry (2, 3, 0.0);
    elementStiffnessMatrix.setEntry (2, 4, 0.0);
    elementStiffnessMatrix.setEntry (2, 5, -1.0);
    elementStiffnessMatrix.setEntry (3, 0, 0.0);
    elementStiffnessMatrix.setEntry (3, 1, 0.0);
    elementStiffnessMatrix.setEntry (3, 2, 0.0);
    elementStiffnessMatrix.setEntry (3, 3, 0.0);
    elementStiffnessMatrix.setEntry (3, 4, 0.0);
    elementStiffnessMatrix.setEntry (3, 5, 0.0);
    elementStiffnessMatrix.setEntry (4, 0, 0.0);
    elementStiffnessMatrix.setEntry (4, 1, 0.0);
    elementStiffnessMatrix.setEntry (4, 2, 0.0);
    elementStiffnessMatrix.setEntry (4, 3, 0.0);
    elementStiffnessMatrix.setEntry (4, 4, 0.0);
    elementStiffnessMatrix.setEntry (4, 5, 0.0);
```

```
    elementStiffnessMatrix.setEntry (5, 0, 0.0);
    elementStiffnessMatrix.setEntry (5, 1, 0.0);
    elementStiffnessMatrix.setEntry (5, 2, -1.0);
    elementStiffnessMatrix.setEntry (5, 3, 0.0);
    elementStiffnessMatrix.setEntry (5, 4, 0.0);
    elementStiffnessMatrix.setEntry (5, 5, 1.0);


    for (int strutNr = nrOfStruts - nrOfPiles; strutNr < nrOfStruts;
strutNr++)
    {

        double k = strutEA.getEntry (strutNr) / strutLength.getEntry
(strutNr);


        for (int ii = 0; ii < 6; ii++)
        {

            int i = strutDOF.getEntry (strutNr, ii) - 1;


            for (int jj = 0; jj < 6; jj++)
            {

                int j = strutDOF.getEntry (strutNr, jj) - 1;
                double temp = systemStiffnessMatrix.getEntry (i, j)
                              + k * elementStiffnessMatrix.getEntry (ii,
jj);
                systemStiffnessMatrix.setEntry (i, j, temp);

            }

        }

    }

}
```

Appendix B2: Generating and processing imposed forces

This appendix gives the source code for generating and processing the imposed forces.

Generating imposed forces

```
private void generateImposedForces ()
{

    // variable declaration

    int nx = nrOfRebarsY - 1;  // number of shear panels in X-direction
    int ny = nrOfRebarsX - 1;  // number of shear panels in Y-direction
    int k = (2 * nx + 1) * nrOfRebarsX + (2 * ny + 1) * nrOfRebarsY;


    // create a vector which stores the degrees of freedom per imposed
normal force

    loadingDOF = new IntVector (nrOfColumns);


    // determine on which degrees of freedom a force is applied

    for (int i = 0; i < nrOfColumns; i++)
    {

        int n = k + 3 * nrOfPiles + 3 * (i + 1);
        loadingDOF.setEntry (i, n);
    }

}
```

Processing imposed forces

```
private void processImposedForces ()
{

    // create the force vector

    forceVector = new DoubleVector (nrOfDOF);


    // initialise the force vector

    for (int i = 0; i < nrOfDOF; i++)
    {

        forceVector.setEntry (i, 0.0);

    }


    // process the normal forces imposed by the columns

    for (int i = 0; i < nrOfColumns; i++)
    {

        int n = loadingDOF.getEntry (i) - 1;
        double temp = forceVector.getEntry (i) + columnNormalForces.getEntry
(i);
        forceVector.setEntry (n, temp);

    }

}
```

Appendix B3: Generating and processing tying

This appendix gives the source code for generating and processing tying.

Generating tying

```
private void generateTyings ()
{

    // variable declaration

    int nx = nrOfRebarsY - 1;  // number of shear panels in X-direction
    int ny = nrOfRebarsX - 1;  // number of shear panels in Y-direction
    int k = (2 * nx + 1) * nrOfRebarsX + (2 * ny + 1) * nrOfRebarsY;


    // determine the number of tyings

    nrOfTyings = 2 * nrOfPiles;


    // create a matrix which stores the degrees of freedom per tying
    // the first entry stores the degree of freedom of master1
    // the second entry stores the degree of freedom of master2
    // the third entry stores the degree of freedom of slave

    tyingDOF = new IntMatrix (nrOfTyings, 3);


    // create a matrix which stores the tying factors
    // the first entry stores factor1
    // the second entry stores factor2

    tyingFactors = new DoubleMatrix (nrOfTyings, 2);


    for (int i = 0; i < nrOfPiles; i++)
    {

        double temp = (pileXY.getEntry (i, 0) - concreteCover -
(rebarDiameterY / 2)) / ctcDistanceOfRebarsY;
        int nxTying = (int) Math.floor (temp);
        temp = (pileXY.getEntry (i, 1) - concreteCover - (rebarDiameterX /
2)) / ctcDistanceOfRebarsX;
        int nyTying = (int) Math.floor (temp);


        // tying in X-direction

        int tyingNr = 2 * i;
        int n = nyTying * (2 * nx + 1) + (nxTying + 1) * 2;
        tyingDOF.setEntry (tyingNr, 0, n);
        n = (nyTying + 1) * (2 * nx + 1) + (nxTying + 1) * 2;
        tyingDOF.setEntry (tyingNr, 1, n);
        n = k + 3 * (i + 1) - 2;
        tyingDOF.setEntry (tyingNr, 2, n);
        temp = ((nyTying + 1) * ctcDistanceOfRebarsX + concreteCover +
(rebarDiameterX / 2)
                - pileXY.getEntry (i, 1)) / ctcDistanceOfRebarsX;
        tyingFactors.setEntry (tyingNr, 0, temp);
        tyingFactors.setEntry (tyingNr, 1, (1 - temp));


        // tying in Y-direction
```

```
        tyingNr = 2 * i + 1;
        n = (2 * nx + 1) * nrOfRebarsX + nxTying * (2 * ny + 1) + (nyTying +
1) * 2;
        tyingDOF.setEntry (tyingNr, 0, n);
        n = (2 * nx + 1) * nrOfRebarsX + (nxTying + 1) * (2 * ny + 1) +
(nyTying + 1) * 2;
        tyingDOF.setEntry (tyingNr, 1, n);
        n = k + 3 * (i + 1) - 1;
        tyingDOF.setEntry (tyingNr, 2, n);
        temp = ((nxTying + 1) * ctcDistanceOfRebarsY + concreteCover +
(rebarDiameterY / 2)
               - pileXY.getEntry (i, 0)) / ctcDistanceOfRebarsY;
        tyingFactors.setEntry (tyingNr, 0, temp);
        tyingFactors.setEntry (tyingNr, 1, (1 - temp));

    }

}
```

Processing tying

```
private void processTyings ()
{

    for (int tyingNr = 0; tyingNr < nrOfTyings; tyingNr++)
    {

        int master1DOF = tyingDOF.getEntry (tyingNr, 0) - 1;
        int master2DOF = tyingDOF.getEntry (tyingNr, 1) - 1;
        int slaveDOF = tyingDOF.getEntry (tyingNr, 2) - 1;
        double factor1 = tyingFactors.getEntry (tyingNr, 0);
        double factor2 = tyingFactors.getEntry (tyingNr, 1);


        for (int j = 0; j < nrOfDOF; j++)
        {

            double temp = systemStiffnessMatrix.getEntry (master1DOF, j)
                        + factor1 * systemStiffnessMatrix.getEntry
(slaveDOF, j);
            systemStiffnessMatrix.setEntry (master1DOF, j, temp);
            temp = systemStiffnessMatrix.getEntry (master2DOF, j)
                   + factor2 * systemStiffnessMatrix.getEntry (slaveDOF, j);
            systemStiffnessMatrix.setEntry (master2DOF, j, temp);
            systemStiffnessMatrix.setEntry (slaveDOF, j, 0.0);

        }


        systemStiffnessMatrix.setEntry (slaveDOF, master1DOF, factor1);
        systemStiffnessMatrix.setEntry (slaveDOF, master2DOF, factor2);
        systemStiffnessMatrix.setEntry (slaveDOF, slaveDOF, -1.0);
        double temp = forceVector.getEntry (master1DOF)
                    + factor1 * forceVector.getEntry (slaveDOF);
        forceVector.setEntry (master1DOF, temp);
        temp = forceVector.getEntry (master2DOF)
               + factor2 * forceVector.getEntry (slaveDOF);
        forceVector.setEntry (master2DOF, temp);
        forceVector.setEntry (slaveDOF, 0.0);

    }

}
```

Appendix B4: Generating and processing imposed displacements

This appendix contains the source code which implements the processing of the imposed displacements.

The source code of the procedure `generateImposedDisplacements ()` reads:

```
    // method which generates the imposed displacements

    private void generateImposedDisplacements ()
    {

        // variable declaration

        int nx = nrOfRebarsY - 1;  // number of shear panels in X-direction
        int ny = nrOfRebarsX - 1;  // number of shear panels in Y-direction
        int k = (2 * nx + 1) * nrOfRebarsX + (2 * ny + 1) * nrOfRebarsY;
        int l = k + 3 * nrOfPiles + 3 * nrOfColumns;


        // determine the number of fixed degrees of freedom

        nrOfFixedDOF = 3 + 3 * nrOfPiles;


        // create a vector which stores the degree of freedom number per
fixed degree of freedom
        // and one which stores the magnitude of the imposed displacements

        fixedDOF = new IntVector (nrOfFixedDOF);
        prescribedDisplacements = new DoubleVector (nrOfFixedDOF);


        // determine which degrees of freedom are fixed

        fixedDOF.setEntry (0, 1);
        prescribedDisplacements.setEntry (0, 0.0);
        int n = (2 * nx + 1) * nrOfRebarsX + 1;
        fixedDOF.setEntry (1, n);
        prescribedDisplacements.setEntry (1, 0.0);
        n = (2 * nx + 1) * (nrOfRebarsX - 1) + 1;
        fixedDOF.setEntry (2, n);
        prescribedDisplacements.setEntry (2, 0.0);


        for (int i = 1; i <= nrOfPiles; i++)
        {

            n = l + 3 * i - 2;
            fixedDOF.setEntry ((3 * i), n);
            prescribedDisplacements.setEntry ((3 * i), 0.0);
            n = l + 3 * i - 1;
            fixedDOF.setEntry ((3 * i + 1), n);
            prescribedDisplacements.setEntry ((3 * i + 1), 0.0);
            n = l + 3 * i;
            fixedDOF.setEntry ((3 * i + 2), n);
            prescribedDisplacements.setEntry ((3 * i + 2), 0.0);

        }

    }
```

The source code of the procedure `processImposedDisplacements ()` reads:

```
     // method which processes the imposed displacements (fixed degrees of
freedom)

     private void processImposedDisplacements ()
     {

          // create a matrix which stores the entries from lines that will be
removed
          // from the system stiffness matrix

          removedLines = new DoubleMatrix (nrOfFixedDOF, nrOfDOF);


          for (int fixedDOF_Nr = 0; fixedDOF_Nr < nrOfFixedDOF;
fixedDOF_Nr++)
          {

              int n = fixedDOF.getEntry (fixedDOF_Nr) - 1;


              for (int j = 0; j < nrOfDOF; j++)
              {

                  double temp = systemStiffnessMatrix.getEntry (n, j);
                  removedLines.setEntry (fixedDOF_Nr, j, temp);
                  systemStiffnessMatrix.setEntry (n, j, 0.0);

              }


              systemStiffnessMatrix.setEntry (n, n, 1.0);
              double temp = prescribedDisplacements.getEntry (fixedDOF_Nr);
              forceVector.setEntry (n, temp);

          }

     }
```

Appendix B5: Detailed consideration on LU decomposition

This appendix gives a detailed consideration on how to solve for vector $\underline{x}$ in the system $\underline{\underline{A}} \cdot \underline{x} = \underline{b}$, given matrix $\underline{\underline{A}}$ and vector $\underline{b}$. The first three parts of this appendix are an elaboration on the steps presented in Section 4.5. This means that part one considers the theory of the actual decomposition of matrix $\underline{\underline{A}}$ in matrices $\underline{\underline{L}}$ and $\underline{\underline{U}}$. Then part two explains how to solve for vector $\underline{y}$ starting from matrix $\underline{\underline{L}}$ and vector $\underline{b}$. Part three discusses how to solve for vector $\underline{x}$ starting from matrix $\underline{\underline{U}}$ and vector $\underline{y}$. The fourth part of this appendix gives the source code of the class which implements the solution procedure. For reusability reasons this procedure has been implemented in a separate module. Therefore, separate testing of this module is possible, which has been elaborated in the fifth and final part of this appendix. The text in this appendix, accept for the part that concerns testing, has been based on Press (1988) [9].

Decomposition of matrix $\underline{\underline{A}}$ in matrices $\underline{\underline{L}}$ and $\underline{\underline{U}}$

The basis of LU decomposition is that matrix $\underline{\underline{A}}$ may be written as the product of two matrices, namely a lower triangular matrix and an upper triangular matrix:

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & \cdots & 0 \\ \alpha_{21} & \alpha_{22} & 0 & \cdots & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{n1} & \alpha_{n2} & \alpha_{n3} & \cdots & \alpha_{nn} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \cdots & \beta_{1n} \\ 0 & \beta_{22} & \beta_{23} & \cdots & \beta_{2n} \\ 0 & 0 & \beta_{33} & \cdots & \beta_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \beta_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix},$$

where all α_{ij} and β_{ij} are to be determined and all a_{ij} are known. An arbitrary element a_{ij} can be calculated from the following formula:

$$a_{ij} = \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + ... + \alpha_{in}\beta_{nj} .$$

However, since the lower triangular and upper triangular matrices contain many zeros, it is better to leave the multiplication terms containing one or more zeros out. To do this, three different cases have to be distinguished:

$$i < j: \ \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + ... + \alpha_{ii}\beta_{ij} = a_{ij} \qquad (38)$$

$$i = j: \ \alpha_{i1}\beta_{1i} + \alpha_{i2}\beta_{2i} + ... + \alpha_{ii}\beta_{ii} = a_{ii} \qquad (39)$$

$$i > j: \ \alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + ... + \alpha_{ij}\beta_{jj} = a_{ij} \qquad (40)$$

The number of unknowns in the lower triangular matrix is equal to $\left(n^2 - n\right)/2 + n$. The same applies to the number of unknowns in the upper triangular matrix. Therefore, the total number of unknowns in both matrices comes down to $n^2 + n$. But only n^2 equations (the number of elements in $\underline{\underline{A}}$) can be formulated for these unknowns. This

means that n unknowns may be specified arbitrarily and then try to solve for the remaining unknowns. Press (1988) [9] states that it is always possible to take:

$\alpha_{ii} \equiv 1$ for $i = 1, 2, ..., n$.

The remaining unknowns are solved using Crout's algorithm. This method decomposes matrix $\underline{\underline{A}}$ column by column. For a certain column j two procedures have to be carried out. First for $i = 1, 2, ..., j$:

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik} \beta_{kj} \qquad (41)$$

This formulation can be deduced from equations (38) and (39):

$$\beta_{ij} = \frac{1}{\alpha_{ii}} \left[a_{ij} - \left(\alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + ... + \alpha_{i(i-1)}\beta_{(i-1)j} \right) \right] \xrightarrow{\alpha_{ii}=1} \beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik} \beta_{kj}$$

Secondly, calculate for $i = j+1, j+2, ..., n$

$$\alpha_{ij} = \frac{1}{\beta_{jj}} \left[a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik} \beta_{kj} \right] \qquad (42)$$

This formulation can be deduced from equation (40):

$$\alpha_{ij} = \frac{1}{\beta_{jj}} \left[a_{ij} - \left(\alpha_{i1}\beta_{1j} + \alpha_{i2}\beta_{2j} + ... + \alpha_{i(j-1)}\beta_{(j-1)j} \right) \right] \longrightarrow \alpha_{ij} = \frac{1}{\beta_{jj}} \left[a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik} \beta_{kj} \right]$$

By doing the above calculations by hand for a few elements reveals that all α's and β's needed for a certain calculation are already determined by the time they are needed. Furthermore it can be seen that every a_{ij} is used only once and never again. This means that the corresponding α_{ij} or β_{ij} can be stored in the location that a used to occupy. This is advantageous with the computer implementation in mind: only one array is needed to perform the decomposition, since $\underline{\underline{A}}$ is destroyed while simultaneously $\underline{\underline{LU}}$ is produced. Moreover, $\underline{\underline{L}}$ and $\underline{\underline{U}}$ can be stored in same matrix, because of their triangular forms:

$$\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \cdots & \beta_{1n} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \cdots & \beta_{2n} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \cdots & \beta_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{n1} & \alpha_{n2} & \alpha_{n3} & \cdots & \beta_{nn} \end{bmatrix}.$$

Since $\alpha_{ii} = 1$ for $i = 1, 2, ..., n$, these values do not need to be stored.

One more attention point is the stability of Crout's algorithm. Just like other numerical methods for solving systems of linear equations, pivoting is essential. This can be done by

picking the largest absolute value that occurs in a column. That is, only the element on the main diagonal or the elements below may be selected. If this element is not on the main diagonal, the corresponding rows should be swapped. This procedure is called partial pivoting. Of course, if rows in matrix $\underline{\underline{A}}$ are swapped, the same should be done to vector $\underline{b}$. It is important to notice that in the case of $i = j$ equation (41) is exactly the same as equation (42) except for the division by β_{jj} in the latter equation. The upper limit of the sum is in both cases equal to $j-1$.
Choosing the pivot element by simply picking the largest absolute value is obviously not the best way, because scaling a certain row is always allowed. If for example a certain row is multiplied by a million, then it is almost certain that this row will deliver the pivot element. To avoid this problem use is made of implicit pivoting. The idea of implicit pivoting is as follows: calculate per row the scaling factor that is needed to obtain 1 as the largest absolute coefficient. Then calculate per element the product of this scaling factor and the real value of the element. Picking the largest absolute value after performing this operation gives a good choice for the pivot element. It is recalled that only elements on the main diagonal or in the same column below it can be chosen. Choosing an element above the main diagonal element destroys the already formed part of the decomposed matrix. If the largest element is not on the main diagonal, the two corresponding rows have to be swapped. Now knowing the value of the pivot element, the division from equation (42) can be carried out.

Solving for vector $\underline{y}$

Now that matrix $\underline{\underline{A}}$ has been decomposed in matrices $\underline{\underline{L}}$ and $\underline{\underline{U}}$, use can be made of the fact that triangular matrices can be solved in a simple way. Writing $\underline{\underline{L}} \cdot \underline{y} = \underline{b}$ gives:

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & \cdots & 0 \\ \alpha_{21} & \alpha_{22} & 0 & \cdots & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{n1} & \alpha_{n2} & \alpha_{n3} & \cdots & \alpha_{nn} \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix} \qquad (43)$$

These equations can be solved by forward substitution:

$$y_1 = \frac{b_1}{\alpha_{11}},$$

$$y_2 = \frac{b_2 - \alpha_{21} y_1}{\alpha_{22}} = \frac{1}{\alpha_{22}}\left[b_2 - \alpha_{21} y_1\right],$$

$$y_3 = \frac{b_3 - \alpha_{31} y_1 - \alpha_{32} y_2}{\alpha_{33}} = \frac{1}{\alpha_{33}}\left[b_3 - \left(\alpha_{31} y_1 + \alpha_{32} y_2\right)\right],$$

and so forth.
In general, for element y_i it holds that:

$$y_i = \frac{1}{\alpha_{ii}}\left[b_i - \sum_{j=1}^{i-1} \alpha_{ij} y_j \right] \text{ for } i = 1,2,...,n .$$

Solving for vector $\underline{x}$

In a similar manner as for the system $\underline{\underline{L}} \cdot \underline{y} = \underline{b}$, the system $\underline{\underline{U}} \cdot \underline{x} = \underline{y}$ can be solved. But this time the procedure is called backward substitution. Writing $\underline{\underline{U}} \cdot \underline{x} = \underline{y}$ gives:

$$\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \cdots & \beta_{1n} \\ 0 & \beta_{22} & \beta_{23} & \cdots & \beta_{2n} \\ 0 & 0 & \beta_{33} & \cdots & \beta_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \beta_{nn} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$

Vector $\underline{x}$ can now be solved as follows:

$$x_n = \frac{y_n}{\beta_{nn}},$$

$$x_{n-1} = \frac{y_{n-1}\beta_{(n-1)n}x_n}{\beta_{(n-1)(n-1)}} = \frac{1}{\beta_{(n-1)(n-1)}}\left[y_{n-1} - \beta_{(n-1)n}x_n \right],$$

$$x_{n-2} = \frac{y_{n-2} - \beta_{(n-2)(n-1)}x_{n-1} - \beta_{(n-2)n}x_n}{\beta_{(n-2)(n-2)}} = \frac{1}{\beta_{(n-2)(n-2)}}\left[y_{n-2} - \left(\beta_{(n-2)(n-1)}x_{n-1} + \beta_{(n-2)n}x_n \right) \right],$$

and so forth.
In general, for element x_i it holds that:

$$x_i = \frac{1}{\beta_{ii}}\left[y_i - \sum_{j=i+1}^{n} \beta_{ij} x_j \right] \text{ for } i = n-1, n-2, ..., 1 .$$

Source code of class `LU_Decomposition.java`

The solution procedure has been implemented as follows:

```
// include this source file to the linear algebra package

package anne.linalg;


final public class LU_Decomposition
{

    public static DoubleVector solve (DoubleMatrix A, DoubleVector b)
    {

        // constants declaration

        final int N;    // number of rows of the square matrix
        final double TINY_NUMBER = 1.0E-20;


        // variable declaration
```

```
int pivotRow = 0;
DoubleMatrix LU;              // decomposed matrix
DoubleVector y;               // solution vector
DoubleVector scaleFactors;    // vector which stores the implicit
                              // scaling of each row of A


// check the dimensions of matrix A and vector b

if (A.getNumRows () != A.getNumCols ())
{

    throw new RuntimeException ("Error: matrix in module
                                  LU_Decomposition is not square.");

}


if (A.getNumCols () != b.getNumEntries ())
{

    throw new RuntimeException ("Error: number of columns in matrix
                                  does not match number of entries in
                                  vector in module
                                  LU_Decomposition.");

}


// make a copy of matrix A and vector b

LU = new DoubleMatrix (A);
y = new DoubleVector (b);


// create an empty scale factor vector

N = y.getNumEntries ();
scaleFactors = new DoubleVector (N);


// calculate implicit scale factor for each row of A

for (int i = 0; i < N; i++)
{

    double largest = 0.0;


    for (int j = 0; j < N; j++)
    {

        double temp = Math.abs (LU.getEntry (i, j));
        if (temp > largest)
        {

            largest = temp;

        }

    }


    if (largest == 0.0)
    {

        throw new RuntimeException ("Error: matrix in module
                                      LU_Decomposition is
                                      singular.");

    }
```

```
    else
    {

        double temp = 1.0 / largest;
        scaleFactors.setEntry (i, temp);

    }

}


// perform decomposition of A in L and U

for (int j = 0; j < N; j++)
{

    for (int i = 0; i < j; i++)
    {

        double sum = LU.getEntry (i, j);


        for (int k = 0; k < i; k++)
        {

            sum = sum - LU.getEntry (i, k) * LU.getEntry (k, j);

        }


        LU.setEntry (i, j, sum);

    }


    double largest = 0.0;


    for (int i = j; i < N; i++)
    {

        double sum = LU.getEntry (i, j);


        for (int k = 0; k < j; k++)
        {

            sum = sum - LU.getEntry (i, k) * LU.getEntry (k, j);

        }


        LU.setEntry (i, j, sum);
        double temp = scaleFactors.getEntry (i) * Math.abs (sum);


        if (temp >= largest)
        {

            largest = temp;
            pivotRow = i;

        }

    }


    if (j != pivotRow)
    {

        LU = LU.swapRows (pivotRow, j);
```

```
            y = y.swapEntries (pivotRow, j);

        }


        if (LU.getEntry (j, j) == 0.0)
        {

            LU.setEntry (j, j, TINY_NUMBER);

        }


        if (j != (N - 1))
        {

            double temp = 1.0 / LU.getEntry (j, j);


            for (int i = (j + 1); i < N; i++)
            {

                double temp2 = LU.getEntry (i, j) * temp;
                LU.setEntry (i, j, temp2);

            }

        }

    }


    // calculate solution vector

    for (int i = 0; i < N; i++)
    {

        double sum = y.getEntry (i);


        for (int j = 0; j <= (i - 1); j++)
        {

            sum = sum - LU.getEntry (i, j) * y.getEntry (j);

        }


        y.setEntry(i, sum);

    }


    for (int i = (N - 1); i >= 0; i--)
    {

        double sum = y.getEntry (i);


        for (int j = (i + 1); j < N; j++)
        {

            sum = sum - LU.getEntry (i, j) * y.getEntry (j);

        }


        double temp = sum / LU.getEntry (i, i);
        y.setEntry (i, temp);

    }
```

```
        return y;
    }
}
```

Testing module `LU_Decomposition`

The module `LU_Decomposition` was tested by calculating example systems of linear equations taken from literature. Two small examples are given here. The first example is taken from Lay (2000) [7]. Consider the system $\underline{\underline{A}} \cdot \underline{x} = \underline{b}$, where

$$\underline{\underline{A}} = \begin{bmatrix} 1 & -2 & 1 \\ 0 & 2 & -8 \\ -4 & 5 & 9 \end{bmatrix} \text{ and } \underline{b} = \begin{bmatrix} 0 \\ 8 \\ -9 \end{bmatrix}.$$

The module `LU_Decomposition` gives the following solution

$$\underline{x} = \begin{bmatrix} 29.000000000000018 \\ 16.00000000000001 \\ 3.000000000000027 \end{bmatrix},$$

which is correct accepting a small round-off error.
The second example (taken from [19]) tests partial pivoting explicitly. Let

$$\underline{\underline{A}} = \begin{bmatrix} 10^{-4} & 1 \\ 1 & 1 \end{bmatrix} \text{ and } \underline{b} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

and consider the system $\underline{\underline{A}} \cdot \underline{x} = \underline{b}$. The solution provided by the module is

$$\underline{x} = \begin{bmatrix} 1.0001000100010002 \\ 0.9998999899989999 \end{bmatrix},$$

which is correct and proves that the used algorithm is stable [19].

Appendix C: Matrix and vector classes in Java

This appendix provides the source code for the classes `DoubleMatrix`, `DoubleVector`, `IntMatrix` and `IntVector`. Since variable types have to be specified inside a class, a distinction has been made in matrices which contain double-precision numbers and which contain integer numbers. The same applies to the `Vector` classes. The source code in this appendix is inspired by [18].

Source code of class `DoubleMatrix`

```
// include this source file to the linear algebra package

package anne.linalg;


final public class DoubleMatrix
{

    private final double[][] array;    // the 2D array
    private final int numRows;         // number of rows
    private final int numCols;         // number of columns



    // constructor for creating an empty matrix

    public DoubleMatrix(int numRows, int numCols)
    {

        this.numRows = numRows;
        this.numCols = numCols;
        this.array = new double[numRows][numCols];

    }



    // constructor for creating a matrix from an array

    public DoubleMatrix(double[][] matrix)
    {

        this.numRows = matrix.length;
        this.numCols = matrix[0].length;
        this.array = new double[numRows][numCols];
        for(int i = 0; i < numRows; i++)
        {

            for(int j = 0; j < numCols; j++)
            {

                this.array[i][j] = matrix[i][j];

            }

        }

    }



    // copy constructor
```

```
public DoubleMatrix (DoubleMatrix matrix)
{

    this.numRows = matrix.numRows;
    this.numCols = matrix.numCols;
    this.array = new double[numRows][numCols];
    for(int i = 0; i < numRows; i++)
    {

        for(int j = 0; j < numRows; j++)
        {

            this.array[i][j] = matrix.array[i][j];

        }

    }

}



// method for getting number of rows

public int getNumRows()
{

    return this.numRows;

}



// method for getting number of columns

public int getNumCols()
{

    return this.numCols;

}



// method for setting the value of an entry

public void setEntry(int i, int j, double value)
{

    this.array[i][j] = value;

}



// method for getting the value of an entry

public double getEntry(int i, int j)
{

    return this.array[i][j];

}



// method for writing the matrix to the standard output

public void showMatrix()
{
```

```
            for(int i = 0; i < this.numRows; i++)
            {

                for(int j = 0; j < this.numCols; j++)
                {

                    System.out.print(this.array[i][j] + "   ");

                }
                System.out.println();

            }

        }



        // method for interchanging two rows

        public DoubleMatrix swapRows(int i, int j)
        {

            DoubleMatrix matrix = new DoubleMatrix(this);
            double[] temp = matrix.array[i];
            matrix.array[i] = matrix.array[j];
            matrix.array[j] = temp;
            return matrix;

        }

    }
```

Source code of class `DoubleVector`

```
// include this source file to the linear algebra package

package anne.linalg;


final public class DoubleVector
{

    private final double[] array;     // the 1D array
    private final int numEntries;     // number of entries


    // constructor for creating a vector

    public DoubleVector(int numEntries)
    {

        this.numEntries = numEntries;
        this.array = new double[numEntries];

    }


    // constructor for creating a vector from an array

    public DoubleVector(double[] vector)
    {

        this.numEntries = vector.length;
        this.array = new double[numEntries];
        for(int i = 0; i < numEntries; i++)
        {

            this.array[i] = vector[i];

        }

    }


    // copy constructor

    public DoubleVector(DoubleVector vector)
    {

        this.numEntries = vector.numEntries;
        this.array = new double[numEntries];
        for(int i = 0; i < numEntries; i++)
        {

            this.array[i] = vector.array[i];

        }

    }


    // method for getting number of entries

    public int getNumEntries()
    {

        return this.numEntries;
```

```
    }


    // method for setting the value of an entry

    public void setEntry(int i, double value)
    {

        this.array[i] = value;

    }


    // method for getting the value of an entry

    public double getEntry(int i)
    {

        return this.array[i];

    }


    // method for writing the vector to the standard output

    public void showVector()
    {

        for(int i = 0; i < this.numEntries; i++)
        {

            System.out.println(this.array[i]);

        }

    }


    // method for interchanging two entries

    public DoubleVector swapEntries(int i, int j)
    {

        DoubleVector vector = new DoubleVector(this);
        double temp = vector.array[i];
        vector.array[i] = vector.array[j];
        vector.array[j] = temp;
        return vector;

    }

}
```

Source code of class `IntMatrix`

```
// include this source file to the linear algebra package

package anne.linalg;



final public class IntMatrix
{

    private final int[][] array;    // the 2D array
    private final int numRows;      // number of rows
    private final int numCols;      // number of columns



    // constructor for creating an empty matrix

    public IntMatrix(int numRows, int numCols)
    {

        this.numRows = numRows;
        this.numCols = numCols;
        this.array = new int[numRows][numCols];

    }



    // constructor for creating a matrix from an array

    public IntMatrix(int[][] matrix)
    {

        this.numRows = matrix.length;
        this.numCols = matrix[0].length;
        this.array = new int[numRows][numCols];
        for(int i = 0; i < numRows; i++)
        {

            for(int j = 0; j < numCols; j++)
            {

                this.array[i][j] = matrix[i][j];

            }

        }

    }



    // copy constructor

    public IntMatrix (IntMatrix matrix)
    {

        this.numRows = matrix.numRows;
        this.numCols = matrix.numCols;
        this.array = new int[numRows][numCols];
        for(int i = 0; i < numRows; i++)
        {

            for(int j = 0; j < numRows; j++)
            {

                this.array[i][j] = matrix.array[i][j];
```

```
        }

    }

}


// method for getting number of rows

public int getNumRows()
{

    return this.numRows;

}


// method for getting number of columns

public int getNumCols()
{

    return this.numCols;

}


// method for setting the value of an entry

public void setEntry(int i, int j, int value)
{

    this.array[i][j] = value;

}


// method for getting the value of an entry

public int getEntry(int i, int j)
{

    return this.array[i][j];

}


// method for writing the matrix to the standard output

public void showMatrix()
{

    for(int i = 0; i < this.numRows; i++)
    {

        for(int j = 0; j < this.numCols; j++)
        {

            System.out.print(this.array[i][j] + "   ");

        }
        System.out.println();

    }

}
```

```
    // method for interchanging two rows

    public IntMatrix swapRows(int i, int j)
    {

        IntMatrix matrix = new IntMatrix(this);
        int[] temp = matrix.array[i];
        matrix.array[i] = matrix.array[j];
        matrix.array[j] = temp;
        return matrix;

    }

}
```

Source code of class `IntVector`

```
// include this source file to the linear algebra package

package anne.linalg;



final public class IntVector
{

    private final int[] array;        // the 1D array
    private final int numEntries;     // number of entries



    // constructor for creating a vector

    public IntVector(int numEntries)
    {

        this.numEntries = numEntries;
        this.array = new int[numEntries];

    }



    // constructor for creating a vector from an array

    public IntVector(int[] vector)
    {

        this.numEntries = vector.length;
        this.array = new int[numEntries];
        for(int i = 0; i < numEntries; i++)
        {

            this.array[i] = vector[i];

        }

    }



    // copy constructor

    public IntVector(IntVector vector)
    {

        this.numEntries = vector.numEntries;
        this.array = new int[numEntries];
        for(int i = 0; i < numEntries; i++)
        {

            this.array[i] = vector.array[i];

        }

    }



    // method for getting number of entries

    public int getNumEntries()
    {

        return this.numEntries;
```

```
    }


    // method for setting the value of an entry

    public void setEntry(int i, int value)
    {

        this.array[i] = value;

    }


    // method for getting the value of an entry

    public int getEntry(int i)
    {

        return this.array[i];

    }


    // method for writing the vector to the standard output

    public void showVector()
    {

        for(int i = 0; i < this.numEntries; i++)
        {

            System.out.println(this.array[i]);

        }

    }


    // method for interchanging two entries

    public IntVector swapEntries(int i, int j)
    {

        IntVector vector = new IntVector(this);
        int temp = vector.array[i];
        vector.array[i] = vector.array[j];
        vector.array[j] = temp;
        return vector;

    }

}
```