Concrete column failure analysis using FDM

A FDM analysis of the bearing capacity of a conrete column in relation to the bearing distance

L.W. Bleker - 4452151

Technische Universiteit Delft

TU Delft Delft University of Technology

Challenge the future

Contents

1

Introduction

Multiple different bearing systems exist for when a concrete column has to support a load. For some of these bearing types, the load is applied at a certain distance from the column edge, the ”bearing distance”. When this distance becomes too small a failure mechanism can occur where part of the top corner of the concrete column breaks off where no reinforcement is present. As of writing this paper no accurate model exists to describe this failure mechanism. The aim of this paper is to gain more understanding of the influence of different variables on the capacity of the column head with regards to this failure mechanism. The following research question has been formulated:

”How does the bearing capacity of a concrete column relate to the dimensioning and positioning of the bearing?”

The preceding paper to this tried to answer a similar research question with the use of equilibrium conditions. The conclusion was however that too many assumptions had to be made to make the model feasible [1]. This is why for this paper a different approach is chosen, namely a finite difference discretisation.

The structure of this paper is as follows. Chapter 2 will discuss the method used and the way in which it is implemented. Chapter 3 contains the results. Finally, a conclusion is made in chapter 4.

2 Method

2.1. Model definition

To make the problem easier to solve and simpler to analyse, the 3-dimensional problem is reduced to a 2-dimensional one. This is done by taking a vertical cross-section of the column along the width of the bearing. For this reduction in dimension the simplification has to be made that the column stretches to infinity in both directions perpendicular to the cross-section, which is of course not the case. It is assumed however that this difference with the 3-dimensional problem is very small and therefore does not have a significant impact on the results. In Figure 2.1 the bearing load is shown.

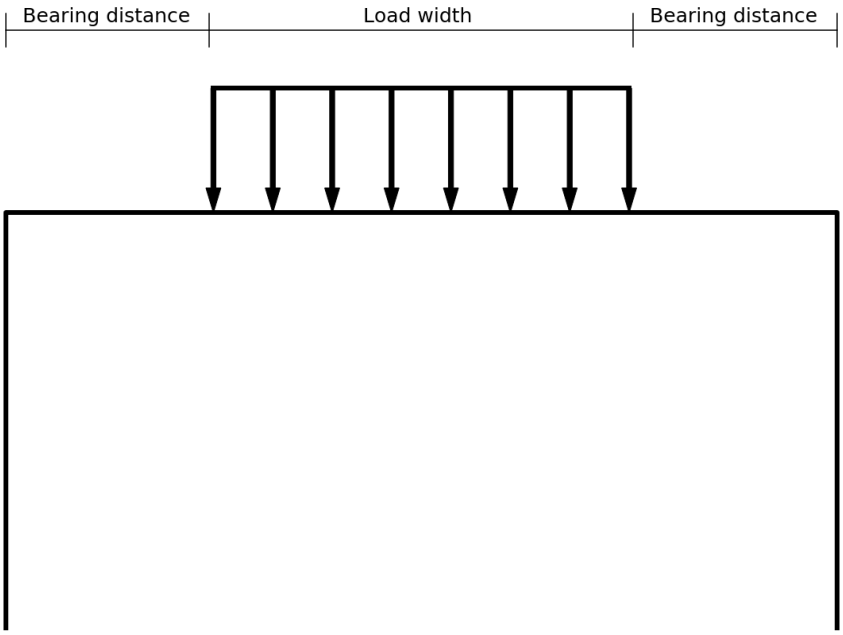

Figure 2.1: Model representing the bearing load

The load has been placed symmetrically on the column so that any bending moments are avoided. This way the failure mechanism of interest is isolated as much as possible. Along the entire width of the base of the column vertical and horizontal forces are resisted to simulate a clamped column.

2.2. Discretisation

Because it is unknown where in the cross-section exactly the first crack appears it is very useful to get an image of the stress-distribution. For this, a discretisation method is very well suited. The problem

is simple in the sense that the geometric form of the cross-section is always rectangular. Therefore the finite difference method should suffice and is even preferred over the inherently more complicated finite element method (which does allow for more complicated shapes).

The starting point for the discretisation are the following differntial equations [2]:

$$-Et\left(\frac{\partial^2 u_x}{\partial x^2} + \frac{1}{2}\frac{\partial^2 u_x}{\partial y^2} + \frac{1}{2}\frac{\partial^2 u_y}{\partial x \partial y}\right) = p_x$$

$$-Et\left(\frac{\partial^2 u_y}{\partial y^2} + \frac{1}{2}\frac{\partial^2 u_y}{\partial x^2} + \frac{1}{2}\frac{\partial^2 u_x}{\partial x \partial y}\right) = p_y$$

These equations are the result of substituting the kinematic, constitutive and equilibrium equations into one another. It describes the displacement field of a cross-section in terms of the external load p applied per unit area in both x and y direction. The Poison ratio has been set to zero in these equations to keep the discretisation as simple as possible.

To solve these equations the continuous displacement field of the cross-section is discretised into a finite number of displacements in x and y direction as seen in Figure 2.2.

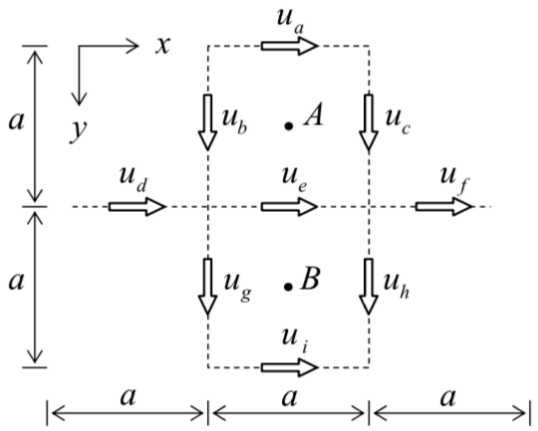

Figure 2.2: Displacement discretisation inside the cross-section

With the use of a central difference method the displacement derivatives can be written in terms of different displacements. Substitution leads to the following equation for the area force in x direction (for a more extensive derivation, see [3]):

$$\frac{Et}{a^2}\left(-\frac{1}{2}u_a - \frac{1}{2}u_b + \frac{1}{2}u_c - u_d + 3u_e - u_f + \frac{1}{2}u_g - \frac{1}{2}u_h - \frac{1}{2}u_i\right) = p_x$$

A graphical way of representing this equation is given in Figure 2.3. The number for each displacements represents the factor by which it has to be multiplied in the equation.

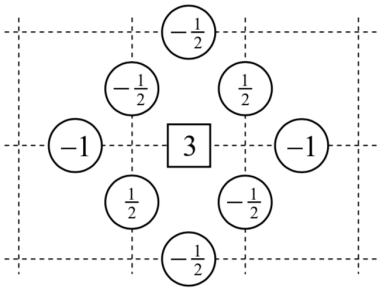

Figure 2.3: Molecule for displacement in x-direction

From Figure 2.3 it becomes apparent that near the borders this equation is not applicable because some of the neighboring displacements will be outside of the material. To overcome this, different molecules have to be derived for the edges and corners of the material. The way these molecules are derived, is by interpreting the displacements shown in Figure 2.2 as different springs and shear panels instead of applying the central difference theorem.

For the lower boundary of the column the displacement in both x and y direction is fixed over the entire length of the border. For these displacements the same molecule as shown in Figure 2.3 can be used, however u_g, u_h and u_i are all set to zero. Deriving the upper boundary molecules where the load is applied on the column would be slightly more complicated. To avoid this complication the load on top of the column is approximated as an area load on the cross-section in y direction (p_y) along the most upper row of molecules. By replacing the boundary load with an area load the same molecule for the free edges can be used for the edges where the load is present.

When every displacement (unknown) in the system has an accompanying molecule (equation), a square matrix can be created because there is an equal number of unknowns and equations. An example of such a matrix is given in Figure 2.4, which is the matrix for the discretisation shown in Figure 2.5. Every displacement in Figure 2.5 has a number which corresponds with the row number in Figure 2.4. The molecule of Figure 2.3, for example, is used in rows 11, 12, 20 and 21.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1	1	-0,5			0,5	-0,5				-0,5																					
2	-0,5	1,5	-0,5			0,5	-0,5				-0,5																				
3		-0,5	1,5	-0,5			0,5	-0,5				-0,5																			
4			-0,5	1				0,5	-0,5				-0,5																		
5	0,5				1	-0,5				-0,5				-0,5																	
6	-0,5	0,5			-0,5	2	-0,5			0,5	-0,5				-1																
7		-0,5	0,5			-0,5	2	-0,5			0,5	-0,5				-1															
8			-0,5	0,5			-0,5	2	-0,5			0,5	-0,5				-1														
9				-0,5				-0,5	1				0,5					-0,5													
10	-0,5				-0,5	0,5				2	-1			0,5	-0,5				-0,5												
11		-0,5				-0,5	0,5			-1	3	-1			0,5	-0,5				-0,5											
12			-0,5				-0,5	-0,5			-1	3	-1			0,5	-0,5				-0,5										
13				-0,5				-0,5	0,5			-1	2				0,5	-0,5				-0,5									
14					-0,5					0,5				1,5	-0,5				-0,5				-0,5								
15						-1				-0,5	0,5			-0,5	3	-0,5			0,5	-0,5				-1							
16							-1				-0,5	0,5			-0,5	3	-0,5			0,5	-0,5				-1						
17								-1				-0,5	0,5			-0,5	3	-0,5			0,5	-0,5				-1					
18									-0,5				-0,5				-0,5	1,5				0,5					-0,5				
19										-0,5				-0,5	0,5				2	-1			0,5	-0,5				-0,5			
20											-0,5				-0,5	0,5			-1	3	-1			0,5	-0,5				-0,5		
21												-0,5				-0,5	0,5			-1	3	-1			0,5	-0,5				-0,5	
22													-0,5				-0,5	0,5			-1	2				0,5	-0,5				-0,5
23														-0,5					0,5				1,5	-0,5				-0,5			
24															-1				-0,5	0,5			-0,5	3	-0,5			0,5	-0,5		
25																-1				-0,5	0,5			-0,5	3	-0,5			0,5	-0,5	
26																	-1				-0,5	0,5			-0,5	3	-0,5			0,5	-0,5
27																		-0,5				-0,5				-0,5	1,5				0,5
28																			-0,5				-0,5	0,5				2	-1		
29																				-0,5				-0,5	0,5			-1	3	-1	
30																					-0,5				-0,5	0,5			-1	3	-1
31																						-0,5				-0,5	0,5			-1	2

Figure 2.4: Stiffness matrix for the discretisation in 2.5

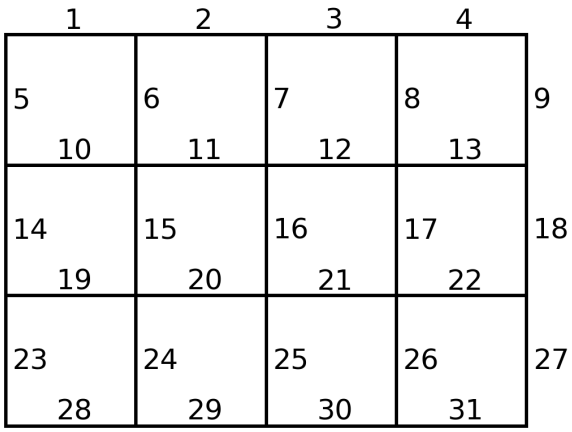

Figure 2.5: Example of a discretisation for a column with a width greater than its height

The matrix is created in a python program (see Appendix A) and stored as a sparse matrix, because of its diagonal nature. When the system is solved the result is one vector containing the different displacements of the discretisation. In Figure 2.6 the deformation of the boundary of a solved system is shown.

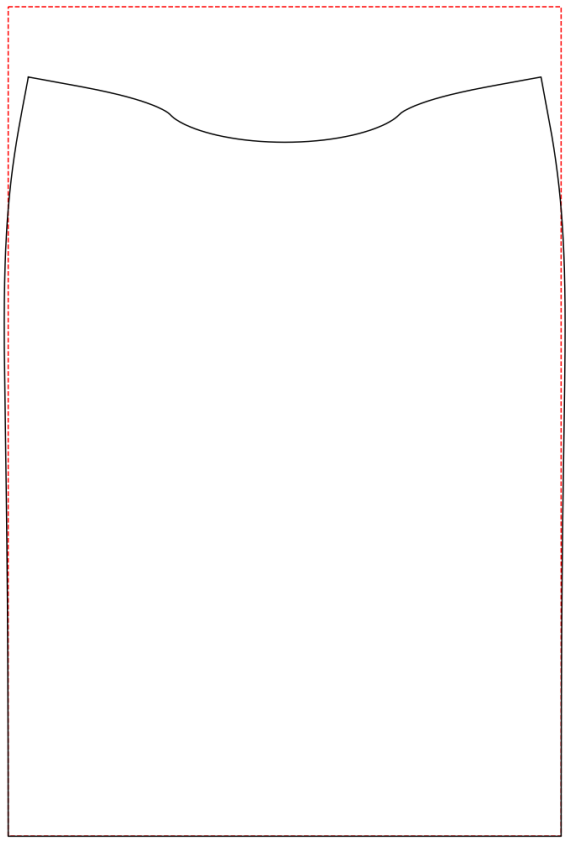

Figure 2.6: Example of a column deformation with an area load centrally applied on top with a gridsize of 400 by 600. (Red = Original, Black = Deformed)

2.3. Stress-distribution

From the deformation-field it is possible to derive the stress-distributions σ_{xx}, σ_{yy} and σ_{xy} with the following equations:

$$\sigma_{xx} = E\frac{\partial u_x}{\partial x}$$

$$\sigma_{yy} = E\frac{\partial u_y}{\partial y}$$

$$\sigma_{xy} = \frac{E}{2}(\frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x})$$

Again the central difference theorem is used to approximate the displacement derivatives. The stress distributions derived from the displacement field in Figure 2.6 are shown in Figure 2.7

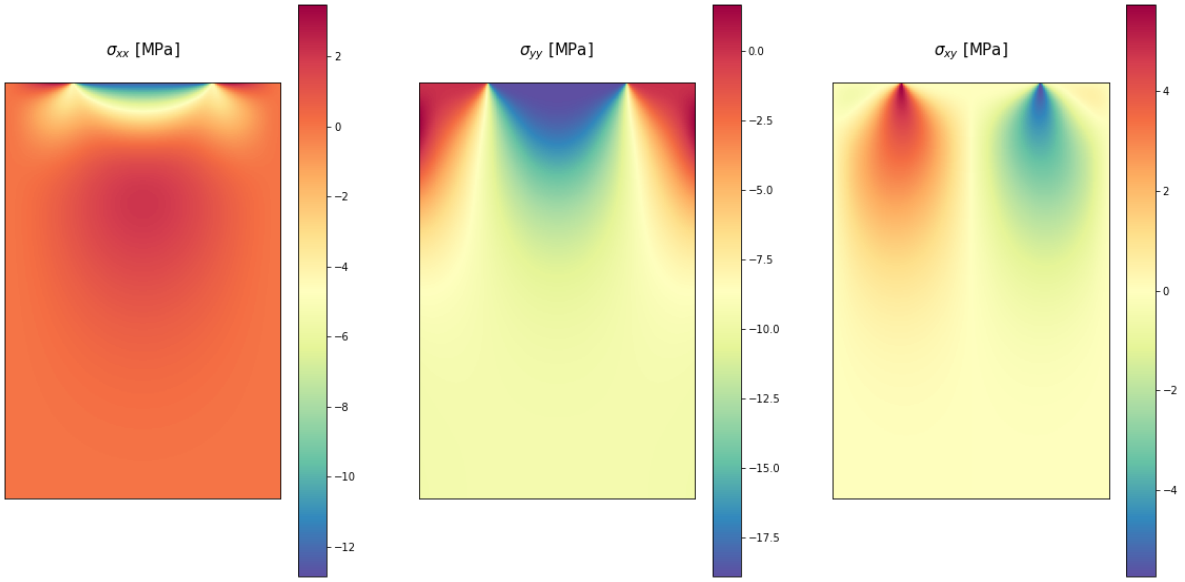

Figure 2.7: Stress distribution derived from the deformation in Figure 2.6

According to Mohr's circle the three different stress-distributions can be reduced to the two principal stresses with these equations:

$$\sigma_{1,2} = \frac{\sigma_{xx} + \sigma_{yy}}{2} \pm \sqrt{(\frac{\sigma_{xx} - \sigma_{yy}}{2})^2 + \sigma_{xy}^2}$$

The angle under which σ_1 and σ_2 act can be calculated according to the equations:

$$\alpha_1 = \frac{\arctan\frac{2\sigma_{xy}}{\sigma_{xx}}}{2}$$

$$\alpha_2 = \frac{\arctan\frac{2\sigma_{xy}}{\sigma_{xx}}}{2} + \frac{1}{2\pi}$$

In Figure 2.8 the stress distributions of Figure 2.7 have been converted to the two principal stress distributions. Figure 2.9 shows the principal stress trajectories belonging to the stress distributions in Figure 2.8.

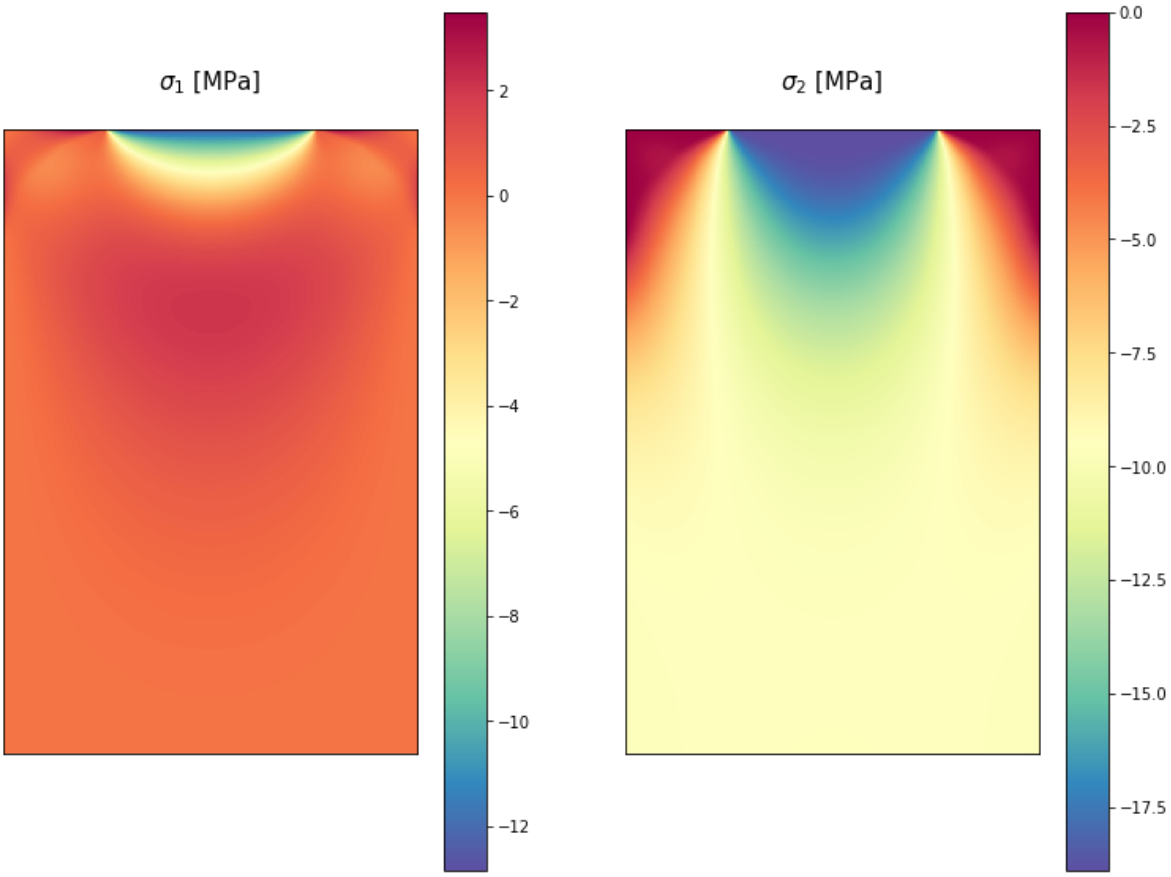

Figure 2.8: Principal stress-distributions derived from the stress-distributions in Figure 2.7

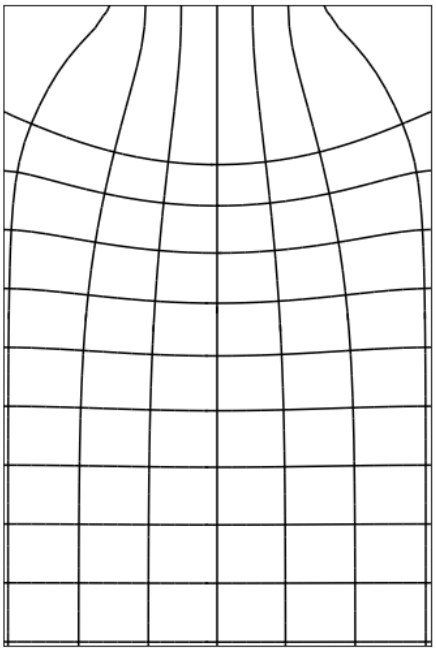

Figure 2.9: Principal stress trajectories for the principal stresses in Figure 2.8

3

Results

In this chapter the location in the column where the first crack appears is examined. Furthermore several variables of the model are studied one by one to see how they influence the magnitude of the maximum occurring internal stress levels. The variables to be studied are the element size, the bearing distance, the load magnitude as well as the load width.

3.1. Crack location

In Figure 3.1 the principal stress-distributions of a certain load case are shown. The colors are scaled in such a way that only tensile forces are shown and all compressive forces are blue.

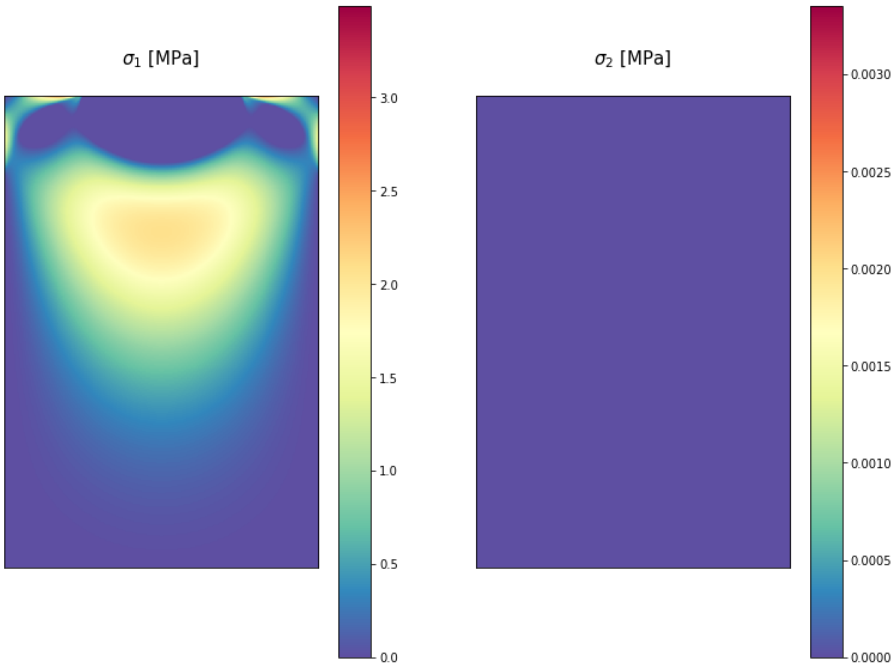

Figure 3.1: Principal stress-distributions scaled to only show tensile forces

From Figure 3.1 it becomes clear tensile forces only occur in σ_1 and not in σ_2. There are five tensile areas within the cross-section. The first area is a tensile tie at the center of the column. This tie is caused by the redistribution of the more concentrated compressive stress at the top of the column to an even distribution at the base of the column. In this area reinforcement is usually applied applied and therefore any cracks in the concrete do not cause any damage. The other 4 tensile areas are located near the corners of the column. The tensile forces at the top border are greater than those on

the side, therefore this is where the first crack appears. The single element with the highest tensile force is located on the uppermost row, on the edge of the concrete. Because this is a free edge $\sigma_y y$ is per definition equal to zero. Therefore the crack tensile force is in x direction and thus the crack forms perpendicular to the tensile force, straight down in y-direction. It is however likely the crack will quickly bend towards the side of the column since that is where more tensile forces are present, as can be seen in Figure 3.1. This would also correspond with real-life damage cases, where the top corners of the column break off.

3.2. Element size

Figure 3.2 displays a fit through several data points of different element sizes with corresponding maxima in the internal stress.

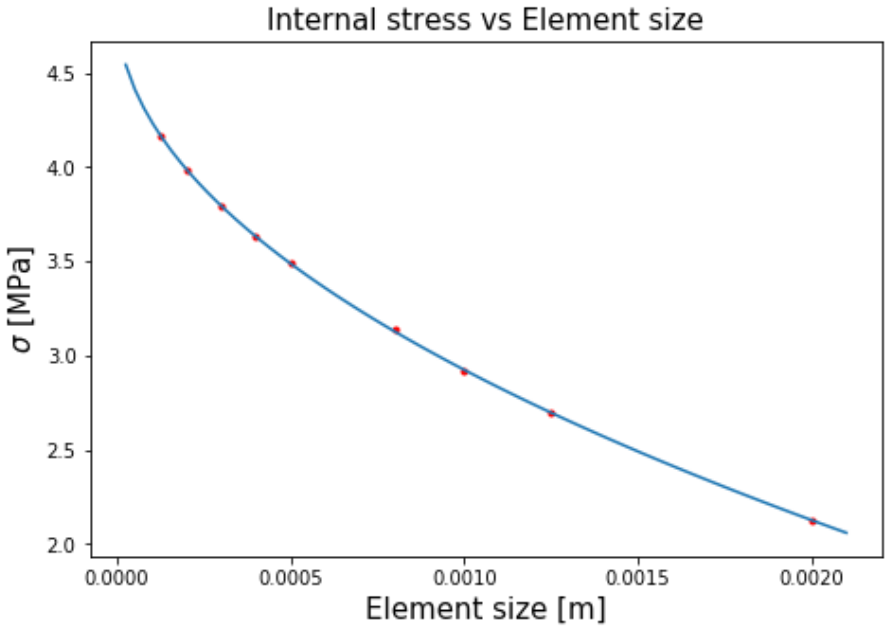

Figure 3.2

From Figure 3.2 it is immediately apparent smaller element sizes significantly increase the maximum occurring stress level. It should also be noted the second smallest element size (0.0002 m) takes around 10 minutes to compute, whilst the single smallest element size (0.000125 m) takes approximately 90 minutes to compute. This drastic increase in computation time makes this simulation unfit to compute models with smaller time steps.

The difference between the stress level of the smallest element size and the second smallest element size is too large to be able to predict the actual stress level. Because of this fact all further analyses in this chapter will only present relative changes in stress levels and no absolute cracking force will be discussed.

3.3. Load stress

Figure 3.3 shows a graph of the maximum internal stress versus the external load pressure. All data points have been generated with the same element size, load with and bearing distance.

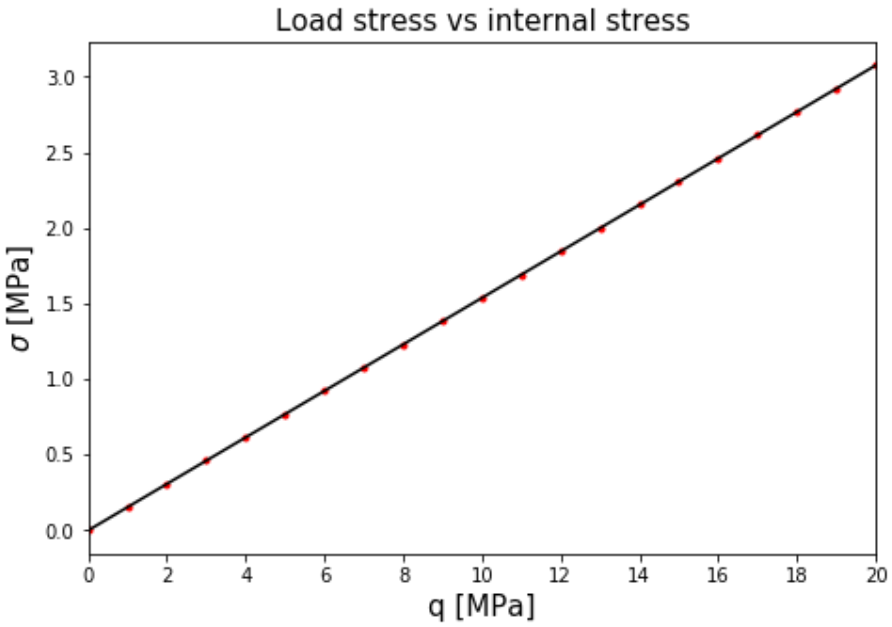

Figure 3.3

The graph shows a linear relationship between the maximum internal stress and the external pressure. This result is to be expected because of the simulation is entirely linear elastic.

3.4. Bearing distance

Figure 3.4 presents multiple data points where for each simulation the bearing distance is changed while keeping the load width and magnitude the same. Because the bearing distance changes and the load width stays constant, this implies that the entire width of the column for each simulation is different. The unit of the bearing distance (or edge distance), is given as how many times it is the width of the load. For example a bearing distance of 0.5 would mean the bearing distance is equal to half the width of the load.

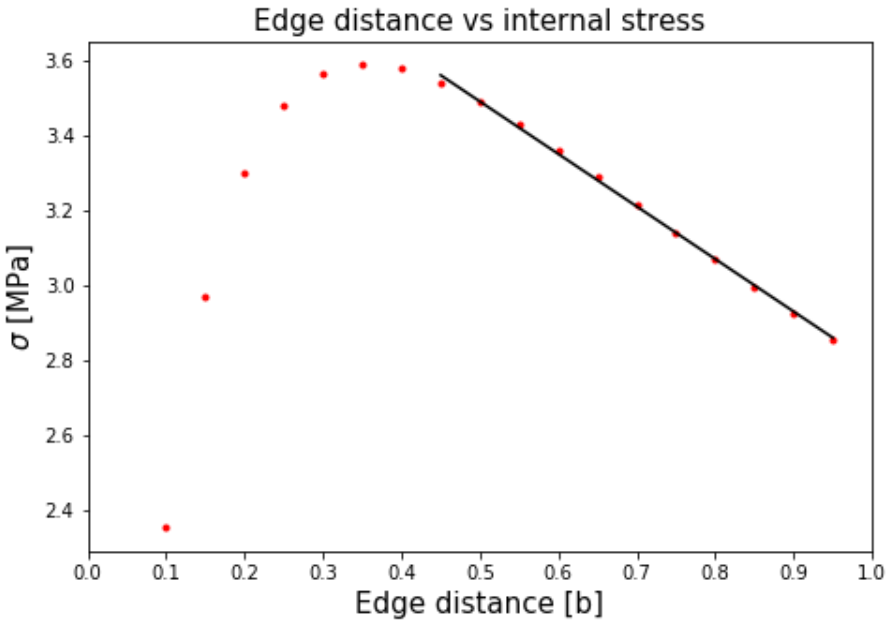

Figure 3.4

For large bearing distances, there is a linear relationship between the internal stress and the bearing distance. As the bearing distance becomes smaller the increase in stress decreases, and eventually reaches a maximum. For even closer bearing distances the stress quickly drops off to zero, as would be expected (when the bearing distance is zero, the entire column is under uniform pressure). For what bearing distance the maximum internal stress is reached, is however very dependant on the chosen element size. For smaller element sizes, the maximum is closer to a bearing distance of zero.

3.5. Load width

In this section two ways of changing the load width are discussed. First a case where as the load gets wider, the load pressure stays the same. Because the load pressure is constant and the load gets wider, the total force becomes larger. The second case is where as the load gets wider, the total force is kept constant. This implies that for wider load cases, the load pressure is lower. For both cases the change in load width and the constant bearing distance implies the total column width also changes between the simulations.

3.5.1. Constant stress

Figure 3.5 shows the load width with a constant load pressure plotted against the maximum internal stress.

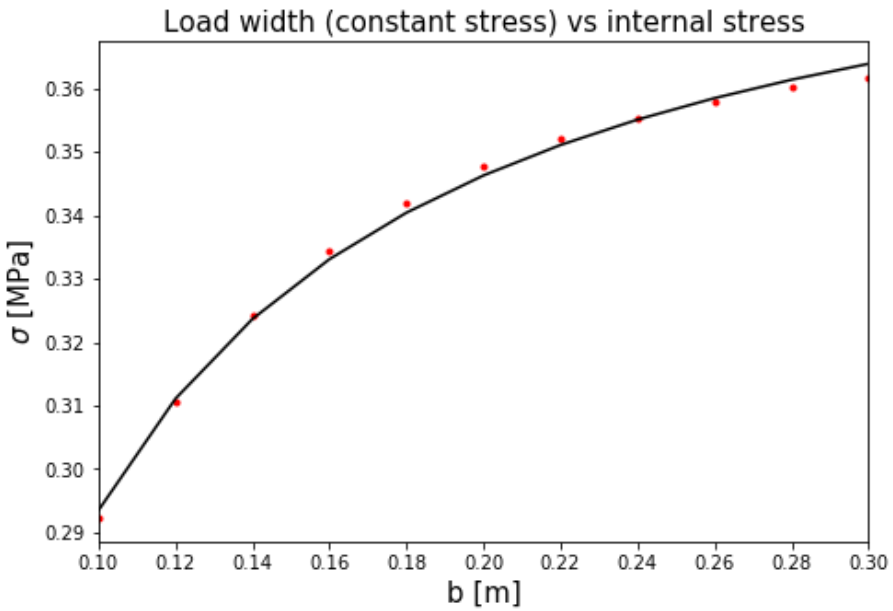

Figure 3.5

The Figure shows that for an increase in load width the maximum observed internal stress also increases. The increase decreases as the width gets larger. This can be explained with the theory that parts of the load close to the edge have a large contribution to the maximum internal stress, and loads farther away from the edge have less of an impact. As the load gets wider the maximum internal stress nears a limit as newly added loads are too far away from the edge to have any significant impact.

3.5.2. Constant force

Figure 3.6 is similar to Figure 3.5 with the major difference being that the load pressure decreases as the load width gets larger to retain a constant total force.

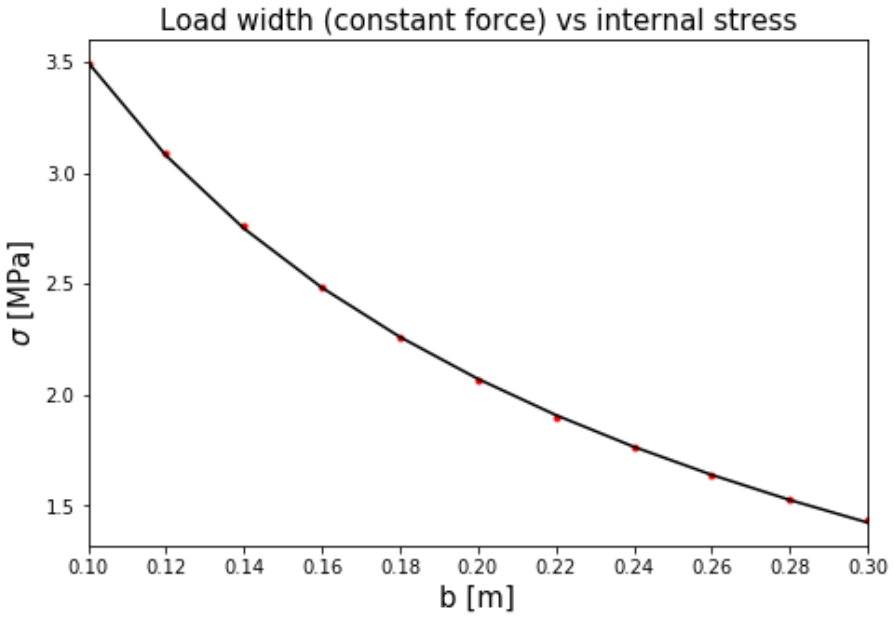

Figure 3.6

The graph shows a decrease in internal stress for a wider, more spread out load. This is in accordance to the previously made theory that loads closer to the edge have a larger impact on the internal stress than load farther away from the edge.

3.6. Practical case

The previous analyses isolated one variable to create an image of how only that variable influences the internal stress. The consequence of this is that the total width of the column changes in between simulations. When designing a bearing for a fixed size column those results are however not very practical. Figure 3.7 shows a more practical analysis where the column width as well as the total force stays constant.

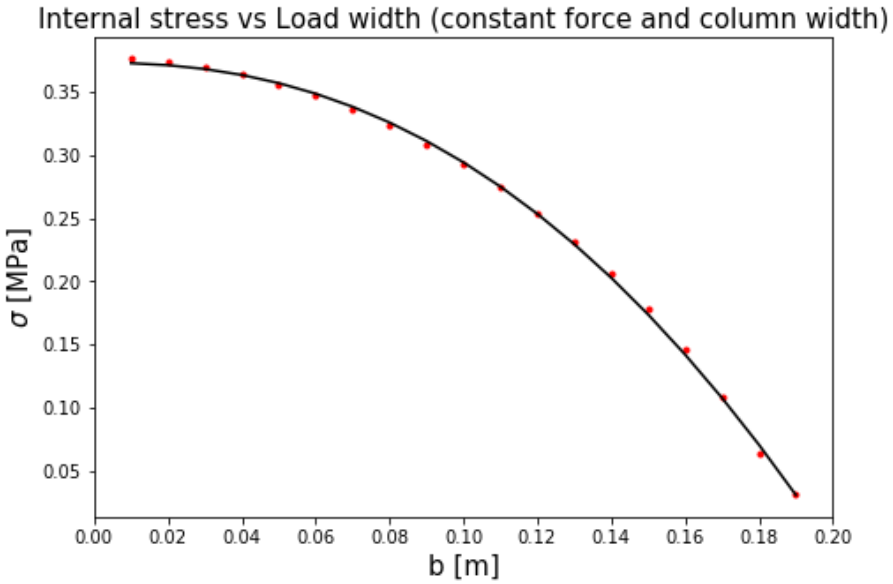

Figure 3.7: Column width = 0.2 m

It should be obvious the internal stress is zero when the load width is equal to the column width

(0.2 m). When the load width decreases the internal stress increases in a non-linear fashion. It should be noted however that when the load gets farther away from the column edge than the concrete cover is thick, these high stresses pose much less of a risk. When the concrete cracks farther away than the concrete cover, the crack would go through the reinforcement, and so the tensile forces resisted by the reinforcement.

4

Conclusion

The main research question to be answered in this paper was as follows:

"How does the bearing capacity of a concrete column relate to the dimensioning and positioning of the bearing?"

The magnitude of the tensile forces present in the cross-section are in the range of the tensile capacity of commonly used concrete types [4], so it is very much possible the column gets damaged due to the observed tensile forces. An increase in maximum observed tensile stress, therefore implies a decrease in the bearing capacity of the column. From the different parameter studies in chapter 3 the following design recommendation can be made to maximize the column bearing capacity.

- The bearing distance should either be (almost) equal to zero, or be made as large as possible.
- The bearing plate should have a big contact area with the column in order to spread the force in the column as even as possible.
- When a compromise has to be made between creating a larger bearing area or increasing the bearing distance, a larger bearing area is preferred. An exception to this is when the increase in bearing distance is large enough that the crack formation is far away enough from the concrete cover that that the crack would have to go though the reinforcement.

The created simulation is only useful for relative changes in the stress levels and not for calculating the maximum capacity. This is because of the large change in results for different element sizes. Perhaps a FEM model instead of the used FDM model would create more accurate results. An shortcoming of the simulation is the fact that when the tensile capacity of concrete is reached failure is immediately assumed, while it could be the case that only a small crack appears after which a new equilibrium is found. A way to be able to overcome this problem would be with a non-linear analysis instead.

Bibliography

[1] M. D. Klomp, *Concrete column support failure,* TU Delft , 22.

[2] J. Blauwendraad, *Plate analysis, theory and application volume 1, theory,* TU Delft , 7 (2006).

[3] J. Blauwendraad, *Plate analysis, theory and application volume 2, numerical methods,* TU Delft , 18 (2006).

[4] *En 1992-1-1,* (2004).

A

Python code

```
import numpy as np
import matplotlib.pyplot as plt
import numpy.linalg as lg
import scipy as sp
import scipy.sparse
from scipy.sparse.linalg import spsolve
from matplotlib import cm

def A_create(nx, ny, a, E, v, t):
    n = 2*nx*ny + nx + ny
    d0 = (3 - v) * np.ones(n)
    d0[0] = 1
    d0[1:nx - 1] = 1.5
    d0[nx - 1:nx + 1] = 1
    d0[nx + 1:2*nx] = 2
    d0[2*nx] = 1
    d0[2*nx + 1] = 2
    d0[3*nx] = 2
    for i in range(ny - 2):
        d0[3*nx + 1 + i*(2*nx + 1)] = 1.5
        d0[3*nx + nx + 1 + i*(2*nx + 1)] = 1.5
        d0[3*nx + nx + 2 + i*(2*nx + 1)] = 2
        d0[3*nx + 2*nx + 1 + i*(2*nx + 1)] = 2
    d0[-(2*nx + 1)] = 1.5
    d0[-(nx + 1)] = 1.5
    d0[-(nx)] = 2
    d0[-1] = 2

    d1_U = 0 * np.ones(n - 1)
    d1_U[0:nx - 1] = -0.5
    d1_U[nx:2*nx] = -0.5
    for i in range(ny - 1):
        d1_U[2*nx + 1 + i*(2*nx + 1):3*nx + i*(2*nx + 1)] = -1
        d1_U[3*nx + 1 + i*(2*nx + 1):4*nx + 1 + i*(2*nx + 1)] = -0.5 + 0.5*v
        d1_U[3*nx + 1 + i*(2*nx + 1)] = -0.5
    d1_U[-(nx-1):] = -1

    d2_U = (0.5 + 0.5*v) * np.ones(n - nx)
    d2_U[0:nx] = 0.5
```

```
d2_U[nx + 1:2*nx + 2] = 0.5
for i in range(ny):
    d2_U[nx + i*(2*nx + 1)] = 0
for i in range(ny - 1):
    d2_U[2*nx + 1 + i*(2*nx + 1)] = 0.5
    d2_U[3*nx + i*(2*nx + 1)] = 0.5
    d2_U[4*nx + 1 + i*(2*nx + 1)] = 0.5

d3_U = (-0.5 - 0.5*v) * np.ones(n - (nx + 1))
d3_U[0:2*nx] = -0.5
for i in range(ny - 1):
    d3_U[2*nx + i*(2*nx + 1)] = 0
    d3_U[2*nx + 1 + i*(2*nx + 1)] = -0.5
    d3_U[3*nx + i*(2*nx + 1)] = -0.5
    d3_U[3*nx + 1 + i*(2*nx + 1)] = -0.5

d4_U = (-0.5 + 0.5*v) * np.ones(n - (2*nx + 1))
d4_U[0:nx] = -0.5
for i in range(ny - 1):
    d4_U[nx + i*(2*nx + 1)] = -0.5
    d4_U[nx + 1 + i*(2*nx + 1):2*nx + i*(2*nx + 1)] = -1
    d4_U[2*nx + i*(2*nx + 1)] = -0.5
    d4_U[2*nx + 1 + i*(2*nx + 1)] = -0.5
    d4_U[3*nx + i*(2*nx + 1)] = -0.5

d1_L = 0 * np.ones(n - 1)
d1_L[0:nx - 1] = -0.5
d1_L[nx:2*nx] = -0.5
for i in range(ny - 1):
    d1_L[2*nx + 1 + i*(2*nx + 1):3*nx + i*(2*nx + 1)] = -1
    d1_L[3*nx + 1 + i*(2*nx + 1):4*nx + 1 + i*(2*nx + 1)] = -0.5 + 0.5*v
    d1_L[4*nx + i*(2*nx + 1)] = -0.5
d1_L[-(nx-1):] = -1

d2_L = (0.5 + 0.5*v) * np.ones(n - nx)
d2_L[0:nx] = 0.5
d2_L[nx + 1] = 0.5
d2_L[2*nx] = 0.5
for i in range(ny):
    d2_L[nx + i*(2*nx + 1)] = 0
for i in range(ny - 1):
    d2_L[2*nx + 1 + i*(2*nx + 1)] = 0.5
    d2_L[3*nx + 2 + i*(2*nx + 1)] = 0.5
    d2_L[4*nx + 1 + i*(2*nx + 1)] = 0.5

d3_L = (-0.5 - 0.5*v) * np.ones(n - (nx + 1))
d3_L[0:nx + 1] = -0.5
d3_L[2*nx - 1] = -0.5
for i in range(ny - 1):
    d3_L[2*nx + i*(2*nx + 1)] = 0
    d3_L[3*nx + i*(2*nx + 1)] = -0.5
    d3_L[3*nx + 1 + i*(2*nx + 1)] = -0.5
    d3_L[4*nx + i*(2*nx + 1)] = -0.5

d4_L = (-0.5 + 0.5*v) * np.ones(n - (2*nx + 1))
d4_L[0] = -0.5
```

```
    d4_L[nx - 1] = -0.5
    for i in range(ny - 1):
        d4_L[nx + i*(2*nx + 1)] = -0.5
        d4_L[nx + 1 + i*(2*nx + 1):2*nx + i*(2*nx + 1)] = -1
        d4_L[2*nx + i*(2*nx + 1)] = -0.5
        d4_L[2*nx + 1 + i*(2*nx + 1)] = -0.5
        d4_L[3*nx + i*(2*nx + 1)] = -0.5
    A = (E*t/((1 - v**2)*(a**2)))*\
    sp.sparse.diags([d0, d1_U, d1_L, d2_U, d2_L ,d3_U ,d3_L ,d4_U, d4_L],\
                    [0, 1, -1, nx, -nx, nx + 1, -(nx + 1), 2*nx + 1, -(2*nx + 1)],\
                    shape = (n, n), format = 'csc')
    return A

def B_create(nx, ny, a, t, q, q_d, q_h):
    n = 2*nx*ny + nx + ny
    B = 0 * np.ones(n)
    B[nx + int((nx + 1)*q_h/b + 0.5):nx + int((nx + 1)*(q_h + q_d)/b + 0.5)] =\
    q*q_d*t/((a**2)*len(B[nx + int((nx + 1)*q_h/b + 0.5):nx +\
                         int((nx + 1)*(q_h + q_d)/b + 0.5)]))
    return B

def solve(nx, ny, a, E, v, t, q, q_d, q_h):
    A = A_create(nx, ny, a, E, v, t)
    B = B_create(nx, ny, a, t, q, q_d, q_h)
    y = sp.sparse.linalg.spsolve(A, B)
    return y

v = 0 # Poisson ratio
E = 37 * 10**6 # Modulus of elasticity [kN/m^2]
t = 0.001 # Plate thickness [m]

b = 0.2
h = 0.3
a = 0.0005
nx = int(b/a + 0.5)
ny = int(h/a + 0.5)
q_d = 0.1
q_h = 0.05
q = 19000

sol = solve(nx, ny, a, E, v, t, q, q_d, q_h)
x = np.array(list(np.arange(0, b, a)) + list(b*np.ones(ny)) + list(np.arange(b, 0, -a))\
             + list(0*np.ones(ny)) + [0])
y = np.array(list(h*np.ones(nx)) + list(np.arange(h, 0, -a)) + list(0*np.ones(nx)) +\
             list(np.arange(0, h, a)) + [h])
dx = np.array([sol[0]] + list(0.5*sol[0:nx - 1] + 0.5*sol[1:nx]) +\
              list(sol[nx-1::2*nx + 1]) + \
              list(0.5*sol[-2:-(nx + 1):-1] + 0.5*sol[-1:-(nx):-1]) +\
              list(sol[-nx::-(2*nx + 1)]))
dy = np.array(list(sol[nx:2*nx + 1]) + list(0.5*sol[4*nx + 1::2*nx + 1] +\
              0.5*sol[2*nx:-(nx + 1):2*nx + 1]) + \
              list(sol[-(nx + 1):-(2*nx + 2):-1]) + \
              list(0.5*sol[-(2*nx + 1):nx:-(2*nx + 1)] +\
                           0.5*sol[-(4*nx + 2)::-(2*nx + 1)]) + [sol[nx]])
plt.figure(figsize = (20,20))
plt.plot(x, y, 'r--')
```

```
plt.plot(x + 500*dx, y - 500*dy, 'k')
plt.axis('scaled');

def Sigma_Create(sol, nx, ny, a):
    Nxx = np.zeros((nx - 1)*(ny - 1))
    Nyy = np.zeros((nx - 1)*(ny - 1))
    Nxy = np.zeros((nx - 1)*(ny - 1))
    for i in range(len(Nxx)):
        Nu = sol[i + nx + 1 + int(i/(nx - 1))*(nx + 2)]
        Su = sol[i + 3*nx + 2 + int(i/(nx - 1))*(nx + 2)]
        Wu = sol[i + 2*nx + 1 + int(i/(nx - 1))*(nx + 2)]
        Eu = sol[i + 2*nx + 2 + int(i/(nx - 1))*(nx + 2)]
        NWu = sol[i + int(i/(nx - 1))*(nx + 2)]
        NEu = sol[i + 1 + int(i/(nx - 1))*(nx + 2)]
        ENu = sol[i + nx + 2 + int(i/(nx - 1))*(nx + 2)]
        ESu = sol[i + 3*nx + 3 + int(i/(nx - 1))*(nx + 2)]
        SWu = sol[i + 4*nx + 2 + int(i/(nx - 1))*(nx + 2)]
        SEu = sol[i + 4*nx + 3 + int(i/(nx - 1))*(nx + 2)]
        WNu = sol[i + nx + int(i/(nx - 1))*(nx + 2)]
        WSu = sol[i + 3*nx + 1 + int(i/(nx - 1))*(nx + 2)]
        Nxx[i] = (E/(1 - v**2))*((Eu - Wu)/a - v*(Su - Nu)/a)
        Nyy[i] = (E/(1 - v**2))*((Su - Nu)/a - v*(Eu - Wu)/a)
        Nxy[i] = (E/(2*(1 + v)))*(((SEu*0.5 + SWu*0.5) -\
           (NEu*0.5 + NWu*0.5))/(2*a) + ((ENu*0.5 + ESu*0.5) -\
           (WNu*0.5 + WSu*0.5))/(2*a))
    Nxx_M = np.zeros([ny - 1, nx - 1])
    Nyy_M = np.zeros([ny - 1, nx - 1])
    Nxy_M = np.zeros([ny - 1, nx - 1])
    P = 0
    for i in range(ny - 1):
        for j in range(nx - 1):
            Nxx_M[i, j] = Nxx[P]*0.001
            Nyy_M[i, j] = Nyy[P]*0.001
            Nxy_M[i, j] = Nxy[P]*0.001
            P += 1
    return Nxx_M, Nyy_M, Nxy_M
Nxx_M, Nyy_M, Nxy_M = Sigma_Create(sol, nx, ny, a)
plt.figure(figsize = (20, 10))
plt.subplot(131)
plt.title("$\sigma_{xx}$␣[MPa]", fontsize = 15)
plt.matshow(Nxx_M, fignum = 0, cmap = "Spectral_r")
plt.xticks([], [])
plt.yticks([], [])
plt.colorbar();
plt.subplot(132)
plt.title("$\sigma_{yy}$␣[MPa]", fontsize = 15)
plt.matshow(Nyy_M, fignum = 0, cmap = "Spectral_r")
plt.xticks([], [])
plt.yticks([], [])
plt.colorbar();
plt.subplot(133)
plt.title("$\sigma_{xy}$␣[MPa]", fontsize = 15)
plt.matshow(Nxy_M, fignum = 0, cmap = "Spectral_r")
plt.xticks([], [])
plt.yticks([], [])
plt.colorbar();
```

```
def Create_Principal_Sigma(Nxx_M, Nyy_M, Nxy_M):
    sigma1 = 0.5*Nxx_M + 0.5*Nyy_M + np.sqrt((0.5*Nxx_M - 0.5*Nyy_M)**2 + Nxy_M**2)
    sigma2 = 0.5*Nxx_M + 0.5*Nyy_M - np.sqrt((0.5*Nxx_M - 0.5*Nyy_M)**2 + Nxy_M**2)
    return sigma1, sigma2
sigma1, sigma2 = Create_Principal_Sigma(Nxx_M, Nyy_M, Nxy_M)
plt.figure(figsize = (20, 10))
plt.subplot(131)
plt.title("$\sigma_1$␣[MPa]", fontsize = 15)
plt.matshow(sigma1, fignum = 0, cmap = "Spectral_r")
plt.xticks([], [])
plt.yticks([], [])
plt.colorbar();
plt.subplot(132)
plt.title("$\sigma_2$␣[MPa]", fontsize = 15)
plt.matshow(sigma2, fignum = 0, cmap = "Spectral_r")
plt.xticks([], [])
plt.yticks([], [])
plt.colorbar();

plt.figure(figsize = (20, 10))
plt.subplot(131)
plt.title("$\sigma_1$␣[MPa]", fontsize = 15)
sigma1 = 0.5*Nxx_M + 0.5*Nyy_M + np.sqrt((0.5*Nxx_M - 0.5*Nyy_M)**2 + Nxy_M**2)
plt.matshow(sigma1, fignum = 0, cmap = "Spectral_r", vmin = 0)
plt.xticks([], [])
plt.yticks([], [])
plt.colorbar();
plt.subplot(132)
plt.title("$\sigma_2$␣[MPa]", fontsize = 15)
sigma2 = 0.5*Nxx_M + 0.5*Nyy_M - np.sqrt((0.5*Nxx_M - 0.5*Nyy_M)**2 + Nxy_M**2)
plt.matshow(sigma2, fignum = 0, cmap = "Spectral_r", vmin = 0)
plt.xticks([], [])
plt.yticks([], [])
plt.colorbar();

def Master(b, h, a, q_d, q_h, q):
    nx = int(b/a + 0.5)
    ny = int(h/a + 0.5)
    v = 0 # Poisson ratio
    E = 37 * 10**6 # Modulus of elasticity [kN/m^2]
    t = 0.001 # Plate thickness [m]
    sol = solve(nx, ny, a, E, v, t, q, q_d, q_h)
    Nxx_M, Nyy_M, Nxy_M = Sigma_Create(sol, nx, ny, a)
    sigma1, sigma2 = Create_Principal_Sigma(Nxx_M, Nyy_M, Nxy_M)
    result = sigma1[0].max()
    return result

a = 0.0005
q_d = 0.1
q = 20000

x = np.arange(0.01, 0.1, 0.005)
R = np.zeros(len(x))
for i in range(len(x)):
    q_h = x[i]
```

```
    b = 2*q_h + q_d
    h = 1.5*b
    R[i] = Master(b, h, a, q_d, q_h, q)
plt.plot(x, R, 'r.')
def func(x, a, b):
    y = a*x + b
    return y
popt, pcov = sp.optimize.curve_fit(func, x, R)
plt.plot(x, func(x, popt[0], popt[1]), 'k');

plt.figure(figsize = (7.5, 5))
x = np.arange(0.1, 1, 0.05)
plt.xticks(np.arange(0, 1.1, 0.1))
plt.xlim(0,1)
plt.plot(x, R, 'r.')
def func(x, a, b):
    y = a*x + b
    return y
plt.ylabel("$\sigma$ [MPa]", fontsize = 15)
plt.xlabel("Edge distance [b]", fontsize = 15)
popt, pcov = sp.optimize.curve_fit(func, x[(R.argmax() + 2):], R[(R.argmax() + 2):])
plt.title("Internal stress vs Edge distance", fontsize = 15)
plt.plot(x[(R.argmax() + 2):], func(x[(R.argmax() + 2):], popt[0], popt[1]), 'k');

a = 0.001
q_d = 0.1
q_h = 0.05
q = 20000

x = np.arange(0, 21000, 1000)
R = np.zeros(len(x))
for i in range(len(x)):
    q = x[i]
    b = 2*q_h + q_d
    h = 1.5*b
    R[i] = Master(b, h, a, q_d, q_h, q)
plt.plot(x, R, 'r.')
def func(x, a, b):
    y = a*x + b
    return y
popt, pcov = sp.optimize.curve_fit(func, x, R)
plt.plot(x, func(x, popt[0], popt[1]), 'k');

plt.figure(figsize = (7.5, 5))
x = np.arange(0, 21, 1)
plt.xticks(np.arange(0, 21, 2))
plt.xlim(0,20)
plt.plot(x, R, 'r.')
def func(x, a, b):
    y = a*x + b
    return y
plt.title("Internal stress vs Load stress", fontsize = 15)
plt.ylabel("$\sigma$ [MPa]", fontsize = 15)
plt.xlabel("q [MPa]", fontsize = 15)
popt, pcov = sp.optimize.curve_fit(func, x, R)
plt.plot(x, func(x, popt[0], popt[1]), 'k');
```

```
b = 0.15
h = 1.5*b
a = 0.0005
q_d = 0.1
q_h = 0.05
q = 20000

x = np.arange(0.1, 0.32, 0.02)
R = np.zeros(len(x))
for i in range(len(x)):
    q_d = x[i]
    q = 1900/q_d
    q_h = 0.05
    b = 2*q_h + q_d
    h = 1.5*b
    R[i] = Master(b, h, a, q_d, q_h, q)
plt.plot(x, R, 'r.')
def func(x, a, b):
    y = a*x + b
    return y
popt, pcov = sp.optimize.curve_fit(func, x, R)
plt.plot(x, func(x, popt[0], popt[1]), 'k');

plt.figure(figsize = (7.5, 5))
x = np.arange(0.1, 0.32, 0.02)
plt.xticks(x)
plt.xlim(0.1,0.3)
plt.plot(x, R, 'r.')
def func(x, a, b, c):
    y = a*x**c + b
    return y
plt.ylabel("$\sigma$ [MPa]", fontsize = 15)
plt.xlabel("b [m]", fontsize = 15)
plt.title("Internal stress vs Load width (constant force)", fontsize = 15)
popt, pcov = sp.optimize.curve_fit(func, x, R, maxfev = 8000)
plt.plot(x, func(x, popt[0], popt[1], popt[2]), 'k');

a = 0.001
q_h = 0.05
q = 20000

x = np.arange(0.1, 0.32, 0.02)
R = np.zeros(len(x))
for i in range(len(x)):
    q_d = x[i]
    b = 2*q_h + q_d
    h = 1.5*b
    R[i] = Master(b, h, a, q_d, q_h, q)
plt.plot(x, R, 'r.');

plt.figure(figsize = (7.5, 5))
x = np.arange(0.1, 0.32, 0.02)
plt.xticks(x)
plt.xlim(0.1,0.3)
plt.plot(x, R, 'r.')
```

```
def func(x, a, b):
    y = a/x + b
    return y
plt.ylabel("$\sigma$ [MPa]", fontsize = 15)
plt.xlabel("b [m]", fontsize = 15)
plt.title("Internal stress vs Load width (constant stress)", fontsize = 15)
popt, pcov = sp.optimize.curve_fit(func, x, R)
plt.plot(x, func(x, popt[0], popt[1]), 'k');

#Width of load (constant force) (distance = 0.1, load = 19000)
b = 0.15
h = 1.5*b
a = 0.001
q_d = 0.1
q_h = 0.05
q = 19000

x = np.arange(0.01, 0.2, 0.01)
R = np.zeros(len(x))
for i in range(len(x)):
    q_d = x[i]
    q = 0.1*1900/q_d
    b = 0.2
    q_h = 0.5*(b-q_d)
    h = 1.5*b
    print(b, h, a, q_d, q_h, q)
    R[i] = Master(b, h, a, q_d, q_h, q)
plt.plot(x, R, 'r.');

plt.figure(figsize = (7.5, 5))
x = np.arange(0.01, 0.2, 0.01)
plt.xticks(np.arange(0, 0.21, 0.02))
plt.xlim(0, 0.2)
plt.plot(x, R, 'r.')
def func(x, a, b, c):
    y = a*x**c + b
    return y
plt.ylabel("$\sigma$ [MPa]", fontsize = 15)
plt.xlabel("b [m]", fontsize = 15)
plt.title("Internal stress vs Load width (constant force and column width)",\
          fontsize = 15)
popt, pcov = sp.optimize.curve_fit(func, x, R)
plt.plot(x, func(x, popt[0], popt[1], popt[2]), 'k');
```