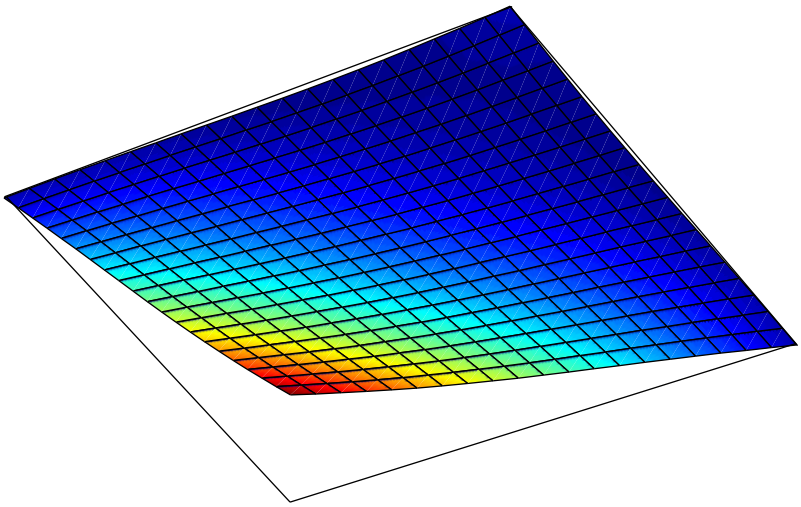
DELFT UNIVERSITY OF TECHNOLOGY

Computational Modeling of Cold Bent Glass Panels



Master Thesis Report

Jelle Matthijs de Wit

DELFT UNIVERSITY OF TECHNOLOGY

Computational Modeling of Cold Bent Glass Panels

by

Jelle Matthijs de Wit

A thesis submitted in partial fulfillment for the
degree of Master of Science

Civil Engineering and Geosciences
Department of Structural Engineering

April 2009

Preface

This document contains the thesis report: Computational Modeling of Cold Bent Glass Panels. The research is carried out at the Section Structural Mechanics of Delft University of Technology. I would like to thank the members of the examination committee for sharing their knowledge and for providing assistance during the realization of this report.

Delft, April 2009
Jelle de Wit

Examination committee:

Prof. Dr. Ir. J.G. Rots
Delft University of Technology - Faculty of Civil Engineering and Geosciences - Section Structural Mechanics
Dr. Ir. P.C.J. Hoogenboom
Delft University of Technology - Faculty of Civil Engineering and Geosciences - Section Structural Mechanics
Dr. Ir. P.H. Feenstra
Delft University of Technology - Faculty of Civil Engineering and Geosciences - Section Structural Mechanics
Dr. Ir. K.J. Vollers
Delft University of Technology - Faculty of Architecture - Section Building Technology
Ir. L.J.M. Houben
Delft University of Technology - Faculty of Civil Engineering and Geosciences - Section Road and Railway Engineering

Abstract

This thesis concerns the development of a finite element model for analyzing cold bent glass panels. The finite element model predicts the nonlinear deformations, stresses, and reaction forces which arise from bending a flat panel to a curved shape. The finite element model is implemented in the MATLAB programming language and subsequently deployed as a stand alone application called PBA (Plate Bending Application). The application is user friendly, demands little input, and requires short computation times. PBA is available on the Internet and can be used freely by others. For downloads see: http://www.mechanics.citg.tudelft.nl/pba/

A major part of the performed activities concentrated on developing a finite element formulation that describes the behavior of flat plates subjected to large displacements. To this end the classic membrane theory is combined with the Reissner-Mindlin bending theory. In order to solve the coupled membrane-bending formulation an iterative linear stiffness method is developed and examined for various influential parameters.

PBA is able to present accurate results for the nonlinear deformations, stresses, and reaction forces of moderately strong curved plates. For strongly curved plates, the implemented solution method does not find a converged solution. It is found that this occurs after transgression of realistic design criteria.

Contents

1. Introduction

1.1 Background

In modern architecture, complex building shapes are made possible due to the increasing use of advanced 3D computer modeling applications. These applications have accelerated the development of free form design, also known as fluid design, in which curved surfaces are an important aspect (Figure 1.1). The rapid upcoming of the modeling tools have led to an increasing gap between the designing party and the building industry [8]. Nonetheless, the building industry has stepped in and has recognized the wishes of modern architects to come up with solutions for making their designs possible.

Figure 1.1: Fluid design, City hall Alphen a/d Rijn (EEA architects)

The value of a building design is often judged by the appearance of its exterior, the facade. Commonly, a large portion of the facade is occupied by transparent parts made of glass. Considering free form designs often exhibit curved surfaces, the glass parts must somehow be formed to a prescribed shape. An approved method to achieve this is 'hot bending of glass panels' [11]. This method, also used in the automotive industry, relies on the ability of glass to deform into a desired shape when heated above the weakening point. A flat glass panel is heated up to 600 degrees Celsius and then pushed into a heat resistant mould. After reaching the desired shape, the panel is cooled down in a controlled manner, resulting in a tension free end product. Panels with very strong curvatures are feasible. When using this method to produce many different facade panels, as one has to do for a free form design, equally many different moulds have to be produced. As a consequence costs are very high and production time is long. Further, the quality of the hot formed panels is a point of discussion. Due to the heating process the thickness is affected, resulting in a decreased optical property and considerable production tolerances [17]. Although this technique has the advantage of making panels with very strong curvatures, it is not ideal for large scale projects in which many different panels are required.

An alternative method for developing curved glass surfaces is 'cold bending of glass panels' [11]. Hereby, flat glass panels are transported to the building site where they are forced into a curved

shape by cold bending. A frame or several fixed points are required to keep the panel in place, hence during its lifetime the panel is under constant stress. The advantage of this method compared to the hot bending method is the omitting of heating and moulding. As a result, costs are low by savings in time- and energy consumption. In addition, the optical quality is better and transportation of the (un-deformed) panels to the building site is more practical.

1.2 Problem description

In his master thesis, Staaks investigated the deformation path of a square flat panel deformed by increasing the displacement of one of the vertical constraints [15]. He observed a transition between an initial deformation and a final deformation. This is illustrated in Figure 1.2. The

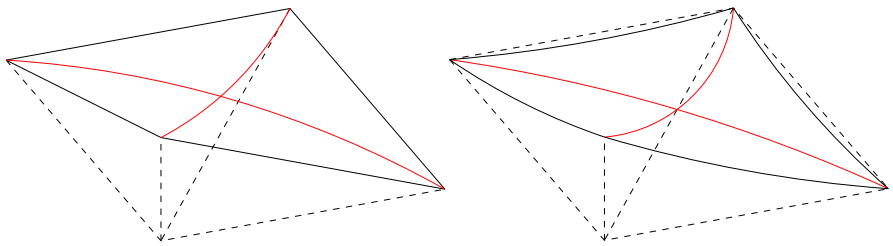


Figure 1.2: (a) First deformation pattern (b) Second deformation pattern

initial deformation pattern shows straight edges and curved diagonals. This shape is mathematically described by a hyperbolical. By increasing the vertical displacement, this double curved shape changes to a more or less single curved shape, with one strongly curved diagonal and one nearly straight diagonal. The plate edges do not longer remain straight. The second deformation pattern is not preferable, mainly for esthetic reasons. Strongly curved panels give undesired reflections and due to the curved edges no clean joints are possible. With data of experiments and finite element solutions, Staaks was able to derive a formula for determining the moment of transition between the initial and folded pattern. This formula is given by:

$$w_{trans} = 16.8t \tag{1.1}$$

Staaks formula states that the amount of torsion - expressed in the vertical displacement of one corner node - at the moment of transition is 16.8 times the thickness of the plate. This formula is valid for square panels only. Applying this formula to rectangular panels leads to deviations up to 30% of the numerical solution.

Van Laar continued this research and concluded in his thesis that the deformation path is governed by a nonlinear differential equation [13]. With the aid of the finite difference method he was able to solve this equation for square panels. Although his results were consistent with results obtained by a commercial finite element program, he was not able to derive design rules for other non-rectangular panels. Therefore, engineers still are not able to predict the deformation path of an arbitrarily shaped panel without the use of a comprehensive finite element program.

1.3 Objective

The objective of this thesis is to develop a small finite element program for analyzing cold bent glass panels. The program is intended to be used as a design tool thus requiring simple input and short computation time. The output of the application should be concentrated on visualizing the deformations, stresses and support reactions which arise from cold forming a flat panel to a curved shape.

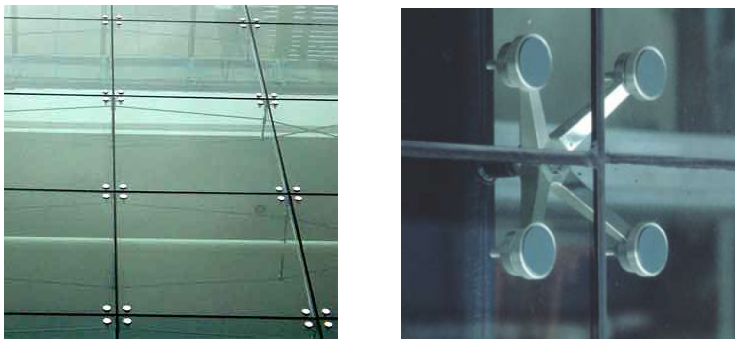
1.4 Research outline

In order to arrive at the proposed objective, some restrictions have to be introduced. In this section the three most prominent restrictions will be dealt with separately.

The first restriction concerns the geometry. From an architectural point of view, a quadrilateral is the most interesting family of shapes to use. Examples are a square shape, rectangular shape, parallelogram shape and a trapezoid shape. This restriction brings forth that the geometry of a panel is defined by the location of four corner nodes and a plate thickness. Other shapes than quadrilateral shapes (e.g. triangles, hexagons), reach beyond the scope of this thesis.

The second restriction concerns the material model. Glass exhibits almost perfect elastic behavior up to a point where it fails abruptly. This is mainly caused by the non-crystalline composition of Si-O bonds and Na-O bonds [4]. Further, the fabrication process by floating molten glass on a tin-bath assures the material to be isotropic. These properties would justify the use of a physically linear relation. Therefore, the type of analysis can be restricted to a pure geometrical nonlinear analysis. At this point no assumptions are made with respect to the exact strength and stiffness values. Primarily because different types of glass (e.g. annealed glass, heat strengthened glass, toughened glass) are available which all have different strength and stiffness properties [1]. A user of the program should be able to specify the specific material properties.

The third restriction concerns the type of support. In practice two types of supports can be used for cold forming a flat glass panel to a curved shape. The first type of support is a point support which can be executed as a glued spider fitting (Figure 1.3b) or a clamp fitting through the joints (Figure 1.4b). To approximate a certain curvature, the spiders or clamps can be adjusted in the direction perpendicular to the plate. A bolted spider fitting is not advised for cold bending since this type of spider requires a hole in the glass panel. Drilling of the hole causes small cracks which severely reduces the strength of the panel.
The second type of support is a frame-support (Figure 1.5b). This type of support only allows pure torsion; three out of four corner nodes are always in one plane, the effective displacement of the fourth corner node determines the amount of torsion. The frame support can further be subdivided in a frame supporting all four edges and a frame supporting two opposite edges, leaving the two other edges free. Only straight frame supports will be regarded. Curved frame supports reach beyond the scope of this thesis.
With both the point and frame support it is possible to impose rotations. However, in case of cold forming flat panels, imposed rotations would presumably lead to higher stresses, especially at the supports. Therefore, the supports will be used as translational restrictions only.

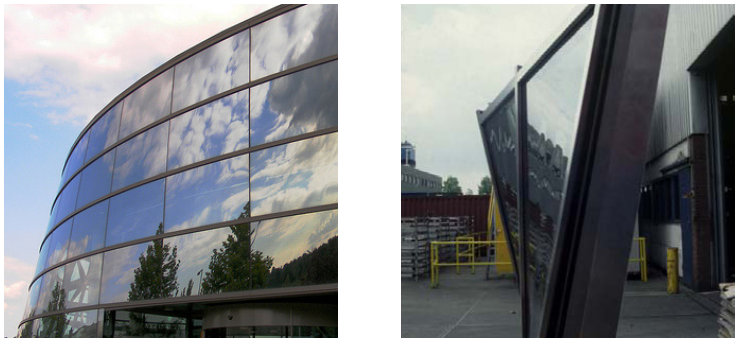


(a) (b)

Figure 1.3: (a) Glass facade panels supported by spider fittings (b) Glued spider fitting (Octatube)

(a) (b)

Figure 1.4: (a) Glass canopy supported by clamps, Zuidpoort Delft (Octatube) (b) Clamp fitting (Octatube)



(a) (b)

Figure 1.5: (a) Glass facade supported by frames, Ford Research Center (Carbonell Figueras) (b) Frame support (Reynolds)

1.5 Organization

In Chapter 2 a nonlinear finite element analysis is performed on several quadrilateral shaped panels using a commercial finite element program. The objective is to clarify the need for a geometrical nonlinear analysis when plates subjected to large displacements are to be analyzed. Moreover, several benchmarks are generated which can be used later on for comparison with the to develop finite element program. In Chapter 3 a first step in the development of the application is taken by presenting the mathematical model which is used to describe the problem. This mathematical model is translated into a finite element formulation in Chapter 4. Chapter 5 deals with the implementation of the finite element formulation into a programming language

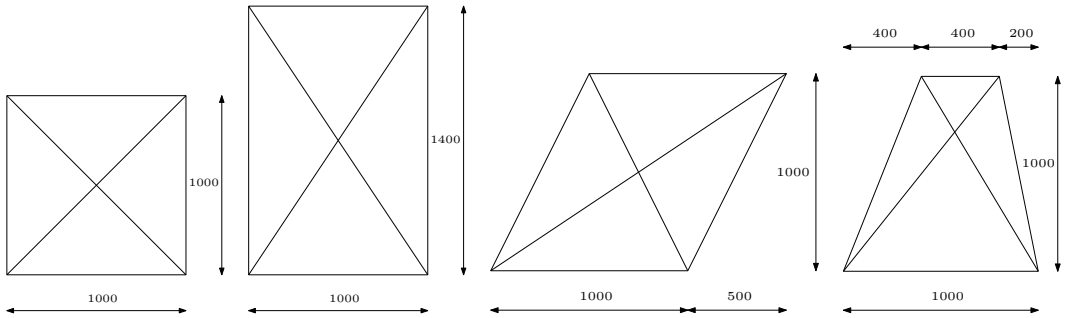
and presents the graphical user interface. The validation of the application is elaborated in Chapter 6. Finally, Chapter 7 presents conclusions and recommendations.

2. Nonlinear finite element benchmarks

The objective of this chapter is to generate several benchmarks which can be used for comparison with the developed finite element program. To this end, nonlinear finite element analyses are performed on several characteristic panel shapes which can be deduced from a quadrilateral. Diana version 9.3 is used to perform the analyses [19].

2.1 Geometrical properties

From the quadrilaterals family four shapes are selected which are shown in Figure 2.1. These four shapes capture the most prominent shapes to be used for practical application. All panels are given a constant thickness of 5 [mm].



(a) Square **(b)** Rectangle **(c)** Parallelogram **(d)** Trapezoid

Figure 2.1: Benchmark panels

2.2 Material properties

For all benchmarks a linear material model is used. The specific property values for the elasticity modulus and Poisson's ratio are set to: $E = 72000$ [N/mm^2] and $\nu = 0.0$ [-].

2.3 Modeling aspects

2.3.1 Modeling environment

All benchmarks are modeled with 2D curved shell elements. These elements assume the stress component perpendicular to the plate is zero ($\sigma_{zz} = 0$). For modeling a flat surface with a relatively thin thickness, this is a valid assumption.
In contrast to the 2D flat shell element - another 2D shell element provided by Diana - the curved shell element incorporates membrane-bending coupling behavior. The flat shell element is essentially a combination of a bending element and a membrane element but without the

membrane-bending coupling behavior and therefore not suitable for a geometrically nonlinear analysis.

2.3.2 Element selection

A four node quadrilateral curved shell element with 2×2 Gauss integration is selected. This integration scheme is the only possible integration scheme for this particular 2D curved shell element. A preliminary analysis proved that higher order curved shell elements do not yield any significant differences compared to linear curved shell elements.

2.3.3 Mesh parameters

The mesh is set to dimensions of $b \times h = 50 \times 50$ [mm]. These dimensions assure a sufficiently fine mesh for all panels.

2.3.4 Iteration procedure and norm tolerance

The iterative procedure is set to the Regular Newton Raphson method. This method assures a fully converged solution*. For the convergence criterium the displacement norm is chosen with a (default) convergence tolerance of $\epsilon = 0.01$ and the maximum number of iterations is set to 10. The loading is imposed in a number of steps. In order to obtain a smooth displacement diagram the number of steps is set to 100.

2.4 Boundary conditions

The most elementary method of cold forming a quadrilateral flat panel to a curved shape is by supporting three corner nodes and giving the fourth corner node a vertical displacement. Panels supported by a frame and panels supported by more than four points are not regarded in this chapter.
All panels are supported as illustrated for the square panel in Figure 2.2. The bottom left corner node is given a vertical displacement equal to 100 [mm] for the square, rectangle and parallelogram and 150 [mm] for the trapezoid.

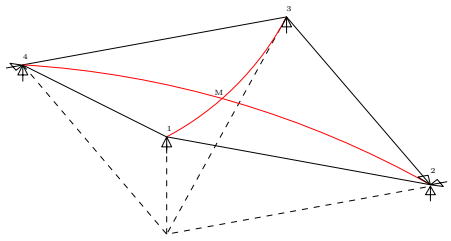


Figure 2.2: Boundary conditions

*The Newton Secant methods also performed well and appeared to yield the exact same solution as was obtained with the Regular Newton Raphson method. Other iterative methods, the constant stiffness method and the Modified Newton Raphson method, did not result in a fully converged solution and were not further examined.

2.5 Evaluation of results

For each plate the displacement path is represented by plotting the prescribed displacement against the observed displacement of the middle node, which is defined by the intersection of the diagonal lines*. In addition, the membrane stresses and the deformed shape is plotted.

2.5.1 Benchmark 1: square

Figure 2.3 shows the load-displacement diagram of the square panel. As van Laar [13] and Staaks [15] concluded, the deformation path follows the linear branch fairly well up to a point where it changes abruptly. Figure 2.4 shows the deformations of the panel.

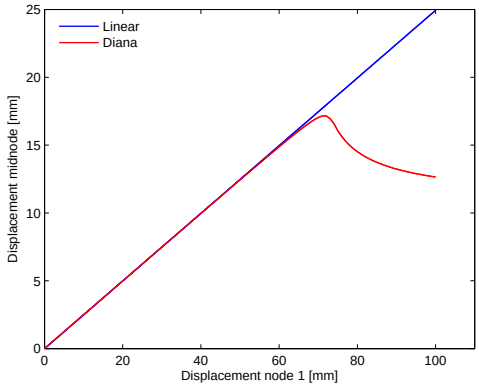
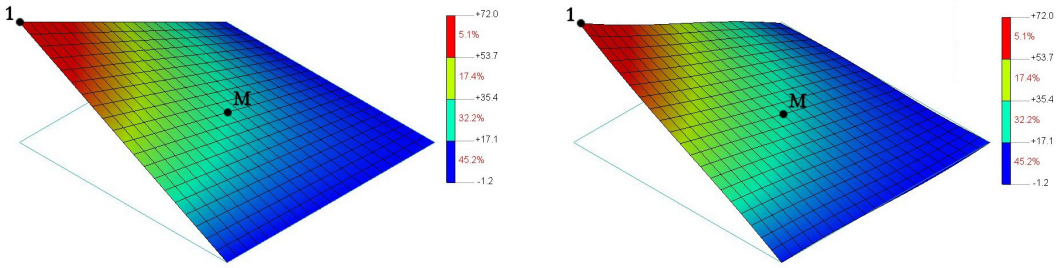


Figure 2.3: Load-displacement diagram square panel

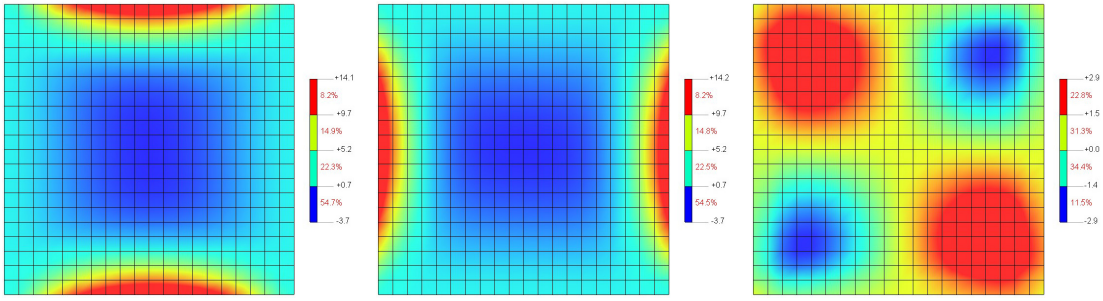


(a) **(b)**

Figure 2.4: (a) Deformation linear analysis (b) Deformation nonlinear analysis at w_{trans} [mm]

From Figure 2.4b can be seen that the plate shows slightly S-curved edges. This is mainly caused by the in z direction projected components of the shear stresses.

*An alternative way of mapping out the nonlinear behavior of plates subjected to large displacements is by plotting the prescribed displacements against the internal forces. As will be clarified in Chapter 6 this appears less illustrative for the behavior of the plate



(a) σ_{xx} **(b)** σ_{yy} **(c)** σ_{xy}

Figure 2.5: Membrane stresses [N/mm^2]

These components are a function of the torsion ($2\kappa_{xy}$) and the shear stresses (Figure 2.5c). The other stresses do not influence the displacements that significantly because their projection perpendicular to the plate is negligible due to the virtually zero curvatures κ_{xx} and κ_{yy}.

2.5.2 Benchmark 2: rectangle

Figure 2.6 shows the load-displacement diagram of a rectangular panel. Also for rectangular panels a rather abrupt moment of transition is observed.

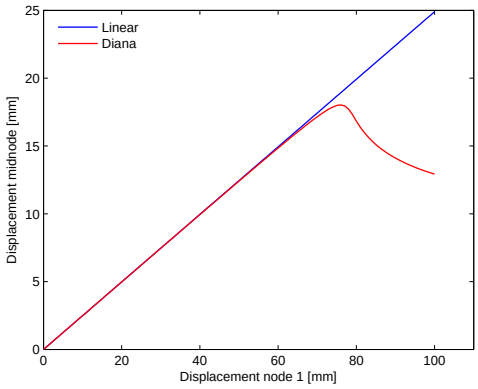
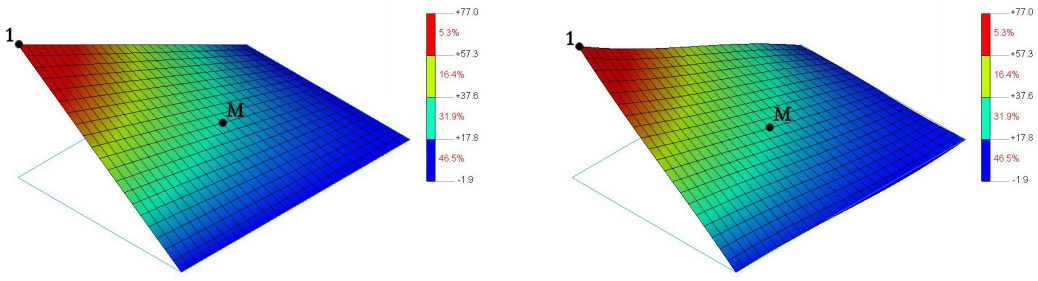


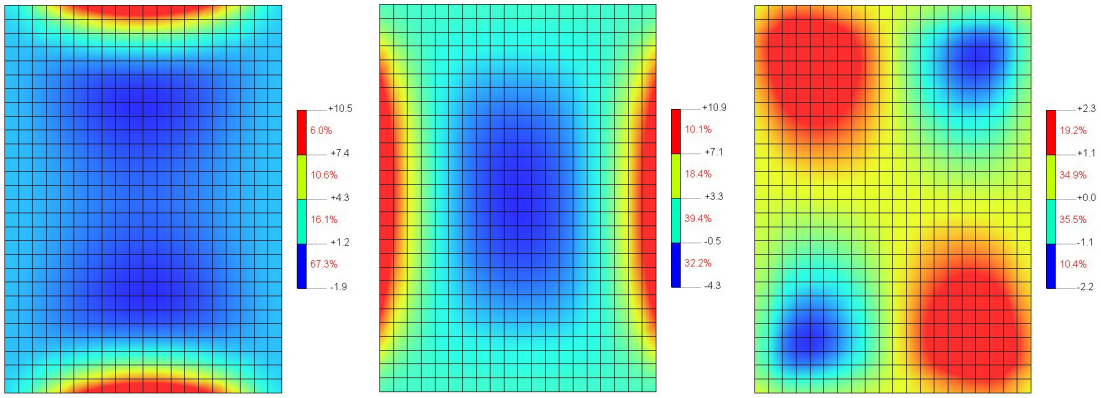
Figure 2.6: Load-displacement diagram rectangular panel



(a) **(b)**

Figure 2.7: (a) Deformation linear analysis (b) Deformation nonlinear analysis at w_{trans} [mm]

The components in z direction of the shear stresses are again causing the S-curved edges (Figure 2.7b).



(a) σ_{xx} **(b)** σ_{yy} **(c)** σ_{xy}

Figure 2.8: Membrane stresses [N/mm^2]

The normal stresses σ_{xx} and σ_{yy} do not influence the behavior of the plate because also for rectangular panels the curvatures κ_{xx} and κ_{yy} can be neglected.

2.5.3 Benchmark 3: parallelogram

Figure 2.9 shows the displacement path of a parallelogram. Right from the start the nonlinear analysis deviates from the linear analysis. Also, there is no sudden change in the deformation path observed.

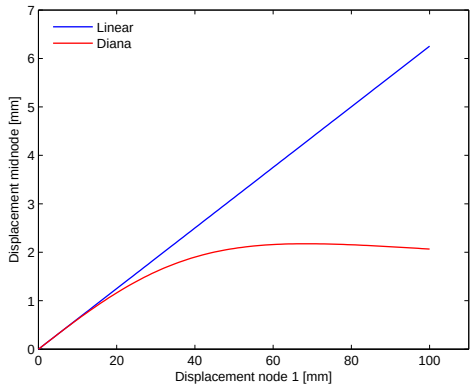
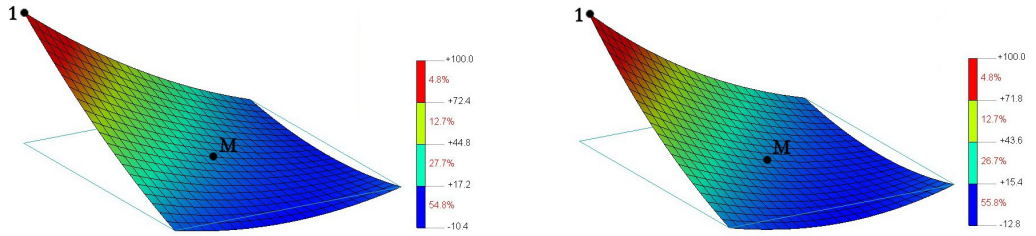


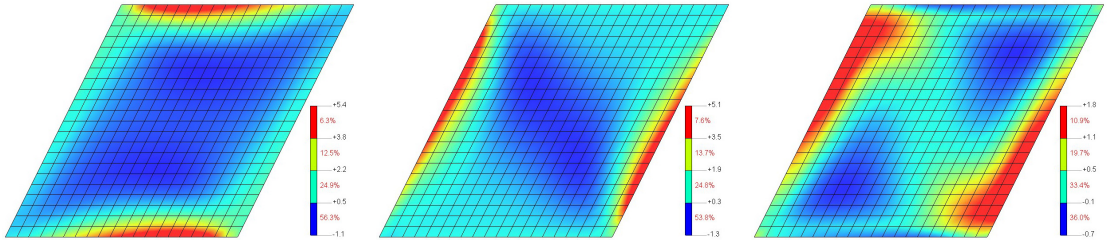
Figure 2.9: Load-displacement diagram rectangular panel



(a) **(b)**

Figure 2.10: (a) Deformation linear analysis (b) Deformation nonlinear analysis at w_{trans} [mm]

Unlike with square and rectangular panels, displacing one corner node of a parallelogram does not merely result in torsion ($2\kappa_{xy}$). The plate is also 'bent' resulting in the curvatures κ_{xx} and κ_{yy}. Therefore the membrane stresses σ_{xx} and σ_{yy} have components in z direction which influence the displacement field.



(a) σ_{xx} **(b)** σ_{yy} **(c)** σ_{xy}

Figure 2.11: Membrane stresses [N/mm^2]

The membrane stresses (Figure 2.11) show some resemblance with the stresses of the square and rectangular panel but their influence on the displacement field is rather significant and difficult to predict.

2.5.4 Benchmark 4: trapezoid

As with the parallelogram, the trapezoid shows a deformation path which deviates considerably from the linear branch.

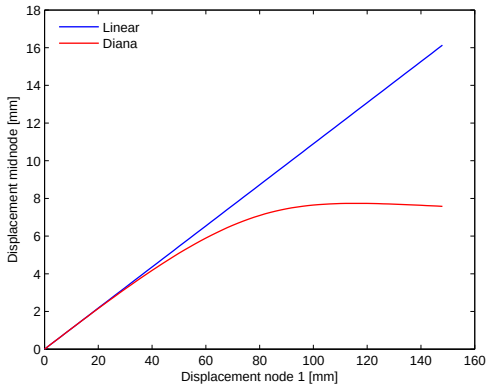
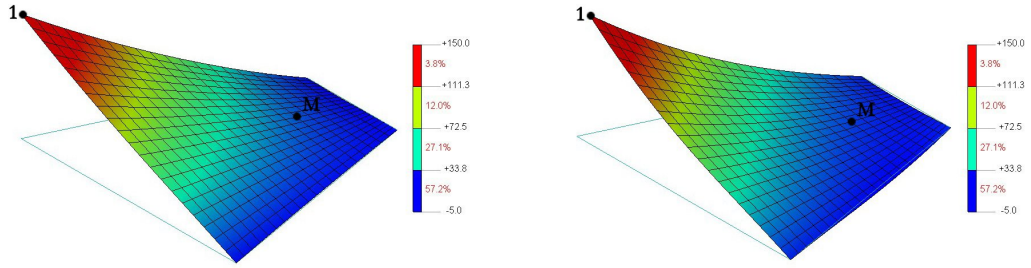


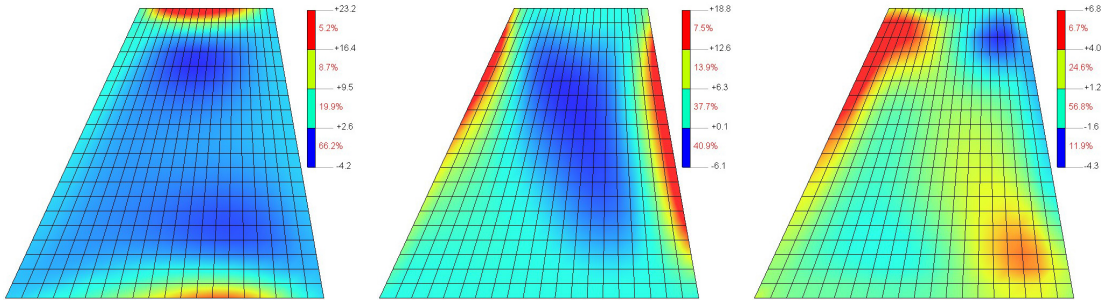
Figure 2.12: Load-displacement diagram trapezoid panel



(a) **(b)**

Figure 2.13: (a) Deformation linear analysis (b) Deformation nonlinear analysis at w_{trans} [mm]

The curvatures and membrane stresses (Figure 2.14) are difficult to predict and therefore a nonlinear finite analysis is needed to achieve a reliable solution.



(a) σ_{xx} **(b)** σ_{yy} **(c)** σ_{xy}

Figure 2.14: Membrane stresses [N/mm^2]

2.6 Conclusions

Conclusions regarding square and rectangular shaped plates:

- The deformed shape can be approximated with a linear analysis. However at a certain imposed displacement the deformation path changes abruptly. For square panels, this moment can be approximated with empirical formulas like Staaks formula: $w_{trans} = 16.8t$.
- The edges of the panels show S-curved edges due to the z components of the shear membrane stresses.
- The curvatures κ_{xx} and κ_{yy} can be neglected. The torsion $2\kappa_{xy}$ is more or less constant over the plate.
- In order to quantify the membrane stresses a nonlinear finite element analysis is needed. There are no empirical formulas available to approximate the nonlinear membrane stresses.

Conclusions regarding parallelogram and trapezoid shaped plates:

- The load-displacement diagram immediately starts to deviate from the linear branch. No abrupt changes in the diagram are observed.
- In addition to torsion the curvatures κ_{xx} and κ_{yy} are present.
- Quantifying both deformations and membrane stresses requires a nonlinear finite element analysis. There are no empirical formulas available to approximate the nonlinear deformations and membrane stresses.

From the results presented in this chapter it can be concluded that quantifying the deformations and stresses for arbitrarily shaped plates subjected to large displacements requires a nonlinear analysis.

3. Large deformation mechanics

In this chapter a first step in the development of the design tool is taken by presenting a mathematical description for plates subjected to large displacements. The mathematical description is obtained by combining a bending theory with a membrane theory.

3.1 Reissner-Mindlin bending theory

The Reissner-Mindlin theory is a plate bending theory suitable for describing both thick and thin plates, as it takes into account shear deformations [2]. Although we are mostly interested in analyzing thin plates, this theory has some advantages over other theories like the classic Kirchhoff theory. The Kirchhoff theory is applicable for thin plates only, because shear deformations are assumed to be negligible. A problem that is inherent to this theory is that the governing equation for equilibrium exhibit derivatives of order two. When casting the bending problem into a finite element model, this feature demands for $C1$ interpolations, meaning that the second derivative of the interpolating functions defined on the finite elements need to exist [18]. Higher order interpolating functions could be used to satisfy $C1$ continuity, however these functions cannot be used for isoparametric mappings. In addition, applying numerical integration over the elements becomes very cumbersome. The Reissner-Mindlin theory does allow for $C0$ continuity and is therefore much easier to cast into a finite element model. In chapter 4 the prior notice will be explained in more detail. For now the kinematic equations, constitutive model and equilibrium equations will be presented for the Reissner-Mindlin theory for plates subjected to large displacements.

3.1.1 Kinematics

The kinematic equations for the Reissner-Mindlin theory are divided in a bending part and a shear part. The curvatures are related to the rotations through:

$$\kappa_{xx} = \frac{\partial \theta_x}{\partial x} \tag{3.1}$$

$$\kappa_{yy} = \frac{\partial \theta_y}{\partial y} \tag{3.2}$$

$$\rho_{xy} = \frac{\partial \theta_x}{\partial y} + \frac{\partial \theta_y}{\partial x} \tag{3.3}$$

The shear strains are given by:

$$\gamma_x = \theta_x + \frac{\partial w}{\partial x} \tag{3.4}$$

$$\gamma_y = \theta_y + \frac{\partial w}{\partial y} \tag{3.5}$$

3.1.2 Constitutive relation

The constitutive model is also divided in a bending part and a shear part. The equations relating the curvatures and moments are:

$$m_{xx} = D_b(\kappa_{xx} + \nu\kappa_{yy}) \tag{3.6}$$

$$m_{yy} = D_b(\kappa_{yy} + \nu\kappa_{xx}) \tag{3.7}$$

$$m_{xy} = D_b\frac{1-\nu}{2}\rho_{xy} \tag{3.8}$$

where D_b is the plate bending stiffness given by:

$$D_b = \frac{Et^3}{12(1-\nu^2)} \tag{3.9}$$

The equations relating the shear strains and shear forces are:

$$v_x = C\gamma_x \tag{3.10}$$

$$v_y = C\gamma_y \tag{3.11}$$

where C is the plate shear stiffness given by:

$$\boldsymbol{C} = \frac{Et}{2(1+\nu)} \tag{3.12}$$

3.1.3 Equilibrium equations

The third set of equations needed to describe the Reissner-Mindlin theory are the equilibrium equations. Consider a small plate element as given in Figure 3.1a.

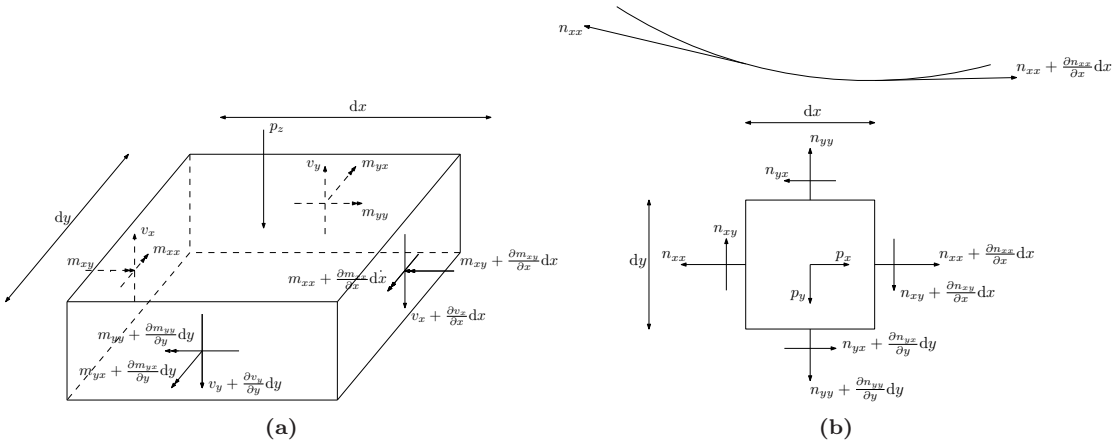


Figure 3.1: (a) Equilibrium of a bending element (b) Equilibrium of a membrane element

Equilibrium of moments gives *

$$\frac{\partial m_{xx}}{\partial x} + \frac{\partial m_{yx}}{\partial y} = v_x \tag{3.13}$$

$$\frac{\partial m_{yy}}{\partial y} + \frac{\partial m_{xy}}{\partial x} = v_y \tag{3.14}$$

$$-\frac{\partial v_x}{\partial x} - \frac{\partial v_y}{\partial y} = p_z \tag{3.15}$$

In addition to the load p_z, the lateral forces p_x and p_y are assumed to be present in the middle surface of the plate (Figure 3.1b). For small displacements these forces can be neglected. For large displacements the components perpendicular to the plate influence the equilibrium significantly [7].
Equilibrium of the membrane forces gives:

$$\frac{\partial n_{xx}}{\partial x} + \frac{\partial n_{xy}}{\partial y} + p_x = 0 \tag{3.16}$$

$$\frac{\partial n_{yy}}{\partial y} + \frac{\partial n_{xy}}{\partial x} + p_y = 0 \tag{3.17}$$

The components of the membrane forces in z direction are formulated as:

$$n_{xx}\frac{\partial \theta_x}{\partial x} \tag{3.18}$$

$$n_{yy}\frac{\partial \theta_y}{\partial y} \tag{3.19}$$

$$2n_{xy}(\frac{\partial \theta_x}{\partial y} + \frac{\partial \theta_y}{\partial x}) \tag{3.20}$$

Adding these components to the right hand side of (3.15) gives the equilibrium equations of the Reissner-Mindlin theory for plates subjected to large displacements.

$$\frac{\partial m_{xx}}{\partial x} + \frac{\partial m_{yx}}{\partial y} = v_x \tag{3.21}$$

$$\frac{\partial m_{yy}}{\partial y} + \frac{\partial m_{xy}}{\partial x} = v_y \tag{3.22}$$

$$-\frac{\partial v_x}{\partial x} - \frac{\partial v_y}{\partial y} = p_z + n_{xx}\frac{\partial \theta_x}{\partial x} + n_{yy}\frac{\partial \theta_y}{\partial y} + 2n_{xy}(\frac{\partial \theta_x}{\partial y} + \frac{\partial \theta_y}{\partial x}) \tag{3.23}$$

3.2 Membrane theory

In the previous section the assumption is made that the membrane forces are applied externally. In this section it will be elaborated how membrane forces arise from large displacements [16].

*As can be seen from (3.13) to (3.15) only derivatives of order 1 are present, thus the Reissner-Mindlin theory allows for $C0$ continuity. This way of formulating the equilibrium equations demands from the kinematic equations and the constitutive model to take into account shear strains and shear forces.

3.2.1 Kinematics

The kinematic equations of a plate subjected to large displacements can be formulated as:

$$\epsilon_{xx} = \frac{\partial u_x}{\partial x} + \frac{1}{2}(\frac{\partial w}{\partial x})^2 \tag{3.24}$$

$$\epsilon_{yy} = \frac{\partial u_y}{\partial y} + \frac{1}{2}(\frac{\partial w}{\partial y})^2 \tag{3.25}$$

$$\gamma_{xy} = \frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} + \frac{\partial w}{\partial x}\frac{\partial w}{\partial y} \tag{3.26}$$

For ϵ_{xx} this is illustrated in Figure 3.2a and for γ_{xy} in Figure 3.2b.

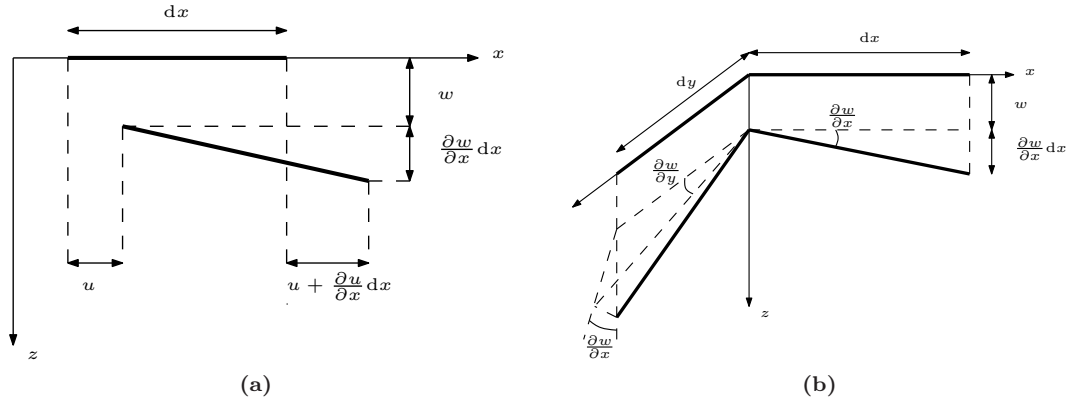


Figure 3.2: (a) Large displacements x direction (b) Large displacements xy direction

Assume that relatively thin plates are to be analyzed:

$$\gamma_x = 0 \rightarrow \theta_x = -\frac{\partial w}{\partial x} \tag{3.27}$$

$$\gamma_y = 0 \rightarrow \theta_y = -\frac{\partial w}{\partial y} \tag{3.28}$$

The kinematic relation becomes:

$$\epsilon_{xx} = \frac{\partial u_x}{\partial x} + \frac{1}{2}\theta_x^2 \tag{3.29}$$

$$\epsilon_{yy} = \frac{\partial u_y}{\partial y} + \frac{1}{2}\theta_y^2 \tag{3.30}$$

$$\gamma_{xy} = \frac{\partial u_x}{\partial y} + \frac{\partial u_y}{\partial x} + \theta_x\theta_y \tag{3.31}$$

3.2.2 Constitutive relation

The constitutive equations relating the membrane strains to the membrane forces are formulated as:

$$n_{xx} = D_m(\epsilon_{xx} + \nu\epsilon_{yy}) \tag{3.32}$$

$$n_{yy} = D_m(\epsilon_{yy} + \nu\epsilon_{xx}) \tag{3.33}$$

$$n_{xy} = D_m\frac{1-\nu}{2}\gamma_{xy} \tag{3.34}$$

where D_m is the membrane stiffness given by:

$$D_m = \frac{Et}{(1-\nu^2)} \tag{3.35}$$

3.2.3 Equilibrium equations

Setting the lateral loads p_x and p_y to zero, equilibrium of a small membrane element (Figure 3.1b) gives:

$$\frac{\partial n_{xx}}{\partial x} + \frac{\partial n_{yx}}{\partial y} = 0 \tag{3.36}$$

$$\frac{\partial n_{yy}}{\partial y} + \frac{\partial n_{xy}}{\partial x} = 0 \tag{3.37}$$

Although the lateral loads p_x and p_y are zero, the membrane forces can be non-zero due to the extra terms appearing in the kinematic equations.

3.3 Summary

A mathematical description for a plate subjected to large displacements can be formulated using the Reissner-Mindlin theory and the classic membrane theory. To this end two modifications are needed:

- The Reissner-Mindlin theory is modified by adding the components in z direction of the membrane stresses to the equilibrium equations
- The membrane theory is modified by adding an extra contribution due to large rotations to the kinematic equations

It is emphasized that for the Reissner-Mindlin theory the kinematic and constitutive equations remain unchanged and for the membrane theory the constitutive and equilibrium equations remain unchanged.

4. Finite element formulation

In this chapter the mathematical description from the previous chapter is cast into a finite element formulation. First a short overview of the finite element method is given. Further the development of an element with combined membrane-bending behavior is presented. Finally, the solution algorithm needed to solve the nonlinear finite element formulation is given.

4.1 Overview

The finite element method is a numerical procedure for obtaining approximate solutions to the governing equations that describe the response of a physical system [12]. In other words, a physical problem, cast in a mathematical formulation, can be solved using the finite element method. A key component of the finite element method is the discretization of the problem into finite elements. For each element the governing equations are transformed into algebraic element equations which are an approximation of the governing equations. These element equations are assembled into the system equations that characterize the response of the entire system. Thus, application of the finite element method leads to a (often large) system of equations.

4.2 Element development

First a linear Reissner-Mindlin bending element and a linear membrane element is developed. Combination of both elements results in the desired formulation for plates subjected to large displacements.

4.2.1 Reissner-Mindlin bending elements

The kinematic equations for the Reissner-Mindlin theory (3.1) to (3.5) reveal that two unknown fields, the vertical displacements and the rotations, need to be determined. For an element both fields can be written in terms of shape functions and nodal values.

$$\boldsymbol{w} = \boldsymbol{N}^w \boldsymbol{a}^w \tag{4.1}$$

$$\boldsymbol{\theta} = \boldsymbol{N}^\theta \boldsymbol{a}^\theta \tag{4.2}$$

The matrix $\boldsymbol{N}$ contains the shape functions and the vectors $\boldsymbol{a}^w$ and $\boldsymbol{a}^\theta$ the nodal values of resp. displacements and rotations.
Taking the derivatives of (4.1) gives:

$$\boldsymbol{w}_{,xy} = \boldsymbol{B}^w \boldsymbol{a}^w \tag{4.3}$$

$$\boldsymbol{\kappa} = \boldsymbol{B}^\theta \boldsymbol{a}^\theta \tag{4.4}$$

For the finite element method it is essential that the governing equations for equilibrium (3.13) to (3.15) are transformed into their weak form, also known as the variational form. To this end, equation (3.13) and equation (3.14) are multiplied by a virtual rotation $\boldsymbol{\delta\theta}$ and (3.15) by a virtual displacement field $\boldsymbol{\delta w}$. Both equations are subsequently integrated over the domain A.

$$\boldsymbol{\delta\theta}^T \boldsymbol{f}^\theta = \int_A \boldsymbol{\delta\kappa}^T \boldsymbol{m} \, \mathrm{d}A - \int_A \boldsymbol{\delta\theta v} \, \mathrm{d}A \tag{4.5}$$

$$\boldsymbol{\delta w}^T \boldsymbol{f}^w = \int_A \boldsymbol{\delta w}_{,}^T \boldsymbol{v} \, \mathrm{d}A \tag{4.6}$$

Inserting the discretized fields into (4.5) yields:

$$\boldsymbol{f}^\theta = \int_A \boldsymbol{B}^{\theta^T} \boldsymbol{D}_b \boldsymbol{B}^\theta \, \mathrm{d}A \boldsymbol{a}^\theta - \int_A \boldsymbol{N}^{\theta^T} \boldsymbol{C} \boldsymbol{B}^w \, \mathrm{d}A \boldsymbol{a}^w + \int_A \boldsymbol{N}^{\theta^T} \boldsymbol{C} \boldsymbol{N}^\theta \, \mathrm{d}A \boldsymbol{a}^\theta \tag{4.7}$$

$$\boldsymbol{f}^w = \int_A \boldsymbol{B}^{w^T} \boldsymbol{C} \boldsymbol{B}^w \, \mathrm{d}A \boldsymbol{a}^w - \int_A \boldsymbol{B}^{w^T} \boldsymbol{C} \boldsymbol{N}^\theta \, \mathrm{d}A \boldsymbol{a}^\theta \tag{4.8}$$

For an element this can be expressed as:

$$\begin{bmatrix} \boldsymbol{k}^{ww} & \boldsymbol{k}^{w\theta} \\ \boldsymbol{k}^{\theta w} & \boldsymbol{k}^{\theta\theta} \end{bmatrix} \begin{Bmatrix} \boldsymbol{a}^w \\ \boldsymbol{a}^\theta \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{f}^w \\ \boldsymbol{f}^\theta \end{Bmatrix} \tag{4.9}$$

where

$$\boldsymbol{k}^{ww} = \int_A \boldsymbol{B}^{w^T} \boldsymbol{C} \boldsymbol{B}^w \, \mathrm{d}A \tag{4.10}$$

$$\boldsymbol{k}^{w\theta} = -\int_A \boldsymbol{B}^{w^T} \boldsymbol{C} \boldsymbol{N}^\theta \, \mathrm{d}A \tag{4.11}$$

$$\boldsymbol{k}^{\theta w} = -\int_A \boldsymbol{N}^{w^T} \boldsymbol{C} \boldsymbol{B}^w \, \mathrm{d}A \tag{4.12}$$

$$\boldsymbol{k}^{\theta\theta} = \int_A \boldsymbol{B}^{\theta^T} \boldsymbol{D}_b \boldsymbol{B}^\theta \, \mathrm{d}A + \int_A \boldsymbol{N}^{\theta^T} \boldsymbol{C} \boldsymbol{N}^\theta \, \mathrm{d}A \tag{4.13}$$

4.2.2 Membrane elements

The displacement field for the membrane elements can be represented by a collection of shape functions and discrete nodal values as:

$$\boldsymbol{u} = \boldsymbol{N}^u \boldsymbol{a}^u \tag{4.14}$$

Taking the derivatives gives:

$$\boldsymbol{\epsilon} = \boldsymbol{B}^u \boldsymbol{a}^u \tag{4.15}$$

To arrive at the variational form of the governing equations for equilibrium, equations (3.16) and (3.17) are multiplied by a virtual displacement $\boldsymbol{\delta u}$ and integrated over the domain A.

$$\boldsymbol{\delta u}^T \boldsymbol{f}^u = \int_A \boldsymbol{\delta\epsilon}^T \boldsymbol{\sigma} \, \mathrm{d}A \tag{4.16}$$

Inserting the discretized fields into (4.16) gives:

$$\boldsymbol{f}^u = \int_A \boldsymbol{B}^{u^T} \boldsymbol{D}_m \boldsymbol{B}^u \, \mathrm{d}A \boldsymbol{a}^u \tag{4.17}$$

For an element this can be expressed as:

$$\boldsymbol{f}^u = \boldsymbol{K}^u \boldsymbol{a}^u \tag{4.18}$$

where

$$\boldsymbol{K}^u = \int_A \boldsymbol{B}^{u^T} \boldsymbol{D}_m \boldsymbol{B}^u \, \mathrm{d}A \tag{4.19}$$

4.2.3 Coupling the membrane and bending elements

The Reissner-Mindlin bending element and the membrane element are now coupled by two operations:

- For the membrane element: Adding an extra contribution to the strains due to large displacements
- For the bending element: Adding the nodal force components in z direction of the membrane stresses

The above operations renders the problem nonlinear; the extra strain contribution is dependent of the vertical displacement field and the vertical displacement field is influenced by the membrane stresses resulting from the extra strain contribution. Due to this dependency it is not possible to simply superimpose the membrane element and the bending element to a single membrane-bending element. Both membrane element and bending element are kept separate but must share the same element configuration. Figure 4.1 shows a four node element which is used as a bending element as well as a membrane element.

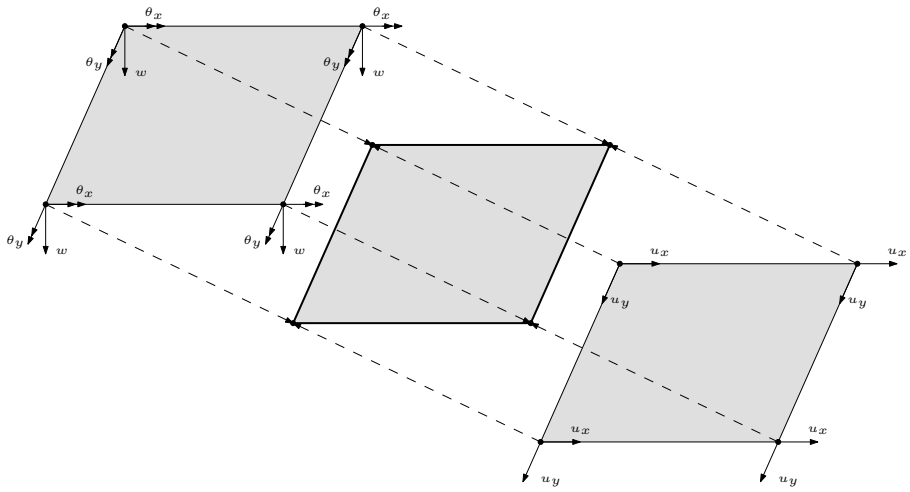


Figure 4.1: A four node element for a membrane element and a bending element

First regard the membrane element. Analogous to equations (3.29) to (3.31) the discretized strain field can be written as

$$\boldsymbol{\epsilon} = \boldsymbol{B}^u \boldsymbol{a}^u + \boldsymbol{\theta}^* \tag{4.20}$$

where

$$\boldsymbol{\theta}^* = \begin{Bmatrix} \frac{1}{2}\theta_x^2 \\ \frac{1}{2}\theta_y^2 \\ \theta_x\theta_y \end{Bmatrix} \tag{4.21}$$

The variational form of the governing equilibrium equations becomes

$$\boldsymbol{f}^u = \int_A \boldsymbol{B}^{u^T} \boldsymbol{D}_m \boldsymbol{B}^u \, \mathrm{d}A \boldsymbol{a}^u + \int_A \boldsymbol{B}^{u^T} \boldsymbol{D}_m \boldsymbol{\theta}^* \, \mathrm{d}A \tag{4.22}$$

When no external membrane forces are imposed on the membrane part, the displacement field can be calculated from

$$\boldsymbol{K}^u \boldsymbol{a}^u = \boldsymbol{f}_{int,\theta^*} \tag{4.23}$$

where

$$\boldsymbol{f}_{int,\theta^*} = -\int_A \boldsymbol{B}^{u^T} \boldsymbol{D}_m \boldsymbol{\theta}^* \, \mathrm{d}A \tag{4.24}$$

$$\boldsymbol{K}^u = \int_A \boldsymbol{B}^{u^T} \boldsymbol{D}_m \boldsymbol{B}^u \, \mathrm{d}A \tag{4.25}$$

If $\boldsymbol{f}_{int,\theta^*}$ which is a function of $\boldsymbol{\theta}^*$ is calculated, the displacement field $\boldsymbol{a}^u$ can be determined by solving the system of equations. From $\boldsymbol{a}^u$ the membrane stresses can subsequently be calculated using:

$$\boldsymbol{\sigma} = \boldsymbol{D}_m(\boldsymbol{B}^u \boldsymbol{a}^u + \boldsymbol{\theta}^*) \tag{4.26}$$

For the bending part the membrane forces in z direction must be added to the governing equations to complete the coupling behavior. Analogous to (3.23) this contribution extends (4.8) to

$$\boldsymbol{f}^w = \int_A \boldsymbol{B}^{w^T} \boldsymbol{C} \boldsymbol{B}^w \, \mathrm{d}A \boldsymbol{a}^w - \int_A \boldsymbol{B}^{w^T} \boldsymbol{C} \boldsymbol{N}^\theta \, \mathrm{d}A \boldsymbol{a}^\theta + \int_A \boldsymbol{\sigma}^* \, \mathrm{d}A \tag{4.27}$$

where

$$\boldsymbol{\sigma}^* = n_{xx}\kappa_{xx} + n_{yy}\kappa_{yy} + n_{xy}2\kappa_{xy} \tag{4.28}$$

For convenience the membrane forces in z direction are gathered in

$$\boldsymbol{f}_{int,\sigma^*} = \int_A \boldsymbol{\sigma}^* \, \mathrm{d}A \tag{4.29}$$

Equation (4.9) can be extended to:

$$\begin{bmatrix} \boldsymbol{k}^{ww} & \boldsymbol{k}^{w\theta} \\ \boldsymbol{k}^{\theta w} & \boldsymbol{k}^{\theta\theta} \end{bmatrix} \begin{Bmatrix} \boldsymbol{a}^{w} \\ \boldsymbol{a}^{\theta} \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{f}^{w} \\ \boldsymbol{f}^{\theta} \end{Bmatrix} + \begin{Bmatrix} \boldsymbol{f}_{int,\sigma^*} \\ 0 \end{Bmatrix} \tag{4.30}$$

or

$$\boldsymbol{K}^{w\theta}\boldsymbol{a}^{w\theta} = \boldsymbol{f}^{w\theta} + \boldsymbol{f}_{int,\sigma^*} \tag{4.31}$$

In total two systems of equations are formulated which describe the behavior of an element subjected to large displacements.

$$\boldsymbol{K}^{u}\boldsymbol{a}^{u} = \boldsymbol{f}_{int,\theta^*} \tag{4.32}$$

$$\boldsymbol{K}^{w\theta}\boldsymbol{a}^{w\theta} = \boldsymbol{f}^{w\theta} + \boldsymbol{f}_{int,\sigma^*} \tag{4.33}$$

In appendix A a four node element and a nine node element are developed from the matrix notation presented in this section.

4.3 Solution techniques

4.3.1 Incremental formulation

Due to the mutual dependency of (4.32) and (4.33) no direct solution method is possible. This can be overcome by casting (4.33) in its incremental form:

$$\boldsymbol{K}^{w\theta}\Delta\boldsymbol{a}^{w\theta} = \boldsymbol{f}_{ext}^{t+\Delta t} - \boldsymbol{f}_{int}^{t} \tag{4.34}$$

The internal force vector $\boldsymbol{f}_{int}$ at time t is represented by

$$\boldsymbol{f}_{int}^{t} = \boldsymbol{K}^{w\theta}\boldsymbol{a}^{w\theta^t} - \boldsymbol{f}_{int,\sigma^*}^{t} \tag{4.35}$$

The internal force vector at time t is generally not in equilibrium with the external loads, thus the solution of (4.34) is likely to diverge from the true equilibrium path, especially when large loading steps are employed. This is illustrated in Figure 4.2.

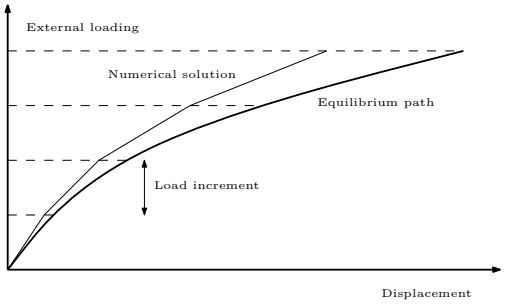


Figure 4.2: Incremental solution procedure

4.3.2 Incremental-iterative formulation

By adding equilibrium iterations within each loading step the drifting away from the equilibrium path can be prevented. There are several iteration schemes available to obtain equilibrium: (1) the Newton Raphson methods, (2) the Secant Newton methods and (3) the linear stiffness method. The most effective schemes are the Newton methods. These methods require some form of updating of the stiffness matrix, either within a loading step or for the next iteration. The linear stiffness method, or constant stiffness method, computes a stiffness matrix at the beginning of the first load step and uses it throughout the following increments (Figure 4.3).

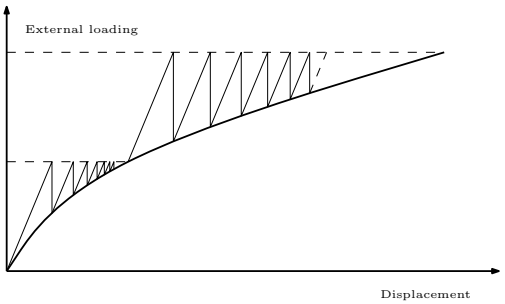


Figure 4.3: Incremental iterative linear stiffness method

This is very robust in terms of convergence but may require many iterations to obtain a reasonable accuracy. However since it is assumed that the stiffness matrix varies rather slow and the difficult operations needed to construct a new tangential stiffness matrix can be omitted, the linear stiffness method is the most suitable method to implement.

4.3.3 Iterative formulation

A merely iterative version of the linear stiffness method is expected to perform much faster than the incremental-iterative version. This would eliminate the increments, meaning that the load is applied in one single step. Chapter 6 will present proof of this assumption. Casting the finite element formulation (4.32) and (4.33) into the iterative linear stiffness method yields the following solution scheme:

1. Compute constant stiffness matrix for the bending part $\boldsymbol{K}^{w\theta}$ and the membrane part $\boldsymbol{K}^{u}$
2. Assemble external force vector for the bending part $\boldsymbol{f}_{ext}$
3. Solve the linear system $\mathbf{d}\boldsymbol{a}^{w\theta}_{j+1} = \boldsymbol{K}^{w\theta-1}(\boldsymbol{f}_{ext} - \boldsymbol{f}_{int,j})$
4. Add the correction to the displacement vector $\boldsymbol{a}^{w\theta}_{j+1} = \boldsymbol{a}^{w\theta}_{j} + \mathbf{d}\boldsymbol{a}^{w\theta}_{j+1}$
5. Determine $\boldsymbol{\theta}^{*}_{j+1}$ in order to compute $\boldsymbol{f}_{int,\theta^{*},j+1}$
6. Solve the linear system $\boldsymbol{a}^{u}_{j+1} = \boldsymbol{K}^{u-1}\boldsymbol{f}_{int,\theta^{*},j+1}$
7. Compute the membrane stresses $\boldsymbol{\sigma}_{j+1} = \boldsymbol{D}_{m}(\boldsymbol{B}^{u}\boldsymbol{a}^{u}_{j+1} + \boldsymbol{\theta}^{*}_{j+1})$
8. Compute the curvature $\boldsymbol{\kappa}_{j+1} = \boldsymbol{B}^{\theta}\boldsymbol{a}^{w\theta}_{j+1}$

9. Use the nodal membrane stresses and nodal curvatures to compute $\boldsymbol{f}_{int,\sigma^*,j+1}$
10. Compute the internal force vector $\boldsymbol{f}_{int,j+1} = \boldsymbol{K}\boldsymbol{a}_{j+1}^{w\theta} - \boldsymbol{f}_{int,\sigma^*,j+1}$
11. Check convergence. If the solution has converged stop, otherwise go to (3)

A graphical representation of the above iterative process is given in Figure (4.4)

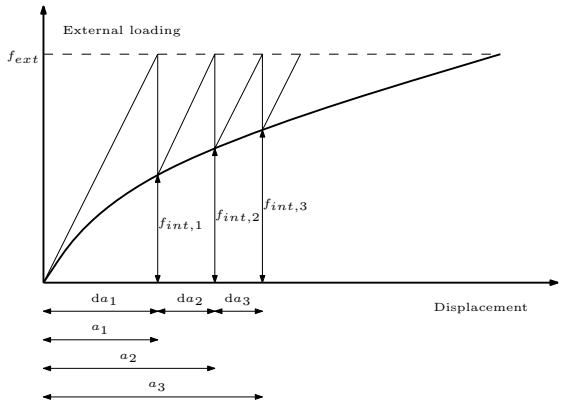


Figure 4.4: Iterative linear stiffness method

4.4 Displacement control

Up till now, no notion has been made about the way the external loading should be imposed. Basically there exist two methods; load control and displacement control. In load control the external forces are applied in a number of steps, or - in case of a merely iterative scheme - in one step. In displacement control on the other hand, a prescribed displacement is applied which causes stresses within the plate, which in turn results in nodal forces. Since we are interested in what happens when a plate is subjected to a certain prescribed displacement rather than when it is subjected to a certain force, displacement control is the method to be used. The next chapter will explain in more detail how displacement control is implemented.

4.5 Convergence criteria

In order to determine convergence of the iterative procedure, a convergence criterion is needed [3]. This means that the error of a certain quantity, e.g. a force or displacement, is to be bound by a prescribed tolerance. If the error does not become smaller than this prescribed tolerance the iterative procedure is said not to have converged. Several criteria exist to detect convergence; force norm, displacement norm, energy norm. Since the displacements are monitored during each iteration, the displacement norm is the most convenient criteria to implement. This criteria is defined as:

$$\frac{\sqrt{\mathbf{d}\boldsymbol{a}_{j+1}^T \mathbf{d}\boldsymbol{a}_{j+1}}}{\sqrt{\mathbf{d}\boldsymbol{a}_1^T \mathbf{d}\boldsymbol{a}_1}} \leq \epsilon \tag{4.36}$$

The value of the convergence tolerance ϵ determines in a great deal accuracy of the solution and the computation time. This value will be determined in Chapter 6.

4.6 Isoparametric mapping

In practice, isoparametric elements are used [18]. These are elements which are defined on a natural coordinate system (ξ, η) and are typically of unit size. This allows for using only one polynomial function for evaluating the shape functions of a specific element, irrespective of its exact size or shape. Moreover, applying numerical integration over a natural coordinate system becomes much easier. An element in the natural domain is related to the real element in the Cartesian system via an isoparametric map, which is built from the element shape functions (Figure 4.5).

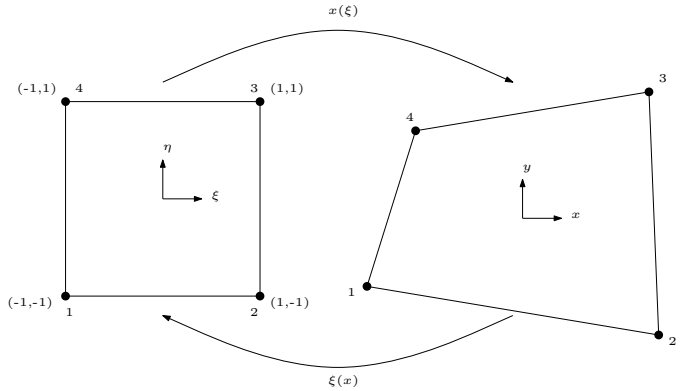


Figure 4.5: Isoparametric mapping of a four node element

For example, the stiffness matrix of an isoparametric membrane element can be written as

$$\int_A \boldsymbol{B}^{u^T} \boldsymbol{D}_m \boldsymbol{B}^u \, \mathrm{d}A = \int_{-1}^{1} \int_{-1}^{1} \boldsymbol{B}^{u^T} \boldsymbol{D}_m \boldsymbol{B}^u j \, \mathrm{d}\xi \mathrm{d}\eta \tag{4.37}$$

where j is the determinant of the Jacobian which relates the Cartesian system to the natural system. All shape functions appearing in the various matrices can be written in the natural coordinate system. In Appendix B the isoparametric shape functions for a four node element and a nine node element are given.

4.7 Numerical integration

After converting all elements to the natural coordinate system, Gauss integration is used to evaluate the (still analytical) integrals. As an example the stiffness matrix of an membrane element is considered to illustrated how numerical integration is applied:

$$\int_{-1}^{1} \int_{-1}^{1} \boldsymbol{B}^{u^T} \boldsymbol{D}_m \boldsymbol{B}^u j \, \mathrm{d}\xi \mathrm{d}\eta \simeq \sum_{i}^{n} \boldsymbol{B}^{u^T}(\xi_i, \eta_i) \boldsymbol{D}_m \boldsymbol{B}^u j(\xi_i, \eta_i) w_i \tag{4.38}$$

Here n is the number of integration points within an element and w_i the weight related to the specific integration point. In Appendix C the integration scheme for 2×2 Gauss integration and 3×3 Gauss integration is given.

4.8 Summary

In this chapter the mathematical description from Chapter 3 is translated into a finite element formulation. In Appendix A this formulation is used to develop a membrane and a bending element for both a four node configuration and a nine node configuration. Due to the coupling of the membrane and bending elements an iterative solution method is needed. A merely iterative constant stiffness method is expected to be the most suitable for implementation since it does not require a form of updating of the stiffness matrices. In order to determine convergence of the iterative constant stiffness method, the displacement norm is chosen as the convergence criterium.
To make the elements suitable for implementation in a computer program, isoparametric shape functions are chosen. This makes formulating the shape functions more practical and allows using Gauss integration to evaluate the integrals.
At this stage the framework of the application is established. The next chapter will deal with the implementation of this framework into a programming language. The element order, convergence tolerance ϵ and type of Gauss integration scheme will be established in Chapter 6.

5. PBA design

In the previous chapter a finite element formulation and solution algorithm for plates subjected to large displacements was presented. This chapter deals with the implementation of the finite element formulation in the MATLAB programming language [14].

5.1 Application setup

PBA is a graphical application, meaning that users do not have to know anything about the MATLAB programming language in order to use it. The Graphical User Interface (GUI) on startup is shown in Figure 5.1.

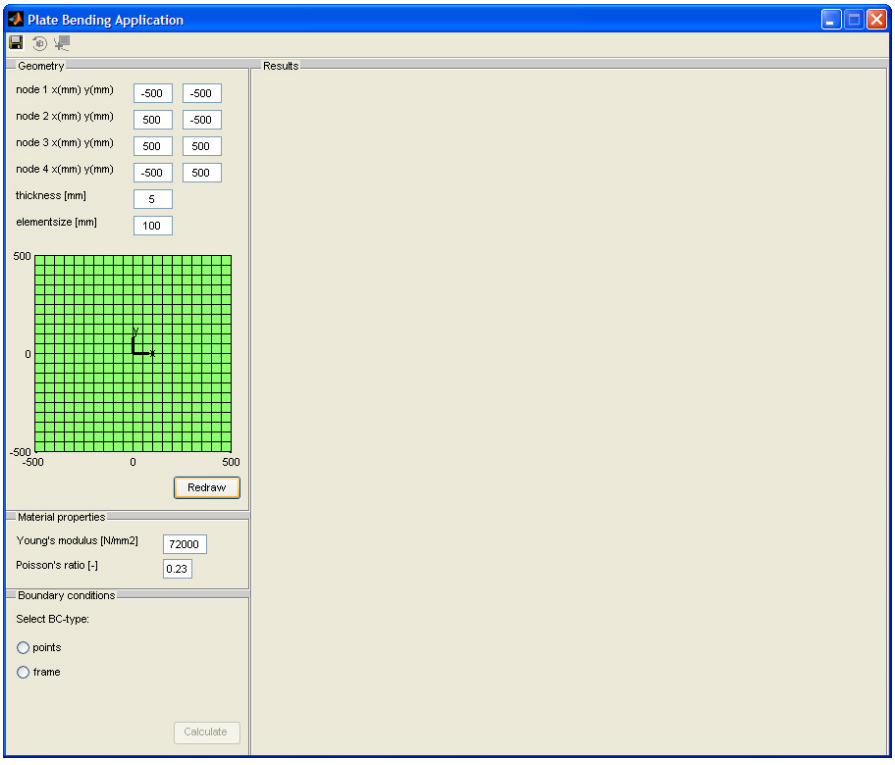


Figure 5.1: Graphical user interface on startup

The input screen is divided in three panels: the Geometry panel, the Material properties panel and the Boundary conditions panel. In the Geometry panel a user can input the geometry of the plate by defining the locations of the four corner nodes and a plate thickness (Figure 5.2). Also an approximate element size must be set in order to generate a preliminary mesh which is needed later on for defining the boundary conditions.

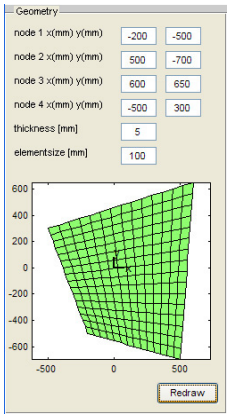


Figure 5.2: Geometry panel

By pressing the Redraw button, the user can verify graphically that the geometrical parameters are correct.
In the Material properties panel a user can specify the Young's modulus E and the Poisson's ratio ν depending on the type of glass to be used for the plate (Figure 5.3).

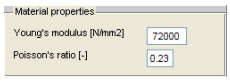
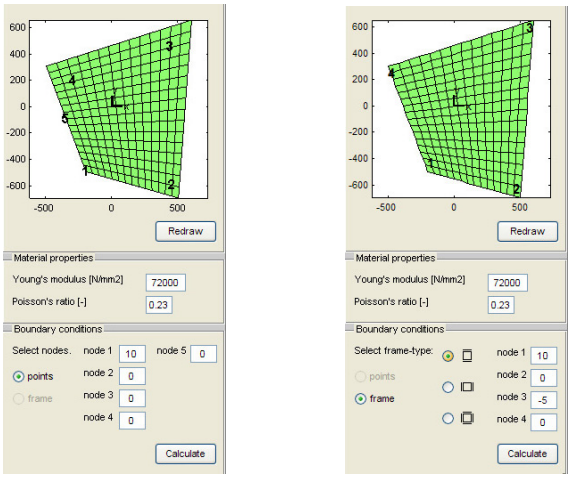


Figure 5.3: Material properties panel

A user can save the geometry and material properties by pressing the save button in the toolbar. The input fields for geometry and material properties are automatically set to the saved values when the application is restarted.
In the Boundary conditions panel a user can choose between two types of boundary conditions (Figure 5.4)



(a) Points **(b)** Frame

Figure 5.4: Boundary conditions panel

When choosing 'Points' (Figure 5.4a), a user can select nodes by clicking on the geometry plot in the Geometry panel. For every new node an input field is enabled where the displacement of the node can be prescribed.
When selecting 'Frame' (Figure 5.4b) a button group is enabled where a user can further specify the frame type. Three types are possible: (1) supported on the top and bottom edge, (2) supported on the left and right edge and (3) supported on all four edges. Selecting one of the frame types enables four input fields, each representing the vertical displacements of the four corner nodes. Clearing the boundary conditions is done by pressing the 'Redraw' button.
By pressing the Calculate button an object `InputData` containing the information of all input fields is created and - after checking its validity - is passed to the function `startFEMCode`. The function `startFEMCode` inflicts the actual finite element calculation. It is divided in three parts; the preprocessor, the kernel and the postprocessor. The source code of each part is given in the next sections. If the finite element calculation is successfully finished, the function `startFEMCode` returns the objects containing the vertical displacement field, the stresses and the reaction forces. These results can be viewed in the Results panel (Figure 5.5).

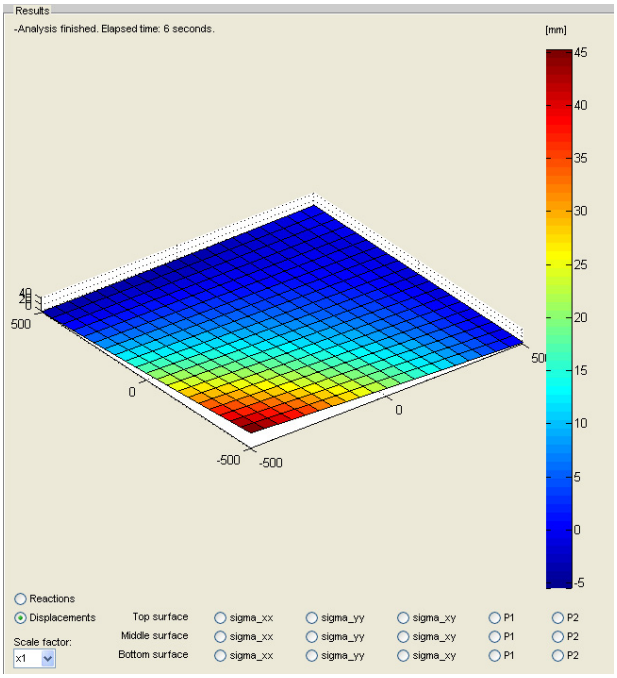


Figure 5.5: Results panel

As can be seen from the radio button group in Figure 5.5, the stresses are divided over three surfaces; the top surface, the middle surface and the bottom surface. For each layer, the normal stresses σ_{xx} and σ_{yy}, the shear stresses σ_{xy} and the largest principal stress P_1 and smallest principal stress P_2 can be selected.

5.2 Preprocessor

The preprocessor generates several objects from the input data. These objects contain information which is needed for the calculation process. The source code of the preprocessor reads:

```
% set element type
ElementData = setElementData();

% generate mesh
Mesh = generateMesh(InputData, ElementData);

% set material model
ModelData = setModelData(InputData);

% generate boundary conditions
BcData = generateBoundaryConditions(InputData, Mesh, ElementData);

% generate points for extrapolation
EpData = generateExtrapolationPoints(Mesh, ElementData);
```

All the functions within the preprocessor will be dealt with separately in the next subsections.

5.2.1 Set element type

The function `setElementData` returns the object `ElementData` in which information is stored about the order of the elements and the type of Gauss integration scheme. This information includes the number of nodes and the number of integration points. The specific order of the elements and integration scheme will be determined in the next chapter. The source code of the function `setElementData` is given in Appendix D.1.1.

5.2.2 Generating a mesh

The function `generateMesh` generates an object `Mesh` containing the locations of the nodes and a connectivity matrix. In the following example an arbitrarily plate geometry - defined by by the locations of four corner nodes and an element size - is meshed with a nine node element*. Consider the plate in Figure 5.6b. The nodes of the mesh are numbered from the bottom left corner to the top right corner. For each row of nodes a plate length l_i is defined and for each column of nodes a plate width b_i is defined. The distance in x direction between the nodes within each row can be computed from:

$$\mathrm{d}x_i = \frac{l_i}{2n} \tag{5.1}$$

where n is the number of elements over the length of the plate defined by:

$$n = \frac{l_1}{el} \tag{5.2}$$

*Generating a mesh with a four node element is done in an almost similar fashion and will therefore not be elaborated.

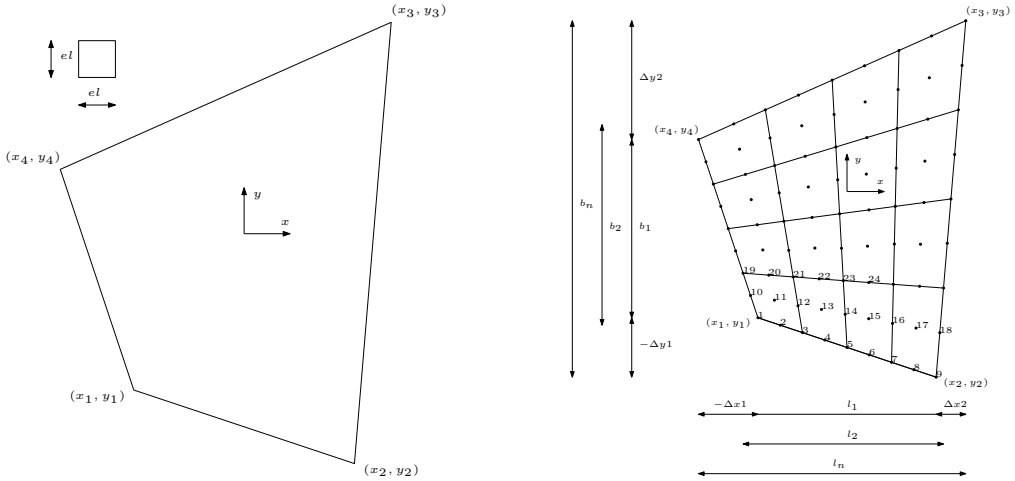


Figure 5.6: (a) Arbitrary plate geometry (b) Mesh with the nine node element configuration

The distance between the nodes in y direction is computed in a similar manner:

$$\mathrm{d}y_i = \frac{b_i}{2k} \tag{5.3}$$

with

$$k = \frac{b_1}{el} \tag{5.4}$$

The x locations of the nodes on the first row are:

$$\begin{aligned}
\mathrm{node}_{1x} &= x_1 \\
\mathrm{node}_{2x} &= x_1 + \mathrm{d}x_1 \\
\mathrm{node}_{3x} &= x_1 + 2\mathrm{d}x_1 \\
&\vdots \\
\mathrm{node}_{nx} &= x_1 + 2n\mathrm{d}x_1
\end{aligned}$$

The y locations of the nodes on the first row are:

$$\begin{aligned}
\mathrm{node}_{1y} &= y_1 \\
\mathrm{node}_{2y} &= y_1 + \frac{\Delta y1}{2k} \\
\mathrm{node}_{3y} &= y_1 + 2\frac{\Delta y1}{2k} \\
&\vdots \\
\mathrm{node}_{ky} &= y_1 + 2n\frac{\Delta y1}{2k}
\end{aligned}$$

For the second row the x locations of the nodes are:

$$\begin{aligned}
\text{node}_{10x} &= x_1 + \frac{\Delta x1}{2n} \\
\text{node}_{11x} &= x_1 + \frac{\Delta x1}{2n} + \text{d}x_2 \\
\text{node}_{12x} &= x_1 + \frac{\Delta x1}{2n} + 2\text{d}x_2 \\
&\vdots \\
\text{node}_{(10+2n)x} &= x_1 + \frac{\Delta x1}{2n} + 2n\text{d}x_2
\end{aligned}$$

For the second row the y locations of the nodes are:

$$\begin{aligned}
\text{node}_{10y} &= y_1 + \text{d}y_1 \\
\text{node}_{11y} &= y_1 + \frac{\Delta y1}{2k} + \text{d}y_2 \\
\text{node}_{12y} &= y_1 + 2\frac{\Delta y1}{2k} + \text{d}y_2 \\
&\vdots \\
\text{node}_{(10+2k)y} &= y_1 + 2k\frac{\Delta y1}{2k} + \text{d}y_2
\end{aligned}$$

The method described above can easily be implemented in a loop resulting in a matrix containing the locations of the nodes:

$$\texttt{Mesh.x} = \begin{bmatrix} \text{node}_{1x} & \text{node}_{2x} & \text{node}_{3x} & \text{node}_{4x} & \text{node}_{5x} & \cdots & \text{node}_{2n+1\times 2k+1x} \\ \text{node}_{1y} & \text{node}_{2y} & \text{node}_{3y} & \text{node}_{4y} & \text{node}_{5y} & \cdots & \text{node}_{2n+1\times 2k+1y} \end{bmatrix} \tag{5.5}$$

This is shown in the first part of the source code of `generateMesh` given in Appendix D.1.2 The second important part of the function `generateMesh` is constructing the connectivity matrix. This matrix contains the node numbers of each separate element and is especially important for assembling the system matrices and system vectors from the element matrices and element vectors. Starting again at the bottom left corner, for each element the nodenumbers are stored in a separate row of the connectivity matrix. For a nine node element, first the element corner nodes are gathered, then the mid nodes and finally the middle node, all in a counterclockwise manner. For the mesh given in Figure 5.6b the connectivity matrix reads:

$$\texttt{Mesh.connect} = \begin{bmatrix} 1 & 3 & 21 & 19 & 2 & 12 & 20 & 10 & 11 \\ 3 & 5 & 23 & 21 & 4 & 14 & 22 & 12 & 13 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \tag{5.6}$$

Again this can easily be implemented in a loop. The source code of constructing the connectivity matrix is given in the second part of `generateMesh` in Appendix D.1.2.

5.2.3 Set material model

The function `setModelData` generates the plate stiffness matrices (for both bending and shear) and the membrane stiffness matrix and stores them in the object `ModelData`. The source code of this function is given in Appendix D.1.3

5.2.4 Generate boundary conditions

The function `generateBoundaryConditions` returns `BcData` in which the boundary conditions for both the membrane part and the plate part are stored. For the membrane part the boundary conditions are set conform Figure 5.7.

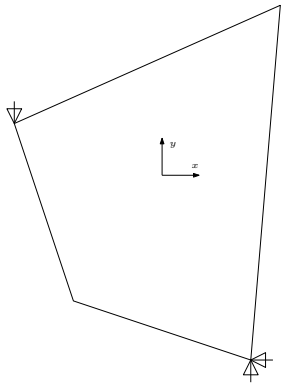


Figure 5.7: Boundary conditions for membrane

Irrespective of the geometry of the panel or the locations of the user prescribed vertical displacements, no horizontal reaction forces can emerge and displacements in x and y direction are prevented.
The boundary conditions for the plate part are obtained from the `InputData` object. Depending on the boundary condition (points or frame) a matrix is filled with the prescribed nodes and the corresponding prescribed displacement. The source code of the function `generateBoundaryConditions` is given in Appendix D.1.4.

5.2.5 Generate extrapolation points

The function `generateExtrapolationPoints` returns an object `EpData` in which three 'rings' of nodes are stored. This information is needed later on in the kernel to extrapolate some results that cannot be calculated directly. In Figure 5.8 the tree rings, colored in red blue and green, are shown for the nine node element*.

*Generating the object `EpData` with a four node element is done in an almost similar fashion and will therefore not be elaborated.

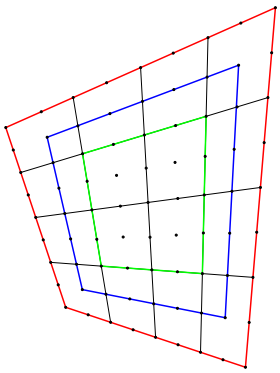


Figure 5.8: Nodes to be stored for extrapolation purposes

The source code of this function is given in D.1.5

5.3 Kernel

When the preprocessor has processed the input data, the kernel is started. In this part of the application the actual finite element calculation takes place. The source code of the kernel is given by:

```
% assemble constant stiffness matrix plate
K_p = assembleMatrixPlate(Mesh, ElementData, ModelData);

% assemble constant stiffness matrix membrane
K_m = assembleMatrixMembrane(Mesh, ElementData, ModelData);

% assemble external displacement vector plate
F_ext = assembleVectorPlate(Mesh, ElementData);

% apply boundary conditions plate
[K_p1, F_ext] = applyBoundaryConditions(K_p, F_ext, BcData.plate, ElementData.dof_p);

% set maximum number of iterations
it = 100;

% set convergence tolerance
eps = 1e-4;

% set counter
jj = 0;

% iterations
while  jj==0 || norm(da_p)/norm(a_p_1)>eps && jj<it

        % solve system of equations for plate
        da_p = K_p1\(F_ext-F_int);

        % add the correction to the displacement vector
        a_p = a_p + da_p;

        % assemble extra strain vector A_star
```

```
        A_star = assembleA_star(Mesh, a_p);

        % compute extra force vector
        F_t_star = computeF_t_star(Mesh, ElementData,  ModelData, A_star);

        % apply boundary conditions membrane
        [K_m1, F_t_star] = applyBoundaryConditions(K_m, F_t_star, BcData.membrane, ElementData.dof_m);

        % solve system of equations for membrane
        a_m = K_m1\-F_t_star;

        % compute membrane stresses
        Sigma_m = computeMembraneStress(Mesh, ElementData, a_m , A_star, ModelData, EpData);

        % compute curvature
        Kappa = computeCurvature(Mesh, ElementData, EpData, a_p);

        % compute membrane forces in z direction
        F_s_star = computeF_s_star(Mesh, ModelData, ElementData, Sigma_m, Kappa, BcData, EpData);

        % compute internal forcevector
        F_int = K_p1*a_p - F_s_star;

        % save internal force vector 1st iteration
        if jj==0
            a_p_1 = a_p;
        end
        jj=jj+1;
end
```

The source code for the kernel corresponds to the iterative solution scheme presented in section 4.3. The next subsections will outline the functions appearing in the kernel.

5.3.1 Assembling the system stiffness matrices

The function `assembleMatrixPlate` assembles the system stiffness matrix from the element stiffness matrices. First an empty system matrix is set up. The nodes of the plate part each have 3 degrees of freedom thus the system stiffness matrix has dimensions of $3n \times 3n$, where n resembles the number of nodes of the complete system [10]. Filling this system stiffness matrix is done by looping over all the elements in order to compute each element stiffness matrix and subsequently adding each contribution to the complete system. Computing each specific element stiffness matrix again requires a loop. This loop takes place over the Gauss points within each element in behalf of numerical integration. The previous assertion is clarified in Figure 5.9 for a four node element with 2×2 integration.

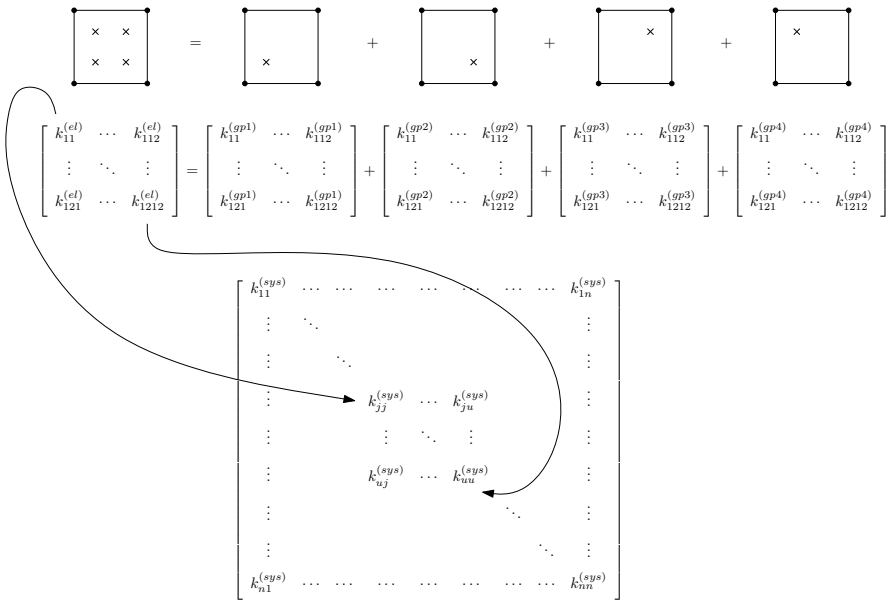


Figure 5.9: Assembling matrices for a four node element with 2×2 integration

In Appendix D.2.1, the source code for this procedure is given.
The same procedure may be applied to the membrane elements, with the difference that the element stiffness matrices have 2 degrees of freedom per node. In Appendix D.2.2, the source code for this procedure is given.

5.3.2 Assembling the external load vector

In function `assembleF_ext` the external load vector `F_ext` is assembled. This is done by setting up an empty system vector of size $3n \times 1$. The external force vector `F_ext` is filled with the external loading in terms of prescribed displacements when it is passed to the function `applyBoundaryConditions`. This will be explained in the next subsection.

5.3.3 Apply boundary conditions for the plate

The functions `applyBoundaryConditions` adjusts the stiffness matrices and the force vectors to comply with the boundary conditions adopted in the object `BcData`. Suppose a constant system stiffness matrix (for either the plate part or the membrane part) is given by:

$$\begin{bmatrix} k_{11} & k_{12} & \cdots & k_{1i} & \cdots & k_{1n} \\ k_{21} & k_{22} & \cdots & k_{2i} & \cdots & k_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ k_{i1} & k_{i2} & \cdots & k_{ii} & \cdots & k_{in} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ k_{n1} & k_{n2} & \cdots & k_{ni} & \cdots & k_{nn} \end{bmatrix} \tag{5.7}$$

If at node i a certain displacement is prescribed, all elements in row number i of the system stiffness matrix are set to 0, except for element k_{ii}, which is set to 1. The system stiffness matrix now reads:

$$\begin{bmatrix} k_{11} & k_{12} & \cdots & k_{1i} & \cdots & k_{1n} \\ k_{21} & k_{22} & \cdots & k_{2i} & \cdots & k_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ k_{n1} & k_{n2} & \cdots & k_{ni} & \cdots & k_{nn} \end{bmatrix} \tag{5.8}$$

All elements of the external load vector now represent imposed forces, except for the element on row i which represents an imposed displacement.

$$\begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_i \\ \vdots \\ f_n \end{bmatrix} \tag{5.9}$$

Since no externally applied forces are present all identities, except for identity i, remain (as constructed in `assembleF_ext`) zero. Solving the system yields a displacement vector containing the displacements of the free nodes a_f as well as the displacements of the prescribed nodes a_p.

$$\begin{bmatrix} a_{f_1} \\ a_{f_2} \\ \vdots \\ a_{p_i} \\ \vdots \\ a_{f_n} \end{bmatrix} = \begin{bmatrix} k_{11} & k_{12} & \cdots & k_{1i} & \cdots & k_{1n} \\ k_{21} & k_{22} & \cdots & k_{2i} & \cdots & k_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ k_{n1} & k_{n2} & \cdots & k_{ni} & \cdots & k_{nn} \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ f_i \\ \vdots \\ 0 \end{bmatrix} \tag{5.10}$$

For the iterative form, the previous example transforms to:

$$\begin{bmatrix} \mathrm{d}a_{f_1,j+1} \\ \mathrm{d}a_{f_2,j+1} \\ \vdots \\ \mathrm{d}a_{p_i,j+1} \\ \vdots \\ \mathrm{d}a_{f_n,j+1} \end{bmatrix} = \begin{bmatrix} k_{11} & k_{12} & \cdots & k_{1i} & \cdots & k_{1n} \\ k_{21} & k_{22} & \cdots & k_{2i} & \cdots & k_{2n} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ k_{n1} & k_{n2} & \cdots & k_{ni} & \cdots & k_{nn} \end{bmatrix}^{-1} \left(\begin{bmatrix} 0 \\ 0 \\ \vdots \\ f_{i,ext} \\ \vdots \\ 0 \end{bmatrix} - \begin{bmatrix} f_{1,int,j} \\ f_{2,int,j} \\ \vdots \\ f_{i,int,j} \\ \vdots \\ f_{n,int,j} \end{bmatrix} \right)$$

After the first iteration the internal force vector $f_{i,int}$ at node i represents the reaction force at that specific node. This value must be set to zero for all iterations. The source code of the function `applyBoundaryConditions` is given in Appendix D.2.3.

5.3.4 Iterations

After assembling the constant stiffness matrices for the plate part and the membrane part, the iteration process commences. The iterative procedure imposed by the while loop is repeated until the force norm has fulfilled the convergence tolerance or the maximum number of iterations is exceeded. In order to loop over the iteration process at least once, the counter `jj` may not be equal to `0`.

First, the system of equations for the plate part is solved. The obtained displacement vector `da_p` contains the nodal displacements and nodal rotations within an iteration. This contribution is subsequently added to the total displacement vector `a_p`
The extra strain vector `A_star` due to large deformations is assembled in `assembleA_star`. Conform equation (4.21) the elements of `A_star` are a function of the rotations which are known from the plate displacement vector `a_p`. The procedure is given in Appendix D.2.4.
The function `computeF_t_star` computes the extra force vector on the membrane part `F_t_star` from the extra strain vector `A_star`. This operation is analogous to (4.24). As in assembling the stiffness matrices, the procedure in `computeF_t_star` also requires looping over the elements and for each element looping over the Gauss points. The source code for this procedure is given in Appendix D.2.5.
From the extra force vector `F_t_star` the membrane displacement vector `a_m` can be determined by solving the system of equations for the membrane part. Next, the function `computeMembraneStress` computes the nodal values of the membrane stresses `Sigma_m` from the membrane displacement vector `u_m`. This is done analogous to (4.26). In order to obtain the **nodal** membrane stresses instead of the **element** membrane stresses, the value in the each Gauss point is interpolated to the nodes. This is illustrated in Figure 5.10a for a nine node element.

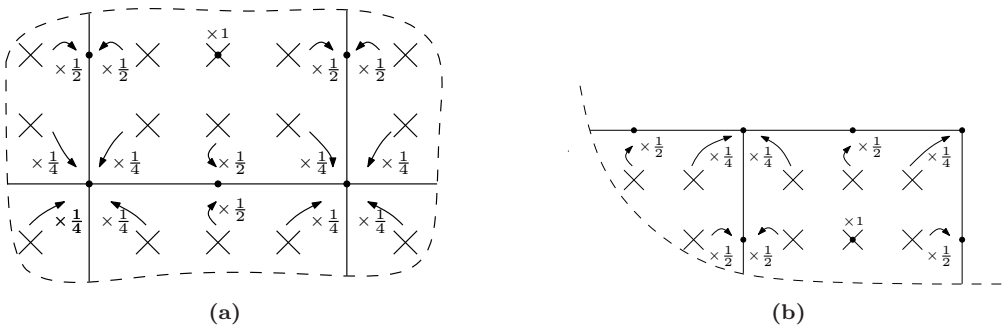


Figure 5.10: (a) Interpolation in the plate (b) Interpolation near the edges

As can be seen from Figure 5.10b interpolation near the edges will result in an underestimation of the node values. This can be corrected by applying extrapolation for the outer most nodes. To this end the (correct) values of the nodes of the second ring and third ring - obtained from the object `EpData` - are linearly extrapolated to the first ring. The source code for this procedure is given in Appendix D.2.6.
Analogous to equation (4.4), the curvatures are computed from the rotational part of the plate displacement vector `a_p`. Again transforming the element curvatures to nodal values requires interpolation and extrapolation. The source code for computing the nodal curvatures `Kappa` is given in Appendix D.2.7.
With the aid of the nodal membrane stresses `Sigma_m` and the nodal curvatures `Kappa` the membrane forces in z direction `F_s_star` can be computed. This is equivalent to the procedure in equation (4.29). The source code is given in Appendix D.2.8. The final step in the iteration loop is to compute the internal force vector `F_int` for the next iteration. In behalf of the displacement norm, the displacement vector of the first iteration is stored in `a_p_1` and the counter `jj` is raised by 1.

5.4 Postprocessor

When the while loop of the kernel has stopped, the solution is checked for convergence. If this is not the case, the `conv` flag is set to `false`, else the `conv` flag is set to `true` and the reaction forces and various stresses are computed. The source code of the postprocessor is given by:

```
% check convergence
if jj≥it || (norm(da_p)/norm(a_p_1))>eps

    % no convergence
    conv = false;

else

    % convergence
    conv = true;

    % compute reaction forces
```

```
    Reactions = computeReactions(Mesh, K_p, BcData, a_p);

    % compute bending stresses
    Sigma_b = computeBendingStress(Mesh, ModelData, Kappa);

    % compute total of membrane and bending stresses for each layer
    Sigma_t = computeTotalStress(Sigma_m, Sigma_b);

    % compute principal stresses for each layer
    Sigma_p = computePrincipalStress(Mesh, Sigma_m, Sigma_b);

end
```

5.4.1 Compute reaction forces

In `computeReactions` the reaction forces are computed. This is equivalent to computing the internal forces at the prescribed nodes. Suppose node i is prescribed, the internal force is computed by multiplying row i of the plate stiffness matrix by the displacement vector `a_p`. It is emphasized that only vertical reaction forces arise because: (1) no external membrane forces are applied and (2) the boundary conditions shown in Figure 5.7 on the membrane part do not allow the internal membrane stresses to develop reaction forces. The source code of `computeReactions` is given in Appendix D.3.1.

5.4.2 Compute stresses

The function `computeBendingStress` computes the bending stresses `Sigma_b` using the curvature `Kappa` and the constitutive model which is stored in `ModelData`. The source code is given in Appendix D.3.2. Next, in `computeTotalStress` the membrane stresses `Sigma_m` are added to the bending stresses `Sigma_b` to form the total stress `Sigma_t`. This procedure is given in Appendix D.3.3.
In `computePrincipalStress` the nodal principal stresses are computed from the membrane stresses `Sigma_m` and bending stresses `Sigma_b`. The source code for this function is given in Appendix D.3.4.

5.5 Deployment

The source code of PBA is deployed to a stand alone application with Mathworks MATLAB Builder. A great advantage is that users do not need to install MATLAB in order to use PBA. However, users do need to have the MATLAB Component Runtime (MCR) installed on their computer. The MCR is a free redistributable that allows users to run programs written in a specific version of MATLAB without installing the MATLAB version itself.
PBA and the MCR are available on the internet and can be used freely by others. For more information see: http://www.mechanics.citg.tudelft.nl/pba/

5.6 Summary

The finite element formulation presented in Chapter 4 was implemented in the MATLAB programming language. The source code of the various functions are collected in Appendix D. PBA is deployed to a stand alone application which, in combination with the MCR, can be used without installing MATLAB.

6. Validation of PBA

In the previous chapter the source code for PBA was presented. At that stage no choice was made regarding: (1) the order of the elements and Gauss integration scheme, (2) the convergence tolerance ϵ and (3) the maximum number of iterations needed. In this chapter these variables will be determined on basis of a comparison between results obtained with PBA and results obtained with Diana version 9.3.

6.1 Element order and mesh size

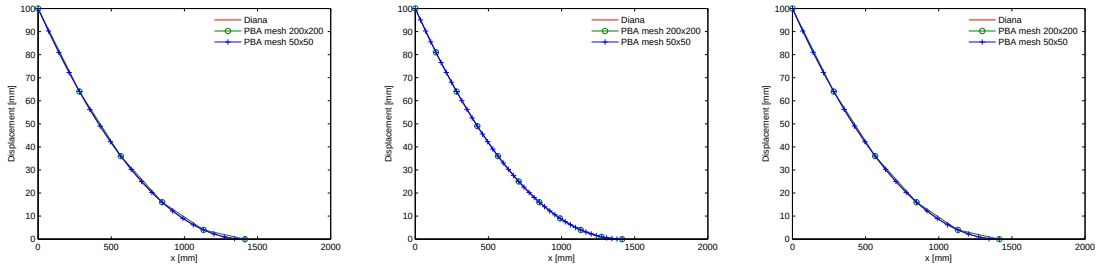
The performance of the linear element and the quadratic element is evaluated for bending action only [*]. Further, a recommendation for the mesh size is established. In order to assess the influence of the element order and mesh size a geometrical linear analysis is performed on the four benchmark geometries of Chapter 2. To this end linear displacement field of the diagonal line for each of the four geometries is examined for a relatively coarse mesh (200×200 [mm]) and a relatively fine mesh (50×50 [mm]).

From Figure 6.1 and 6.2 it can be concluded that both the linear and quadratic Reissner-Mindlin element show accurate results in comparison with Diana. For the other panels, displayed in Figure 6.3 and Figure 6.4, some differences are observed. The linear element shows an overly stiff behavior. This locking behavior is a well known problem for linear Reissner-Mindlin elements [6]. Since a quadratic element is up to 30 times more expensive in terms of computational time than a linear element[†], some effort is put into improving the linear element by applying a form of reduced integration. This will be explained in more detail in the next subsection. Because this effect is not observed for the square and rectangular panel it is thought that the element distortion plays a key component in this locking behavior.

Another conclusion drawn from Figure 6.1 to Figure 6.4 is that the mesh size has a minor influence on the accuracy of the solution at the nodes. However, a finer mesh means more data points and therefore a better representation of the deformed shape.

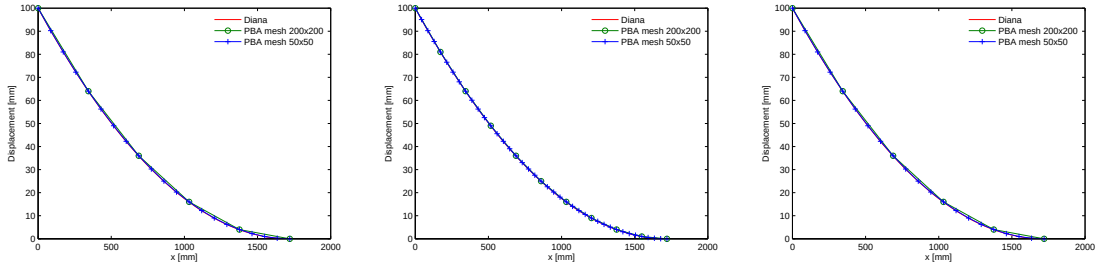
[*]The linear membrane element and quadratic membrane element both show no deviations with Diana whatsoever, hence the assessment of membrane action is omitted

[†]This is caused by: (1) a larger system of equations need to be solved and (2) evaluating the quadratic shape functions requires more time



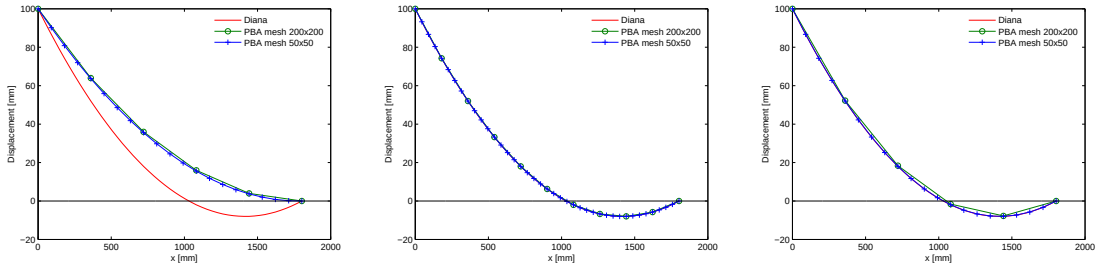
(a) Linear RM and Diana **(b)** Quadratic RM and Diana **(c)** Linear red. int. RM and Diana

Figure 6.1: Displacement diagrams square panel



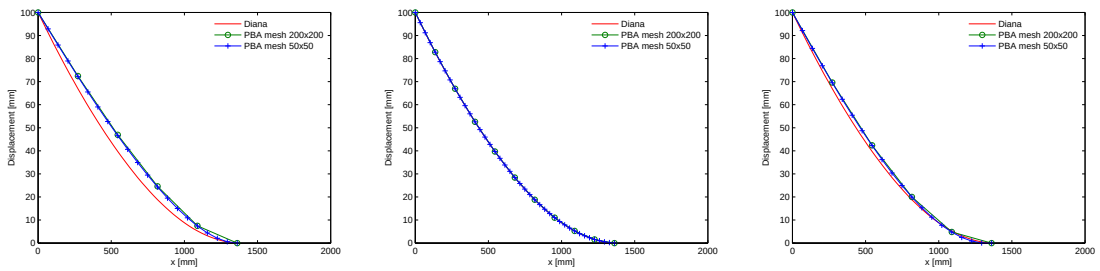
(a) Linear RM and Diana **(b)** Quadratic RM and Diana **(c)** Linear red. int. RM and Diana

Figure 6.2: Displacement diagrams rectangular panel



(a) Linear RM and Diana **(b)** Quadratic RM and Diana **(c)** Linear red. int. RM and Diana

Figure 6.3: Displacement diagrams parallelogram panel



(a) Linear RM and Diana **(b)** Quadratic RM and Diana **(c)** Linear red. int. RM and Diana

Figure 6.4: Displacement diagrams trapezoid panel

6.1.1 Reduced integration

The stiff response of linear Reissner-Mindlin element is presumably caused by shear locking. This phenomenon is observed when relatively thin plates are analyzed. The contribution of the shear deformations to the energy should vanish but this is not necessarily always the case. To overcome this problem one can apply reduced integration to one or more terms of the stiffness matrix [5]. These terms are integrated using a Gauss scheme of one order lower than is used normally. There are two forms of reduced integration possible: full reduced integration and selective reduced integration. In full reduced integration both the bending and shear terms are under-integrated. In selective reduced integration at least one of the shear terms is under-integrated. In order to enhance the performance of the linear element a number of different reduced integration schemes are examined in this section.
As stated in Chapter 4, the stiffness matrix of the Reissner-Mindlin element is given by:

$$\boldsymbol{K^{w\theta}} = \begin{bmatrix} \boldsymbol{k}^{ww} & \boldsymbol{k}^{w\theta} \\ \boldsymbol{k}^{\theta w} & \boldsymbol{k}^{\theta\theta} \end{bmatrix} \tag{6.1}$$

where

$$\boldsymbol{k}^{ww} = \int_A \boldsymbol{B}^{w^T} \boldsymbol{C} \boldsymbol{B}^{w} \, \mathrm{d}A \tag{6.2}$$

$$\boldsymbol{k}^{w\theta} = -\int_A \boldsymbol{B}^{w^T} \boldsymbol{C} \boldsymbol{N}^{\theta} \, \mathrm{d}A \tag{6.3}$$

$$\boldsymbol{k}^{\theta w} = -\int_A \boldsymbol{N}^{w^T} \boldsymbol{C} \boldsymbol{B}^{w} \, \mathrm{d}A \tag{6.4}$$

$$\boldsymbol{k}^{\theta\theta} = \int_A \boldsymbol{B}^{\theta^T} \boldsymbol{D}_b \boldsymbol{B}^{\theta} \, \mathrm{d}A + \int_A \boldsymbol{N}^{\theta^T} \boldsymbol{C} \boldsymbol{N}^{\theta} \, \mathrm{d}A \tag{6.5}$$

For convenience $\boldsymbol{k}^{\theta\theta}$ is subdivided in $\boldsymbol{k}^{\theta\theta^1}$ for the first term and $\boldsymbol{k}^{\theta\theta^2}$ for the second term. The various reduced integration schemes to examine are presented in Table 6.1

	$\boldsymbol{k}^{ww}$	$\boldsymbol{k}^{w\theta}$	$\boldsymbol{k}^{\theta w}$	$\boldsymbol{k}^{\theta\theta^1}$	$\boldsymbol{k}^{\theta\theta^2}$
Full red.	1×1	1×1	1×1	1×1	1×1
Sel. red. 1	1×1	1×1	1×1	2×2	1×1
Sel. red. 2	2×2	2×2	2×2	2×2	1×1
Sel. red. 3	2×2	1×1	1×1	2×2	1×1

Table 6.1: Various reduced integration schemes

The performance of the linear Reissner-Mindlin element for the various reduced integration schemes is displayed in Figure 6.5. This figure shows the displacement of the diagonal line for the parallelogram.

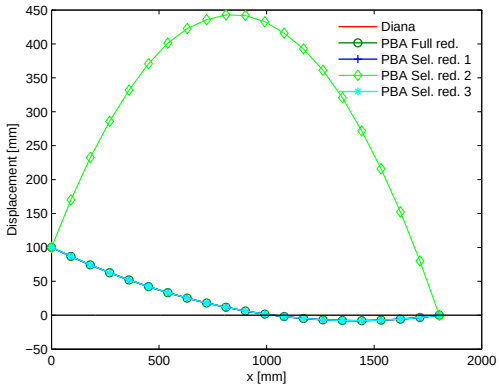
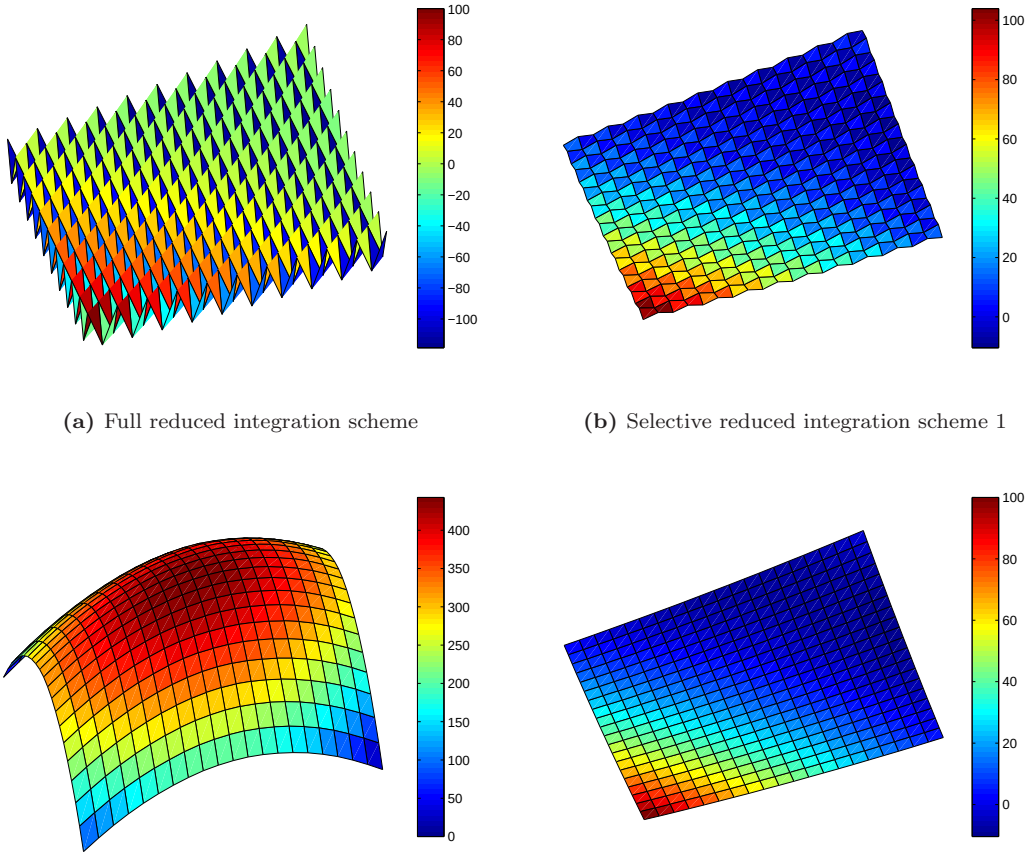


Figure 6.5: Displacement field of parallelogram with various reduced integration schemes applied on the linear Reissner-Mindlin element

As can be observed from Figure 6.5, most of the reduced integration schemes show an improved behavior in comparison with the normal integrated linear Reissner-Mindlin element which is displayed in Figure 6.3a. However, the total deformation of the plate show radical differences. Figure 6.6 displays the deformations of the parallelogram for the various reduced integration schemes.



(a) Full reduced integration scheme

(b) Selective reduced integration scheme 1

(c) Selective reduced integration scheme 2

(d) Selective reduced integration scheme 3

Figure 6.6: Deformations w [mm] of parallelogram shaped panel for different reduced integration schemes

When full reduced integration is applied, the displacement field shows an oscillating pattern. This is caused by a poorly conditioned stiffness matrix meaning that very small and very large identities are present in the matrix [9]. Inversion of the poorly conditioned stiffness matrix - needed to solve the system of equations - results in an almost singular matrix. The displacement field for the second integration scheme (Figure 6.6b) shows the same pattern though with less extreme spikes. This pattern is also caused by a poorly conditioned stiffness matrix.
From the displacement field presented in Figure 6.6c it may be concluded that spurious modes arise. This means deformations can occur which do not contribute to the energy. Spurious modes are often encountered when reduced integration is applied.
Combination of Figure 6.5 and 6.6 reveals that only selective reduced integration scheme 3 improves the performance of the Reissner-Mindlin element. Applying this integration scheme to all panels gives the displacement of the diagonal lines presented in Figure 6.1c to 6.4c. The reduced integrated linear element shows an improvement for all panels, still a somewhat stiff behavior is observed for the trapezoid.

6.1.2 Computational expense

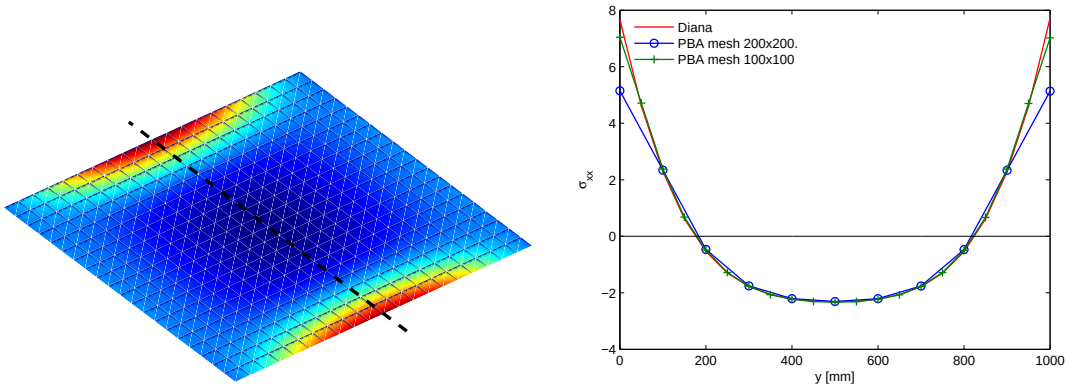
As stated before, the quadratic element is up to 30 times more expensive than the linear element in terms of computation time. This raises the question weather to choose the less accurate - but still reasonable - reduced integrated linear element over the quadratic element. However, the absolute computation time for the benchmark geometries, presented in Table 6.2, is still within reasonable limits. These values are obtained by running the application on a moderately standard machine with an element size of 100×100 [mm].

	Square	Rectangle	Parallelogram	Trapezoid
Linear element	0.009	0.002	0.009	0.013
Quadratic element	0.3	0.7	0.3	0.4

Table 6.2: Absolute computational time [s]

6.1.3 Concluding remarks

From the linear analysis performed in this section it can be concluded that the quadratic Reissner-Mindlin element with a normal 3×3 Gauss integration scheme is preferred over the linear Reissner-Mindlin element and the reduced integrated linear element. The computational time remains within acceptable limits and moreover, the performance is in better agreement with Diana. The linear and quadratic element is not examined for membrane action because both elements perform exactly similar to Diana.
The mesh size seems to have a minor influence on the accuracy of the solution at the nodes. However, changing to a geometrical nonlinear analysis, at some stage in the calculation process, extrapolation to the edges is required *. When a very coarse mesh is used, the extrapolation process returns inaccurate results near the edges. To illustrate this effect, a plot is made of the largest membrane stress σ_{xx} for a square panel (Figure 6.7).



(a) (b)

Figure 6.7: Membrane stress σ_{xx} for square panel for different mesh size

*The extrapolation process was explained in Section 5.3.4

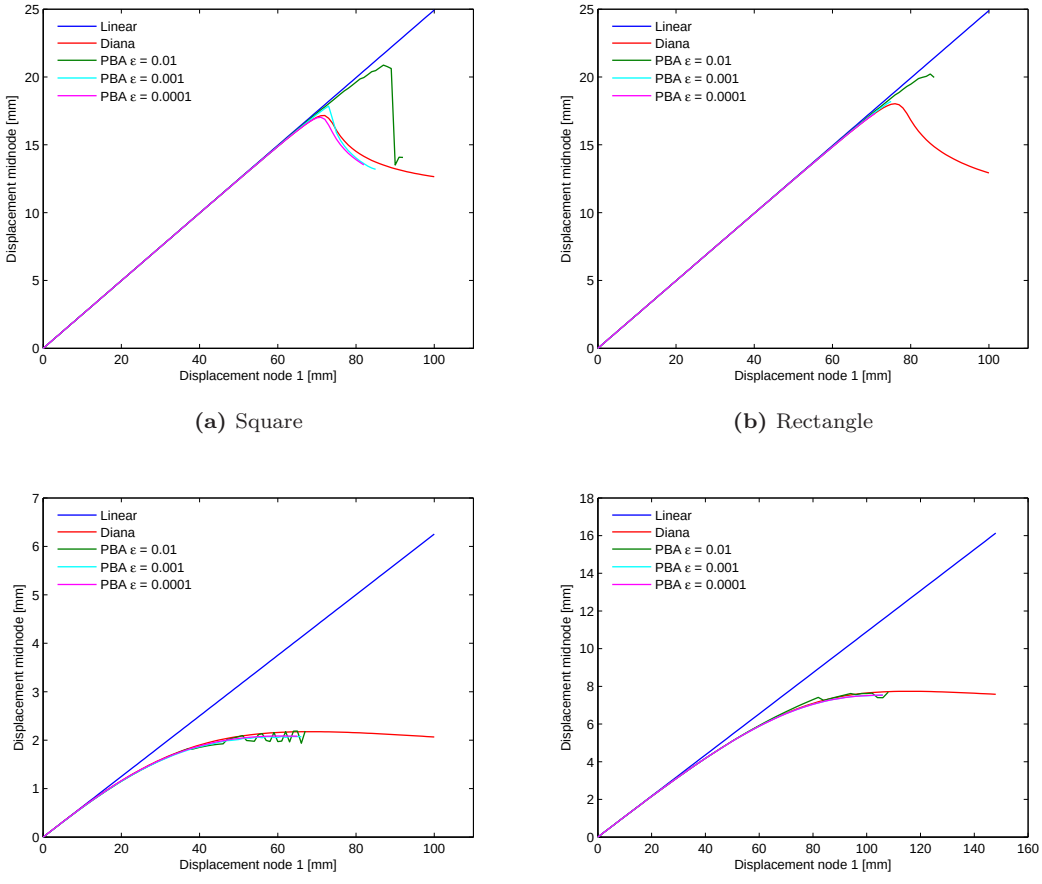
From this figure is concluded that a fine mesh is always preferred over a coarse mesh when the accuracy of the solution is to be optimized. However, Figure 6.7 shows that a reasonable accuracy can be achieved with a division of approximately 10 elements over the longest edge.

6.2 Iteration process

In the previous section a quadratic element for both bending and membrane action proved to render accurate results for a geometrical linear analysis. Further, a spatial discretization of approximately 10 elements over the longest edge was recommended. In this section the iteration process is evaluated in order to determine the convergence tolerance and the maximum number of iterations needed.

6.2.1 Convergence tolerance

The value of convergence tolerance of the displacement norm is determined by examining the load-displacement diagrams of the benchmark geometries from Chapter 2.



(a) Square

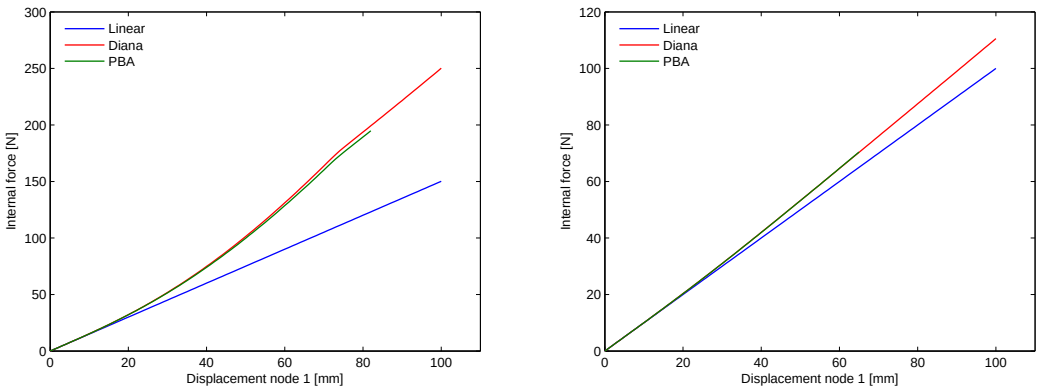
(b) Rectangle

(c) Parallelogram

(d) Trapezoid

Figure 6.8: Load-displacement diagram of the benchmark geometries for different ϵ

Figure 6.8 shows the load-displacement diagrams of the four geometries for different convergence tolerances *. For all benchmarks the strictest tolerance ($\epsilon = 0.0001$) renders the best load-displacement curves. However, for large imposed displacements the solution algorithm does not find convergence anymore. For all panels, except for the square panel, this appears to happen at the limit point of the load-displacement curves. Another way of mapping out the nonlinear behavior of a plate subjected to large displacements is by plotting the prescribed displacement against the internal (reaction) force at the prescribed node. Figure 6.9 shows this diagram for the square and the parallelogram shaped panel.

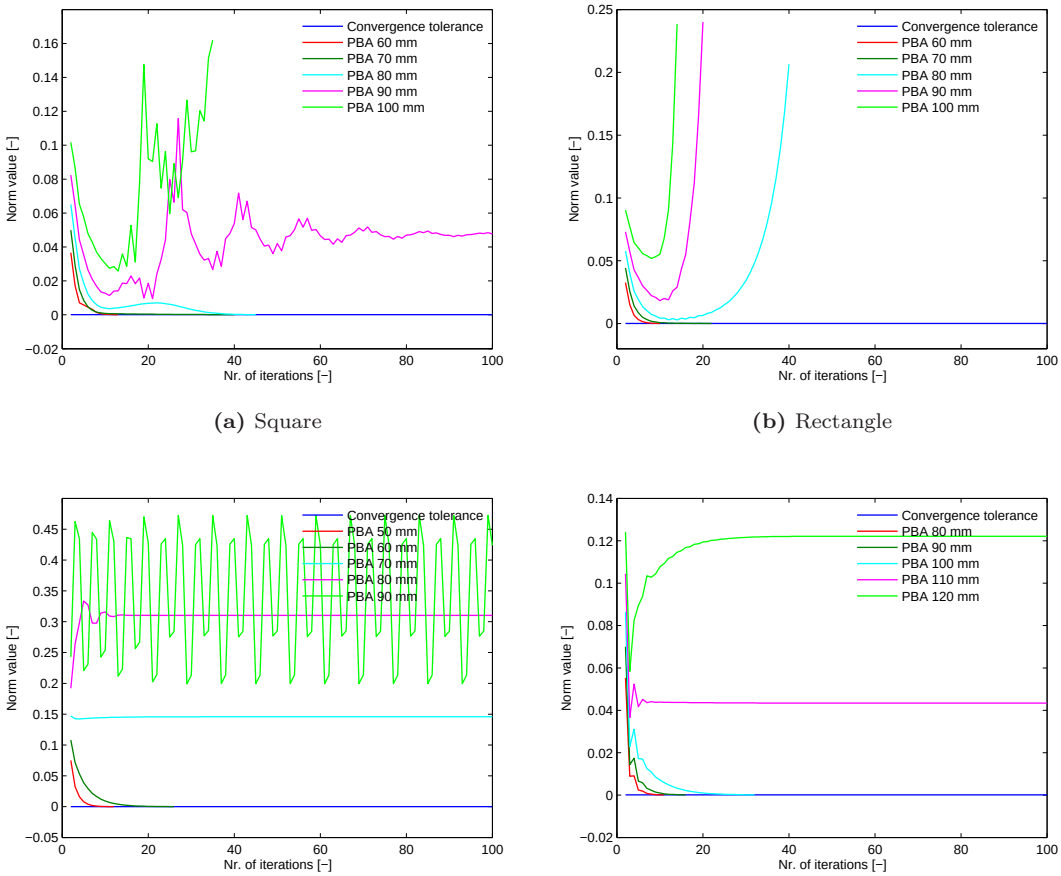


(a) Square **(b)** Parallelogram

Figure 6.9: Load-internal force diagram

Examining the internal reaction forces does not seem to explain why the solution algorithm fails to find convergence. Moreover, the limit point, which is clearly made visible in Figure 6.8, cannot be determined from a load-internal force diagram.
To further investigate divergence of the iterative process of PBA, the norm value is assessed. In Figure 6.10 the characteristic iteration patterns of the four benchmark panels for different imposed displacements are presented. These plots display the evolvement of the iteration process.

*The graphs in Figure 6.8 may suggest that an incremental-iterative calculation has been performed while this is not the case. The load-displacement diagrams are simply constructed by gathering data of a large amount of merely iterative calculations



(a) Square

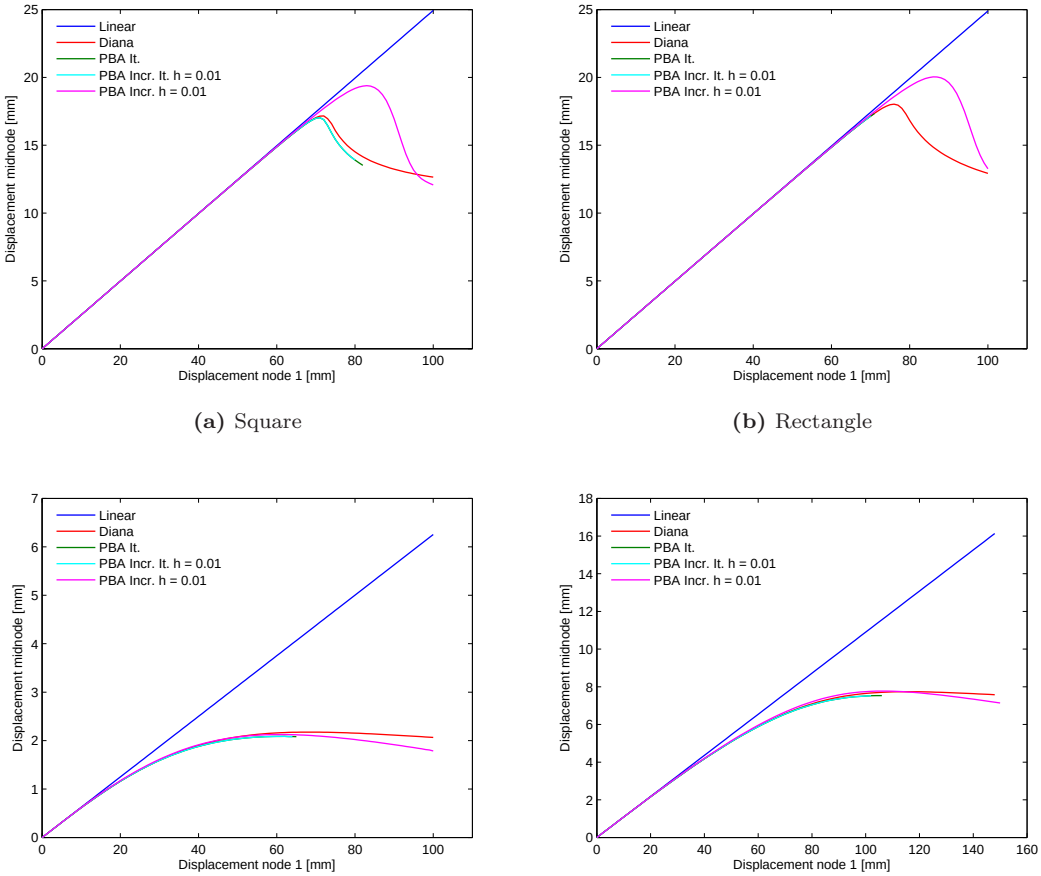
(b) Rectangle

(c) Parallelogram

(d) Trapezoid

Figure 6.10: Displacement norm value of the benchmarks for different imposed displacements

From Figure 6.10 can be concluded that the geometry of the plate has a great influence on the behavior of the norm value. However, there are some overall similarities observed. For all plate geometries holds that the convergence speed is rather large for the first couple of iterations. Also, when divergence arises - that is when the norm value increases - the solution method is not likely to find convergence in the following iterations. Only the graphs of the square panel and trapezoid panel show some resilience. For these panels the norm value may show a slight increase after which it decreases again. Why the solution method fails to find convergence beyond the limit point remains unclear. It is presumed that applying the load in one single step may cause a problem and that an incremental-iterative version of the linear stiffness method is more capable of dealing with local or global limit points. To verify this presumption PBA is modified to an incremental-iterative version. In addition a merely incremental version is constructed. Figure 6.11 displays the load-displacement diagrams of the four benchmark panels for an incremental scheme, an incremental-iterative scheme and the original iterative scheme.



(a) Square

(b) Rectangle

(c) Parallelogram

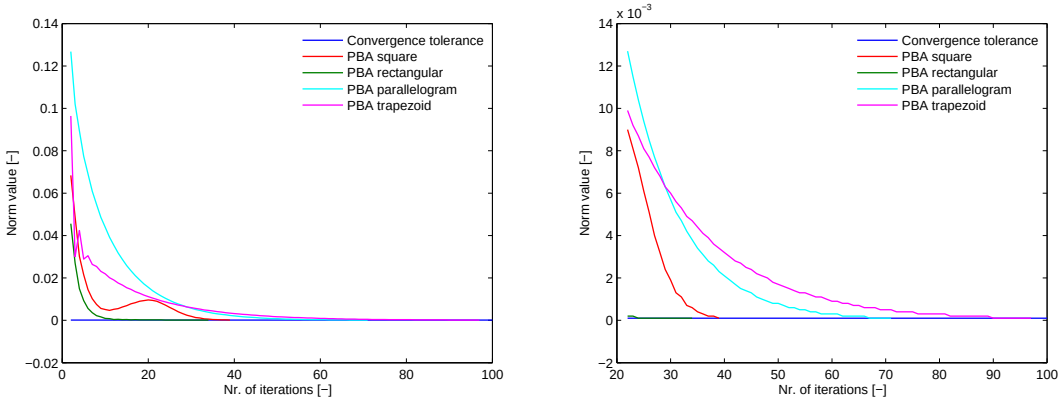
(d) Trapezoid

Figure 6.11: Load-displacement diagram of the benchmarks for different solution methods

The incremental-iterative version shows no improvement to the original iterative version whatsoever. Still the solution diverges at the limit point of the load-displacement diagrams. An incremental analysis proves to be capable of overcoming the limit point. However, the omitting of the equilibrium iterations leads to a meaningless solution since equilibrium is not satisfied at all time. From the previous statements can be concluded that it is the iterative process - with or without increments - that causes divergence. Some acceleration techniques exist to improve the convergence rate (e.g. arc length control, line search technique). Implementation of these techniques reach beyond the scope of this thesis and are therefore not attended.
It is assumed that for the majority of plate bending problems encountered in practice, the iterative version will suffice. Although this method is not suitable to map out the complete post buckling behavior, it yields good results up to divergence.

6.2.2 Number of iterations

In order to prevent PBA from continuously searching for a solution, a maximum number of iterations should be set. In Figure 6.12 the norm value is plotted for the benchmarks just before divergence arises.



(a) Domain [0-100] **(b)** Domain [20-100]

Figure 6.12: Norm value of the benchmark panels and convergence tolerance

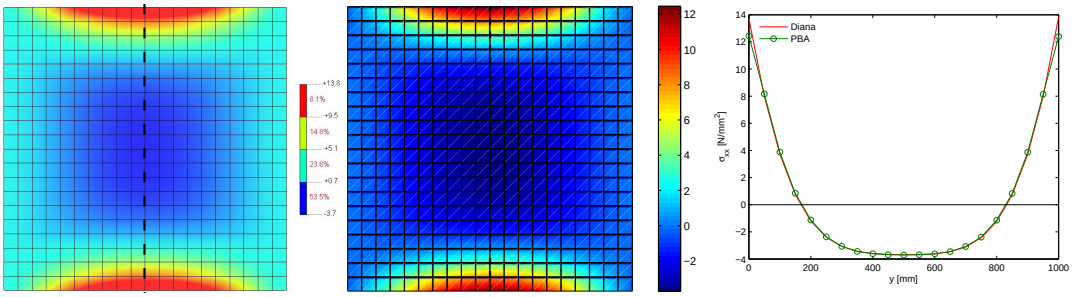
The maximum number of iterations observed is for the trapezoid. Setting the maximum number of iterations in PBA to 100 is thought to be sufficient for solving the majority of plate bending problems encountered in practice.

6.2.3 Concluding remarks

From the geometrical nonlinear analysis performed on the benchmark panels is concluded that an iterative process with a convergence tolerance for the displacement norm set to $\epsilon = 0.0001$ yields a solution which is in agreement with Diana. Although for large imposed displacements the solution algorithm is not able to find convergence, many of the practical plate bending problems can be solved. Further, it is established that a maximum number of iterations set to 100 is sufficient.

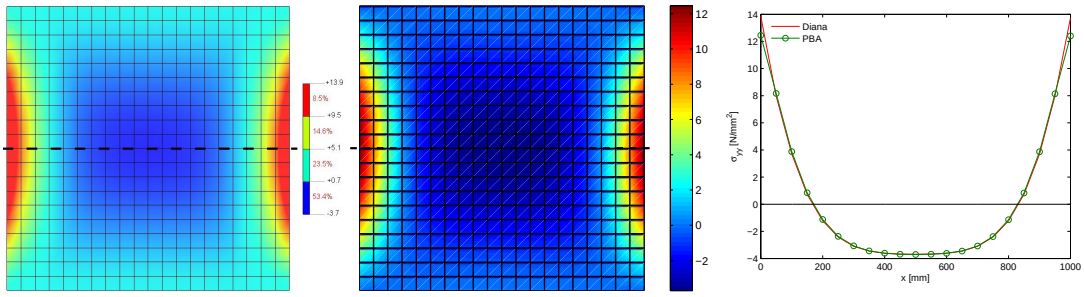
6.3 Verification of stresses

In this section the output of PBA for the stresses is validated. To this end the membrane stresses of the four benchmark geometries are compared to the membrane stresses obtained with Diana. The output of other stresses, i.e. the bending stresses and principal stresses, are not regarded. Figure 6.13 to 6.24 displays the membrane stresses obtained with PBA next to the membrane stresses obtained with Diana. In addition the membrane stresses over the indicated cross-section is shown. All membrane stresses are obtained from an analysis where the imposed displacement is maximized to a point where the solution method is just able to find convergence.



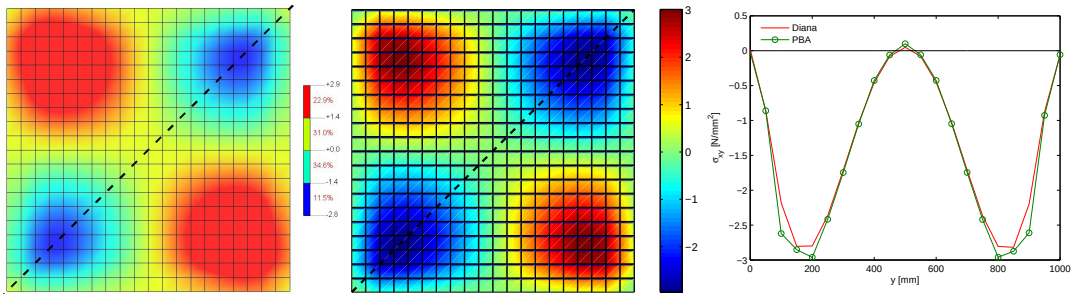
(a) (b) (c)

Figure 6.13: Membrane stress σ_{xx} [N/mm^2] (a) Diana (b) PBA (c) Cross-section



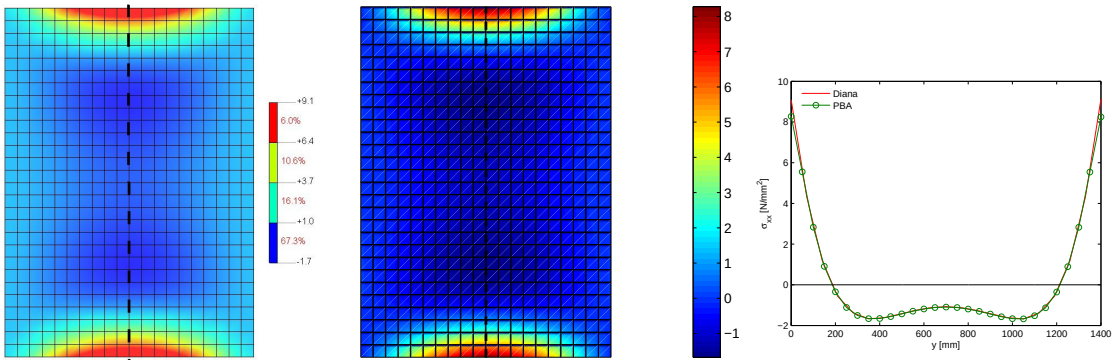
(a) (b) (c)

Figure 6.14: Membrane stress σ_{yy} [N/mm^2] (a) Diana (b) PBA (c) Cross-section



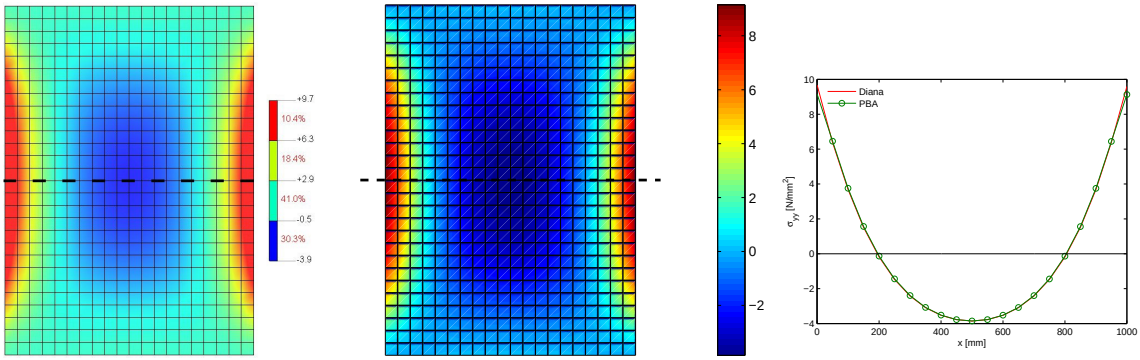
(a) (b) (c)

Figure 6.15: Membrane stress σ_{xy} [N/mm^2] (a) Diana (b) PBA (c) Cross-section



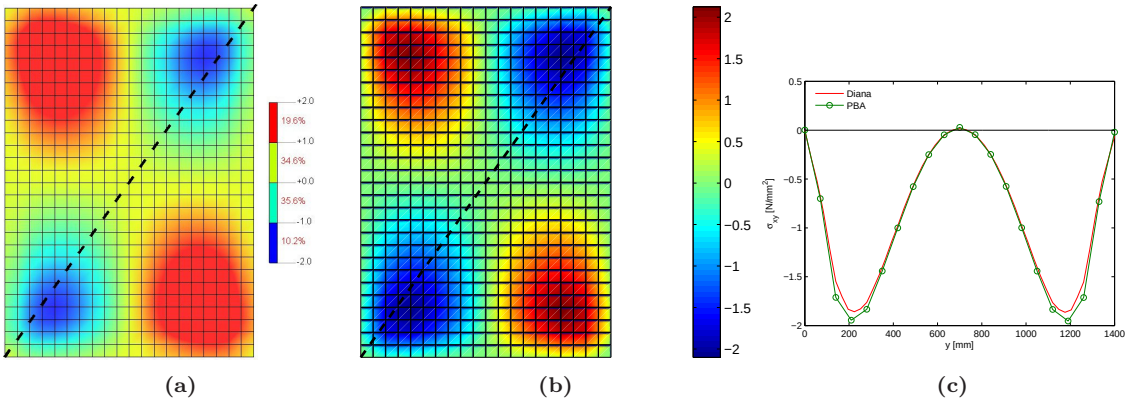
(a) (b) (c)

Figure 6.16: Membrane stress σ_{xx} [N/mm^2] (a) Diana (b) PBA (c) Cross-section



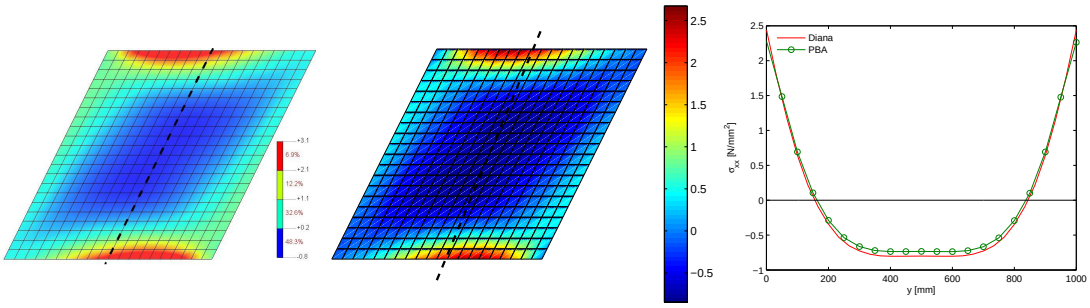
(a) (b) (c)

Figure 6.17: Membrane stress σ_{yy} [N/mm^2] (a) Diana (b) PBA (c) Cross-section



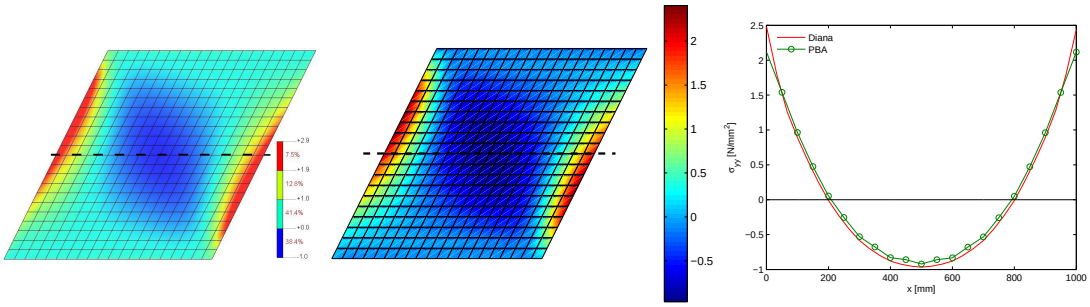
(a) (b) (c)

Figure 6.18: Membrane stress σ_{xy} [N/mm^2] (a) Diana (b) PBA (c) Cross-section



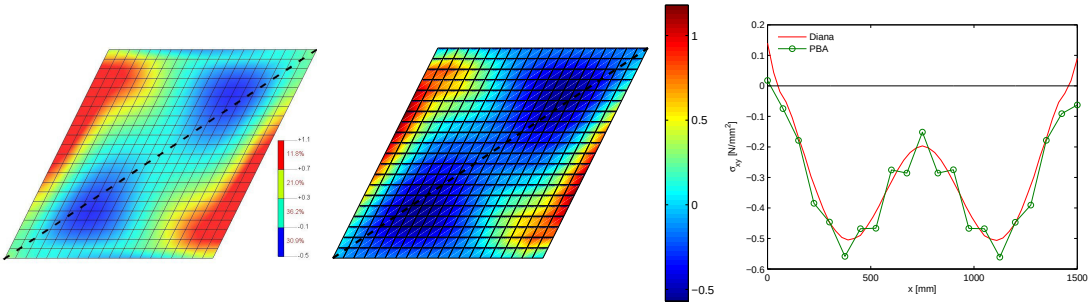
(a) σ_{xx} **(b)** σ_{xx} **(c)** σ_{xx}

Figure 6.19: Membrane stress σ_{xx} [N/mm^2] (a) Diana (b) PBA (c) Cross-section



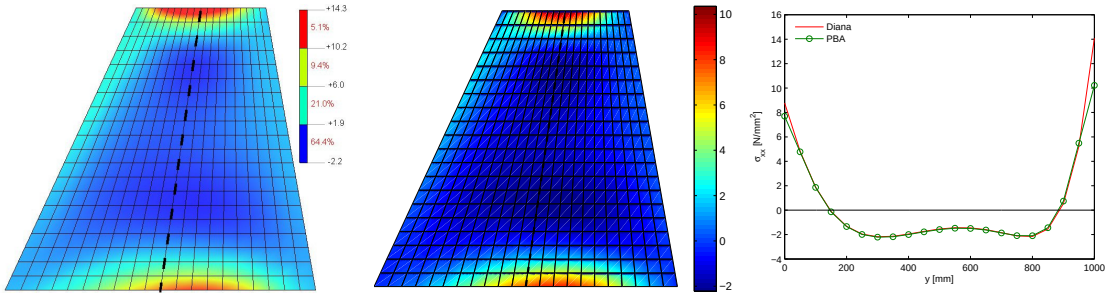
(a) σ_{yy} **(b)** σ_{yy} **(c)** σ_{yy}

Figure 6.20: Membrane stress σ_{yy} [N/mm^2] (a) Diana (b) PBA (c) Cross-section



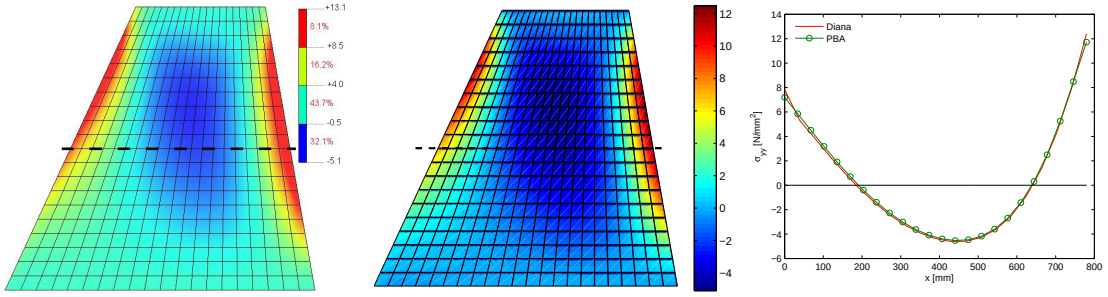
(a) σ_{xy} **(b)** σ_{xy} **(c)** σ_{xy}

Figure 6.21: Membrane stress σ_{xy} [N/mm^2] (a) Diana (b) PBA (c) Cross-section



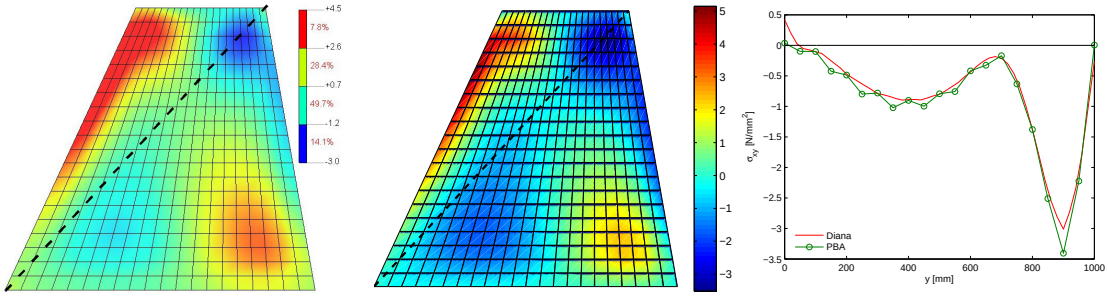
(a) σ_{xx} **(b)** σ_{xx} **(c)** σ_{xx}

Figure 6.22: Membrane stress σ_{xx} [N/mm^2] (a) Diana (b) PBA (c) Cross-section



(a) σ_{yy} **(b)** σ_{yy} **(c)** σ_{yy}

Figure 6.23: Membrane stress σ_{yy} [N/mm^2] (a) Diana (b) PBA (c) Cross-section



(a) σ_{xy} **(b)** σ_{xy} **(c)** σ_{xy}

Figure 6.24: Membrane stress σ_{xy} [N/mm^2] (a) Diana (b) PBA (c) Cross-section

The largest deviations are detected near the edges. Dependent on the geometry of the plate, deviations up to 20% are observed. This is primarily caused by the linear extrapolation process. It is presumed that these deviations decrease when a finer mesh is used than the recommended division of ten elements. Needless to say, the computational cost increases with a finer mesh. Doubling the number of elements increases the accuracy near the edges with approximately 10% but the computational cost becomes over 30 times larger.

7. Conclusions and recommendations

7.1 Conclusions

7.1.1 Mathematical model

A mathematical description for plates subjected to large displacements can be formulated using the classic membrane theory and the Reissner-Mindlin bending theory. The membrane-bending coupling behavior is achieved by two operations. For the membrane theory an extra contribution to the strains is added. For the Reissner-Mindlin theory the out of plane membrane forces are added to the equilibrium. The analytical description can be simply translated to a finite element formulation from which various elements can be developed. Other theories like the shell theory are more complex to translate to a finite element formulation since they are described in a three dimensional space. The developed membrane-bending theory is therefore an attractive alternative to model the behavior of flat plates subjected to large displacements.

7.1.2 Element performance

Two types of elements are developed: a four node linear element and a nine node quadratic element. For membrane action, both the linear and quadratic element with respectively 2×2 and 3×3 Gauss integration proves to perform well. Moreover, comparing the results with results obtained with Diana no deviations are observed whatsoever. For bending action the linear element shows an overly stiff behavior. The stiff behavior is caused by shear locking, meaning that for thin plates the contribution of the shear deformations to the energy does not vanish. Applying a form of reduced integration improves the performance but a slightly stiff response remains present. For bending action, the quadratic element with 3×3 Gauss integration proved to be the most accurate when compared to results obtained with Diana. For both the bending element and membrane element the quadratic element is selected to be implemented in PBA.

7.1.3 Solution technique

In order to solve the coupled membrane-bending finite element formulation an iterative solution procedure is developed based on the linear stiffness method. In order to detect convergence of the solution, the displacement norm proves to render accurate results for a convergence tolerance set to 0.0001 [-].
For large imposed displacements the solution method is not able to find convergence. Modifying the solution algorithm to an incremental-iterative process does not show any improvement. For a purely incremental method - eliminating the equilibrium iterations - a solution for large imposed displacements can be obtained. Since equilibrium is not satisfied at all time, this method yields inaccurate results. The iterative version of the linear stiffness method is selected to be implemented in PBA.

7.1.4 Implementation

MATLAB is very suitable for developing a special purpose finite element program. Designing a graphical user interface can be done easily. Moreover, visualizing the large amounts of data, which is often attended in finite element analysis, is very simple.

7.1.5 Stresses

A comparison between the membrane stresses obtained with PBA and the membrane stresses obtained with Diana is made. Deviations up to 20% are observed. This is mainly caused by linear extrapolation of membrane stresses near the edges. Refining the mesh by doubling the number of elements increases the accuracy by approximately 10% but the computational cost becomes over 30 times larger. It is established that a division of approximately 10 elements over the longest edge of an arbitrarily shaped plate gives a reasonable balance between computational cost and accuracy.

7.1.6 Relation to practice

PBA is useful tool for designers to give a first estimate of the deformations, stresses and reaction forces which arise from cold bending a flat panel to a curved shape. By having access to these quantities in an early stage of the development process, a designer can fully employ the potential of the used material without having to perform a comprehensive analysis. As a result, designs can become more daring and presumably less unexpected problems are encountered later on in the development process.

For large imposed displacements, PBA is not able to find a converged solution. This shortcoming would imply that designers cannot search for the margins in their designs. However, it is found that for the majority of plate bending problems encountered in practice, the maximum stress is the determining factor. In addition, for large imposed displacements the nonlinear deformations become excessive, causing undesired reflections.

7.2 Recommendations

PBA can be used to calculate and visualize the nonlinear deformations of a cold bent glass panel. From the deformed shape a designer cannot make a good assessment of the to be expected reflections. Adding an option to PBA which makes reflections visible could be a valuable extension for designers.

A possible new method for developing curved glass surfaces is cold bending of cylindrical hot bent glass. For this method a cylindrical hot bent panel is further deformed by cold bending. This method would open up new possibilities for developing curved glass surfaces. PBA is not able to model these panels since only flat plates can be described by the developed membrane-bending theory. Analyzing these plates would require a more complex shell theory.

PBA can be used to analyze quadrilateral shaped plates only. Other shapes (e.g. triangles, hexagons) cannot be modeled with PBA. Extending PBA to adopt other shapes requires a more advanced mesh generator but opens up more possibilities for developing curved glass surfaces.

A solution method based on the linear stiffness method is not able to find convergence when large displacements are imposed. A more advanced solution method which is capable of updating the stiffness matrix would presumably give a better convergence behavior. There are several updating methods available. Especially Newton Secant methods are of interest since they do not require to determine the actual stiffness of the deformed shape but use a form of numerical updating of the stiffness matrix.

The obtained stresses near the supports should be appreciated with great care, especially when point supports are considered. These stresses are very much dependent on the geometry of the used fitting. Unfortunately, it is not possible to model a fitting in PBA. More research could be done towards the influence of a specific fitting on the deformations and stresses.

Bibliography

[1] Belis, J. Cold bending of laminated glass panels. *HERON* (2007), 123–146.

[2] Blaauwendraad, J. Plates and slabs, 2003.

[3] Borst de, R., and Sluys, L. Computational methods in nonlinear solid mechanics, 2007.

[4] Bos, F. Towards a combined probabilistic/consequence-based safety approach of structural glass members. *HERON* (2007), 59–83.

[5] Cirak, F. Reissner-mindlin plates, http://www-g.eng.cam.ac.uk/csml/teaching/4d9/plates-2.pdf.

[6] Della Croce, L., and Scapolla, T. Hierarchic finite elements with selective and uniform redueed integration for reissner-mindlin plates. Tech. rep., University of Pavia, 1992.

[7] Doyle, J. *Nonlinear analyis of thin-walled structures.* Springer-Verlag New York, 2001.

[8] Eekhout, M. Design, engineering, production and realisation of glass structures for free-form architecture, 2004.

[9] Gavin, H. Mathematical properties of stiffness matrices, 2006.

[10] Graaf van de, A. Scriptie: Structural design of reinforced concrete pile caps - the strut-and-tie method extended with the stringer-panel method. Master's thesis, Technische Universiteit Delft Faculteit Civiele Techniek en Geowetenschappen, 2006.

[11] Herwijnen van, F. Warm en koud gebogen glas. *Cement* (2008), 32–37.

[12] Hori, and Muneo. Introduction to finite element method. Tech. rep., Earthquake Research Institute, 2000.

[13] Laar, v. H. Scriptie: Stabiliteit van in het werk vervormde glaspanelen voor dubbelgekromde gevels. Master's thesis, Technische Universiteit Delft Faculteit Civiele Techniek en Geowetenschappen, 2004.

[14] The Mathworks. *User's manual*, 2009.

[15] Staaks, D. Scriptie: Koud torderen van glaspanelen in blobs. Master's thesis, Technische Universiteit Eindhoven Faculteit Bouwkunde, 2003.

[16] Timoshenko, S., and Woinowsky-Krieger, S. *Theory of plates and shells, 2nd edition.* McGraw-Hill Auckland, 1989.

[17] Vakar, L. Cold bendable, laminated glass - new possibilities in design, 2004.

[18] Wells, G. The finite element method: An introduction, 2006.

[19] Witte de, F., and Kikstra, W., Eds. *DIANA User's manual release 8.1.* TNO building and construction research, 2002.

A. Elements

A.1 Four node element configuration

In this section a four node element is developed from the matrix notation given in section 4.2.3. The discretized fields for the membrane part are given by:

$$\boldsymbol{u} = \boldsymbol{N}^u \boldsymbol{a}^u \tag{A.1}$$

$$\boldsymbol{\epsilon} = \boldsymbol{B}^u \boldsymbol{a}^u + \boldsymbol{\theta}^* \tag{A.2}$$

For a four-node element the matrix $\boldsymbol{N}^u$ and the matrix $\boldsymbol{B}^u$ are given by

$$\boldsymbol{N}^u = \begin{bmatrix} N_1 & 0 & N_2 & 0 & N_3 & 0 & N_4 & 0 \\ 0 & N_1 & 0 & N_2 & 0 & N_3 & 0 & N_4 \end{bmatrix} \tag{A.3}$$

$$\boldsymbol{B}^u = \begin{bmatrix} N_{1,x} & 0 & N_{2,x} & 0 & N_{3,x} & 0 & N_{4,x} & 0 \\ 0 & N_{1,y} & 0 & N_{2,y} & 0 & N_{3,y} & 0 & N_{4,y} \\ N_{1,y} & N_{1,x} & N_{2,y} & N_{2,x} & N_{3,y} & N_{3,x} & N_{4,y} & N_{4,x} \end{bmatrix} \tag{A.4}$$

The various shape functions are given by:

$$N(x, y) = c_1 xy + c_2 y + c_3 x + c_4 \tag{A.5}$$

where c_1 to c_4 are constants depending on the shape and size of the element. The nodal values for the displacement field $\boldsymbol{a}^u$ is given by

$$\boldsymbol{a}^{u^T} = \begin{bmatrix} u_{x1} & u_{y1} & u_{x2} & u_{y2} & u_{x3} & u_{y3} & u_{x4} & u_{y4} \end{bmatrix} \tag{A.6}$$

The extra strain field in the membrane part due to bending is defined as

$$\boldsymbol{\theta}^* = \boldsymbol{N}^* \boldsymbol{a}^* \tag{A.7}$$

with the matrix $\boldsymbol{N}^*$ and the vector $\boldsymbol{a}^*$ given by

$$\boldsymbol{N}^* = \begin{bmatrix} N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & 0 & N_4 & 0 & 0 \\ 0 & N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & 0 & N_4 & 0 \\ 0 & 0 & N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & 0 & N_4 \end{bmatrix} \tag{A.8}$$

$$\boldsymbol{a}^{*^T} = \begin{bmatrix} \frac{1}{2}\theta_{x1}^2 & \frac{1}{2}\theta_{y1}^2 & \theta_{x1}\theta_{y1} & \frac{1}{2}\theta_{x2}^2 & \frac{1}{2}\theta_{y2}^2 & \theta_{x2}\theta_{y2} & \frac{1}{2}\theta_{x3}^2 & \frac{1}{2}\theta_{y3}^2 & \theta_{x3}\theta_{y3} & \frac{1}{2}\theta_{x4}^2 & \frac{1}{2}\theta_{y4}^2 & \theta_{x4}\theta_{y4} \end{bmatrix} \tag{A.9}$$

The material matrix for the membrane part can be written as

$$\boldsymbol{D}_m = \frac{Et}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix} \tag{A.10}$$

Using the above matrices, the internal force vector due to the extra strains $\boldsymbol{f}_{int,\theta^*}$ can be constructed from:

$$\boldsymbol{f}_{int,\theta^*} = [\int_A \boldsymbol{B}^{u^T} \boldsymbol{D}_m \boldsymbol{N}^* \, \mathrm{d}A]\boldsymbol{a}^* \tag{A.11}$$

Assuming the field $\boldsymbol{a}^*$ is known, the unknown membrane displacements can be solved from:

$$\boldsymbol{f}_{int,\theta^*} = \boldsymbol{K}^u \boldsymbol{a}^u \tag{A.12}$$

The discretized displacement field and the discretized rotational field for the bending element are given by

$$\boldsymbol{w} = \boldsymbol{N}^w \boldsymbol{a}^w \tag{A.13}$$
$$\boldsymbol{\theta} = \boldsymbol{N}^\theta \boldsymbol{a}^\theta \tag{A.14}$$

Taking the derivatives gives

$$\boldsymbol{w}_{,xy} = \boldsymbol{B}^w \boldsymbol{a}^w \tag{A.15}$$
$$\boldsymbol{\kappa} = \boldsymbol{B}^\theta \boldsymbol{a}^\theta \tag{A.16}$$

For a 4 node element the matrices $\boldsymbol{N}^w$ and $\boldsymbol{N}^\theta$ have the form

$$\boldsymbol{N}^w = \begin{bmatrix} N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & 0 & N_4 & 0 & 0 \end{bmatrix} \tag{A.17}$$

$$\boldsymbol{N}^\theta = \begin{bmatrix} 0 & N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & 0 & N_4 & 0 \\ 0 & 0 & N_1 & 0 & 0 & N_2 & 0 & 0 & N_3 & 0 & 0 & N_4 \end{bmatrix} \tag{A.18}$$

and the matrices $\boldsymbol{B}^w$ and $\boldsymbol{B}^\theta$

$$\boldsymbol{B}^w = \begin{bmatrix} N_{1,x} & 0 & 0 & N_{2,x} & 0 & 0 & N_{3,x} & 0 & 0 & N_{4,x} & 0 & 0 \\ N_{1,y} & 0 & 0 & N_{2,y} & 0 & 0 & N_{3,y} & 0 & 0 & N_{4,y} & 0 & 0 \end{bmatrix} \tag{A.19}$$

$$\boldsymbol{B}^\theta = \begin{bmatrix} 0 & N_{1,x} & 0 & 0 & N_{2,x} & 0 & 0 & N_{3,x} & 0 & 0 & N_{4,x} & 0 \\ 0 & 0 & N_{1,y} & 0 & 0 & N_{2,y} & 0 & 0 & N_{3,y} & 0 & 0 & N_{4,y} \\ 0 & N_{1,y} & N_{1,x} & 0 & N_{2,y} & N_{2,x} & 0 & N_{3,y} & N_{3,x} & 0 & N_{4,y} & N_{4,x} \end{bmatrix} \tag{A.20}$$

The various shape functions are given by:

$$N(x,y) = c_1xy + c_2y + c_3x + c_4 \tag{A.21}$$

The nodal values for the displacement and the nodal values for the nodal rotations $\boldsymbol{a}^w$ and $\boldsymbol{a}^\theta$

$$\boldsymbol{a}^{w^T} = \begin{bmatrix} w_1 & 0 & 0 & w_2 & 0 & 0 & w_3 & 0 & 0 & w_4 & 0 & 0 \end{bmatrix} \tag{A.22}$$

$$\boldsymbol{a}^{\theta^T} = \begin{bmatrix} 0 & \theta_{x1} & \theta_{y1} & 0 & \theta_{x2} & \theta_{y2} & 0 & \theta_{x3} & \theta_{y3} & 0 & \theta_{x4} & \theta_{y4} \end{bmatrix} \tag{A.23}$$

are combined to

$$\boldsymbol{a}^{w\theta^T} = \begin{bmatrix} w_1 & \theta_{x1} & \theta_{y1} & w_2 & \theta_{x2} & \theta_{y2} & w_3 & \theta_{x3} & \theta_{y3} & w_4 & \theta_{x4} & \theta_{y4} \end{bmatrix} \tag{A.24}$$

The generalized forces $\boldsymbol{f}^w$ and $\boldsymbol{f}^\theta$

$$\boldsymbol{f}^{w^T} = \begin{bmatrix} f_{w1} & 0 & 0 & f_{w2} & 0 & 0 & f_{w3} & 0 & 0 & f_{w4} & 0 & 0 \end{bmatrix} \tag{A.25}$$

$$\boldsymbol{f}^{\theta^T} = \begin{bmatrix} 0 & f_{\theta_{x1}} & f_{\theta_{y1}} & 0 & f_{\theta_{x2}} & f_{\theta_{y2}} & 0 & f_{\theta_{x3}} & f_{\theta_{y3}} & 0 & f_{\theta_{x4}} & f_{\theta_{y4}} \end{bmatrix} \tag{A.26}$$

are combined to

$$\boldsymbol{f}^{w\theta^T} = \begin{bmatrix} f_{w1} & f_{\theta_{x1}} & f_{\theta_{y1}} & f_{w2} & f_{\theta_{x2}} & f_{\theta_{y2}} & f_{w3} & f_{\theta_{x3}} & f_{\theta_{y3}} & f_{w4} & f_{\theta_{x4}} & f_{\theta_{y4}} \end{bmatrix} \tag{A.27}$$

The internal force vector containing the nodal membrane forces in z direction is given by

$$\boldsymbol{f}_{int,\sigma^*} = \begin{bmatrix} f_{int,\sigma_1^*} & 0 & 0 & f_{int,\sigma_2^*} & 0 & 0 & f_{int,\sigma_3^*} & 0 & 0 & f_{int,\sigma_4^*} & 0 & 0 \end{bmatrix} \tag{A.28}$$

with

$$f_{int,\sigma_i^*} = \int_A n_{xx_i}\kappa_{xx_i}\,\mathrm{d}A + \int_A n_{yy_i}\kappa_{yy_i}\,\mathrm{d}A + \int_A n_{xy_i}2\kappa_{xy_i}\,\mathrm{d}A \tag{A.29}$$

The material matrices can be written as

$$\boldsymbol{D}_b = \frac{Et^3}{12(1-\nu^2)} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \dfrac{1-\nu}{2} \end{bmatrix} \tag{A.30}$$

$$\boldsymbol{C} = \frac{Et}{2+2\nu} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{A.31}$$

Assuming the vector $\boldsymbol{f}_{int,\sigma^*}$ is known, the unknown displacement field and rotational field can be solved from:

$$\boldsymbol{K}\boldsymbol{a}^{w\theta} = \boldsymbol{f}^{w\theta} + \boldsymbol{f}_{int,\sigma^*} \tag{A.32}$$

with

$$\boldsymbol{K} = \int_A \boldsymbol{B}^{w^T}\boldsymbol{C}\boldsymbol{B}^{w}\,\mathrm{d}A - \int_A \boldsymbol{B}^{w^T}\boldsymbol{C}\boldsymbol{N}^{\theta}\,\mathrm{d}A - \int_A \boldsymbol{N}^{w^T}\boldsymbol{C}\boldsymbol{B}^{w}\,\mathrm{d}A + \int_A \boldsymbol{B}^{\theta^T}\boldsymbol{D}_b\boldsymbol{B}^{\theta}\,\mathrm{d}A + \int_A \boldsymbol{N}^{\theta^T}\boldsymbol{C}\boldsymbol{N}^{\theta}\,\mathrm{d}A \tag{A.33}$$

A.2 Nine node element configuration

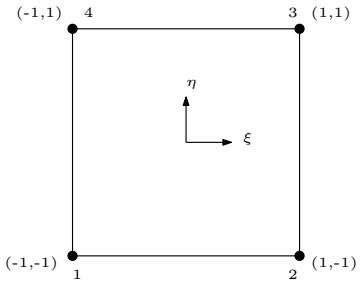
The exact same procedure is adopted as was done for the four node element configuration. Since the matrices and vectors become rather large, this section will only deal with the shape functions. The shape functions of a nine node element are given by:

$$N(x,y) = c_1x^2y^2 + c_2y^2x + c_3x^2y + c_4y^2 + c_5x^2 + c_6xy + c_7y + c_8x + c_9 \tag{A.34}$$

where c_1 to c_9 are constants depending on the shape and size of the element.

B. Isoparametric shape functions

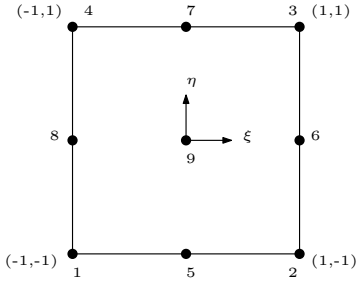
B.1 Four node element



$$N_i(\xi,\eta) = \frac{1}{4}(1+\xi_i\xi)(1+\eta_i\eta)$$

Figure B.1: Four node isoparametric element

B.2 Nine node element

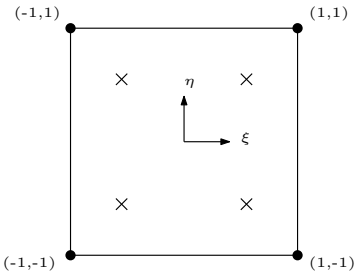


$$N_1(\xi,\eta) = \frac{1}{4}\xi\eta(1-\xi)(1-\eta)$$
$$N_2(\xi,\eta) = -\frac{1}{4}\xi\eta(1+\xi)(1-\eta)$$
$$N_3(\xi,\eta) = \frac{1}{4}\xi\eta(1+\xi)(1+\eta)$$
$$N_4(\xi,\eta) = -\frac{1}{4}\xi\eta(1-\xi)(1+\eta)$$
$$N_5(\xi,\eta) = -\frac{1}{2}\eta(1-\eta)(1-\xi^2)$$
$$N_6(\xi,\eta) = \frac{1}{2}\xi(1+\xi)(1-\eta^2)$$
$$N_7(\xi,\eta) = \frac{1}{2}\eta(1+\eta)(1-\xi^2)$$
$$N_8(\xi,\eta) = -\frac{1}{2}\xi(1-\xi)(1-\eta^2)$$
$$N_9(\xi,\eta) = (1-\xi^2)(1-\eta^2)$$

Figure B.2: Nine node isoparametric element

C. Gauss integration

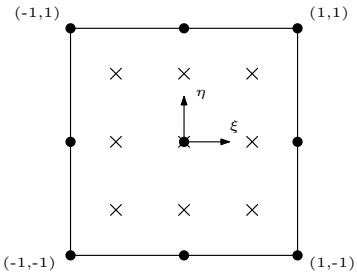
C.1 2×2 integration scheme



points n	location ξ_i,η_i	weight w_i
4	$-\frac{1}{\sqrt{3}}$, $-\frac{1}{\sqrt{3}}$	1
	$\frac{1}{\sqrt{3}}$, $-\frac{1}{\sqrt{3}}$	1
	$\frac{1}{\sqrt{3}}$, $\frac{1}{\sqrt{3}}$	1
	$-\frac{1}{\sqrt{3}}$, $\frac{1}{\sqrt{3}}$	1

Figure C.1: Gauss integration scheme; 2×2 integration

C.2 3×3 integration scheme



points n	location ξ_i,η_i	weight w_i
9	$-\sqrt{\frac{3}{5}}$, $-\sqrt{\frac{3}{5}}$	$\frac{25}{81}$
	$\sqrt{\frac{3}{5}}$, $-\sqrt{\frac{3}{5}}$	$\frac{25}{81}$
	$\sqrt{\frac{3}{5}}$, $\sqrt{\frac{3}{5}}$	$\frac{25}{81}$
	$-\sqrt{\frac{3}{5}}$, $\sqrt{\frac{3}{5}}$	$\frac{25}{81}$
	0 , $-\sqrt{\frac{3}{5}}$	$\frac{40}{81}$
	$\sqrt{\frac{3}{5}}$, 0	$\frac{40}{81}$
	0 , $\sqrt{\frac{3}{5}}$	$\frac{40}{81}$
	$-\sqrt{\frac{3}{5}}$, 0	$\frac{40}{81}$
	0 , 0	$\frac{64}{81}$

Figure C.2: Gauss integration scheme; 3×3 integration

D. Source code

D.1 Preprocessor

D.1.1 Set element type

```
function ElementData = setElementData()

% degrees of freedom per node Reissner–Mindlin element
ElementData.dof_p = 3;

% degrees of freedom per node membrane element
ElementData.dof_m = 2;

% nodes per element
ElementData.nodes = 9;

% number of integration points per element
ElementData.noInt =  9;

end
```

D.1.2 Mesh generator

```
function Mesh = generateMesh(InputData, ElementData)

% location of corner nodes
node1_x = InputData.cornerX(1);
node1_y = InputData.cornerY(1);
node2_x = InputData.cornerX(2);
node2_y = InputData.cornerY(2);
node3_x = InputData.cornerX(3);
node3_y = InputData.cornerY(3);
node4_x = InputData.cornerX(4);
node4_y = InputData.cornerY(4);

% Δ
Δ_x1 = node4_x - node1_x;
Δ_y1 = node2_y - node1_y;
Δ_x2 = node3_x - node2_x;
Δ_y2 = node3_y - node4_y;

% length 1
l_1 = node2_x - node1_x;

% width 1
b_1 = node4_y - node1_y;

% element size
```

```
el = InputData.el;

% number of elements l1
n = round(l_1/el);

% number of elements b2
k = round(b_1/el);

% number of nodes over length
k_l = 2*n+1;

% number of nodes over width
k_b = 2*k+1;

% generate mesh matrix
x = zeros(k_b*k_l,2);

% fill
for p=1:k_b
    ell=(l_1+((p-1)*(Δ_x2-Δ_x1))/(k_b-1))/(2*n);
    for q=1:k_l
        elb=(b_1 +((q-1)*(Δ_y2-Δ_y1))/(k_l-1))/(2*k);
        x(q+p*k_l-k_l,1)=(q*ell) - ell + (p-1)*(Δ_x1)/(k_b-1) + node1_x;    %x
        x(q+p*k_l-k_l,2)=(p*elb) - elb + (q-1)*(Δ_y1)/(k_l-1) + node1_y;    %y
    end
end x=x';

% generate connectivity matrix
connect=zeros(n*k, ElementData.nodes);

for p=1:k

    % column 1&2
    for q=1:n
        connect((p-1)*n+q,1)=(p-1)*(2*k_l)   +2*q-1;
        connect((p-1)*n+q,2)=(p-1)*(2*k_l)   +2*q+1;
    end

    % column 3&4
    for q=1:n
        connect((p-1)*n+q,3)=p*(2*k_l)       +2*q+1;
        connect((p-1)*n+q,4)=p*(2*k_l)       +2*q-1;
    end

    % column 5&6
    for q=1:n
        connect((p-1)*n+q,5)=(p-1)*(2*k_l)       +2*q;
        connect((p-1)*n+q,6)=(p-1)*(2*k_l) +k_l+2*q+1;
    end

    % column 7&8
    for q=1:n
        connect((p-1)*n+q,7)=p*(k_l+k_l)         +2*q;
        connect((p-1)*n+q,8)=(p-1)*(2*k_l) +k_l+2*q-1;
    end

    % column 9
```

```
    for q=1:n
        connect((p-1)*n+q,9)=(p-1)*(2*k_l)    +k_l+2*q;
    end

end connect=connect';

% store
Mesh.connect=connect;
Mesh.x=x;
Mesh.noNodes=size(x, 2);
Mesh.noElements=size(connect, 2);
Mesh.el=el;
Mesh.k_l=k_l;
Mesh.k_b=k_b;

Mesh.corner1=1;
Mesh.corner2=k_l;
Mesh.corner3=Mesh.connect(3,end);
Mesh.corner4=Mesh.corner3-k_l+1;
Mesh.midnode=round((Mesh.corner3-1)/2+1);

end
```

D.1.3 Set material model

```
function ModelData = setModelData(InputData)

% elasticity modulus
E=InputData.E; ModelData.E=E;

% Poisson's ratio
nu=InputData.nu;

% shear modulus
k=E/(2+2*nu); red=InputData.red;

% thickness
t=InputData.t; ModelData.t=t;

% bending stiffness
ModelData.D = E*t^3/(12*(1-nu^2)) *[1   nu  0
                                    nu  1   0
                                    0   0   (1-nu)/2];
% shear stiffness
ModelData.C = red*k*t             *[1  0
                                    0  1];
% membrane stiffness
ModelData.Dmem = E*t/(1-nu^2)     *[1   nu  0
                                    nu  1   0
                                    0   0   (1-nu)/2];
end
```

D.1.4 Generate boundary conditions

```
function BcData = generateBoundaryConditions(InputData, Mesh, ElementData)

pl=InputData.pl;
bctype=InputData.bctype;
noBc=length(InputData.bc)-1;
BcData.noBc=noBc;

% membrane part
BcData.membrane = [
    Mesh.corner2  1  1  0.0
    Mesh.corner2  1  2  0.0
    Mesh.corner3  1  1  0.0
    ]';

% plate part
switch bctype
    case 'points'
        noBc=length(InputData.bc)-1;
        BcData.noBc=noBc;
        for i=1:noBc
            meshPointer(i) = find(Mesh.x(1,:)==InputData.bc(1+i,1) & Mesh.x(2,:)==InputData.bc(1+i,2));
        end

        BcData.plate=zeros(1,1);
        for i=1:noBc
            BcData.plate(i,1)=meshPointer(i);
            BcData.plate(i,2)=1;
            BcData.plate(i,3)=1;
            BcData.plate(i,4)=InputData.nodez(i);
        end
        BcData.plate(noBc+1,1)=Mesh.midnode;
        BcData.plate(noBc+1,2)=0;
        BcData.plate(noBc+1,3)=1;
        BcData.plate(noBc+1,4)=pl;

    case 'topbottom'

        BcData.plate=zeros(1,1);

        p=length(BcData.plate(:,1))-1;

        for i=1:Mesh.k_l
            BcData.plate(p+i,1)=i;
            BcData.plate(p+i,2)=1;
            BcData.plate(p+i,3)=1;
            BcData.plate(p+i,4)=InputData.Framez(1)+
            (InputData.Framez(2)-InputData.Framez(1))/(Mesh.k_l-1)*(i-1);
        end

        p=length(BcData.plate(:,1));

        for i=1:Mesh.k_l
            BcData.plate(p+i,1)=(i)+Mesh.corner4-1;
            BcData.plate(p+i,2)=1;
            BcData.plate(p+i,3)=1;
            BcData.plate(p+i,4)=InputData.Framez(4)+
            (InputData.Framez(3)-InputData.Framez(4))/(Mesh.k_l-1)*(i-1);
```

```
        end

        % midnode
        p=length(BcData.plate(:,1));

        BcData.plate(p+1,1)=Mesh.midnode;
        BcData.plate(p+1,2)=0;
        BcData.plate(p+1,3)=1;
        BcData.plate(p+1,4)=pl;

        noBc=length(BcData.plate)-1;
        BcData.noBc=noBc;

    case 'leftright'

        BcData.plate=zeros(1,1);

        p=length(BcData.plate(:,1))-1;

        for i=1:Mesh.k_b
            BcData.plate(p+i,1)=(i)*Mesh.k_l -(Mesh.k_l-1);
            BcData.plate(p+i,2)=1;
            BcData.plate(p+i,3)=1;
            BcData.plate(p+i,4)=InputData.Framez(1)+
            (InputData.Framez(4)-InputData.Framez(1))/(Mesh.k_b-1)*(i-1);
        end

        p=length(BcData.plate(:,1));

        for i=1:Mesh.k_b
            BcData.plate(p+i,1)=(i)*Mesh.k_l;
            BcData.plate(p+i,2)=1;
            BcData.plate(p+i,3)=1;
            BcData.plate(p+i,4)=InputData.Framez(2)+
            (InputData.Framez(3)-InputData.Framez(2))/(Mesh.k_b-1)*(i-1);
        end

        % midnode
        p=length(BcData.plate(:,1));

        BcData.plate(p+1,1)=Mesh.midnode;
        BcData.plate(p+1,2)=0;
        BcData.plate(p+1,3)=1;
        BcData.plate(p+1,4)=pl;

        noBc=length(BcData.plate)-1;
        BcData.noBc=noBc;

    case 'round'
        BcData.plate=zeros(1,1);

        p=length(BcData.plate(:,1))-1;

        for i=1:Mesh.k_l
            BcData.plate(p+i,1)=i;
            BcData.plate(p+i,2)=1;
            BcData.plate(p+i,3)=1;
```

```
            BcData.plate(p+i,4)=InputData.Framez(1)+
            (InputData.Framez(2)-InputData.Framez(1))/(Mesh.k_l-1)*(i-1);
        end

        p=length(BcData.plate(:,1));

        for i=1:Mesh.k_l
            BcData.plate(p+i,1)=(i)+Mesh.corner4-1;
            BcData.plate(p+i,2)=1;
            BcData.plate(p+i,3)=1;
            BcData.plate(p+i,4)=InputData.Framez(4)+
            (InputData.Framez(3)-InputData.Framez(4))/(Mesh.k_l-1)*(i-1);
        end
        p=length(BcData.plate(:,1));

        for i=1:Mesh.k_b
            BcData.plate(p+i,1)=(i)*Mesh.k_l -(Mesh.k_l-1);
            BcData.plate(p+i,2)=1;
            BcData.plate(p+i,3)=1;
            BcData.plate(p+i,4)=InputData.Framez(1)+
            (InputData.Framez(4)-InputData.Framez(1))/(Mesh.k_b-1)*(i-1);
        end

        p=length(BcData.plate(:,1));

        for i=1:Mesh.k_b
            BcData.plate(p+i,1)=(i)*Mesh.k_l;
            BcData.plate(p+i,2)=1;
            BcData.plate(p+i,3)=1;
            BcData.plate(p+i,4)=InputData.Framez(2)+
            (InputData.Framez(3)-InputData.Framez(2))/(Mesh.k_b-1)*(i-1);
        end

        % midnode
        p=length(BcData.plate(:,1));

        BcData.plate(p+1,1)=Mesh.midnode;
        BcData.plate(p+1,2)=0;
        BcData.plate(p+1,3)=1;
        BcData.plate(p+1,4)=pl;

        noBc=length(BcData.plate)-1;
        BcData.noBc=noBc;

end

BcData.plate=BcData.plate';

end
```

D.1.5 Generate extrapolation data

```
function ExtrapolationData = generateExtrapolationPoints(Mesh)

ExtrapolationData.corner = [
    Mesh.corner1
```

```
    Mesh.corner4
    Mesh.corner2
    Mesh.corner3
    ]';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ExtrapolationData.ring1=zeros(1,1);

%bc1 1e ring

p=length(ExtrapolationData.ring1(1,:))-1;

for i=1:Mesh.k_l
    ExtrapolationData.ring1(1,p+i)=i;
end

p=length(ExtrapolationData.ring1(1,:));

for i=1:Mesh.k_b
    ExtrapolationData.ring1(1,p+i)=(i)*Mesh.k_l -(Mesh.k_l-1);
end

p=length(ExtrapolationData.ring1(1,:));

for i=1:Mesh.k_b
    ExtrapolationData.ring1(1,p+i)=(i)*Mesh.k_l;
end

p=length(ExtrapolationData.ring1(1,:));

for i=1:Mesh.k_l
    ExtrapolationData.ring1(1,p+i)=(i)+Mesh.corner4-1;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ExtrapolationData.ring2=zeros(1,1);

%bc1 2e ring

p=length(ExtrapolationData.ring2(1,:))-1;

for i=1:Mesh.k_l
    ExtrapolationData.ring2(1,p+i)=i+Mesh.k_l;
end

p=length(ExtrapolationData.ring2(1,:));

for i=1:Mesh.k_b
    ExtrapolationData.ring2(1,p+i)=(i)*Mesh.k_l -(Mesh.k_l-1)+1;
end

p=length(ExtrapolationData.ring2(1,:));

for i=1:Mesh.k_b
    ExtrapolationData.ring2(1,p+i)=(i)*Mesh.k_l-1;
end
```

```
p=length(ExtrapolationData.ring2(1,:));

for i=1:Mesh.k_l
    ExtrapolationData.ring2(1,p+i)=(i)+(Mesh.corner4-Mesh.k_l)-1;
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
ExtrapolationData.ring3=zeros(1,1);

%bc1 3e ring

p=length(ExtrapolationData.ring3(1,:))-1;

for i=1:Mesh.k_l
    ExtrapolationData.ring3(1,p+i)=i+2*Mesh.k_l;
end

p=length(ExtrapolationData.ring3(1,:));

for i=1:Mesh.k_b
    ExtrapolationData.ring3(1,p+i)=(i)*Mesh.k_l -(Mesh.k_l-1)+2;
end

p=length(ExtrapolationData.ring3(1,:));

for i=1:Mesh.k_b
    ExtrapolationData.ring3(1,p+i)=(i)*Mesh.k_l-2;
end

p=length(ExtrapolationData.ring3(1,:));

for i=1:Mesh.k_l
    ExtrapolationData.ring3(1,p+i)=(i)+(Mesh.corner4-2*Mesh.k_l)-1;
end

end
```

D.2 Kernel

D.2.1 Assembling stiffness matrix for plate

```
function A = assembleMatrixPlate(Mesh, ElementData,  ModelData)

N = Mesh.noNodes * ElementData.dof_p;
n = ElementData.nodes * ElementData.dof_p;

% allocate system matrix
A=zeros(N,N);

% allocate element matrix
A_e    = zeros(n);

% compute Gauss point locations and weights
gp = IntegrationScheme(ElementData.noInt);
```

```
% loop over all elements to assemble matrix
for element = Mesh.connect

    % set-up local node coordinates and connectivity
    connect_element = element(1:ElementData.nodes);
    x_element       = Mesh.x(:,connect_element);

    % loop over integration points to assemble element matrix
    for gauss_point = gp

        % compute shape functions
        [N, dN, j] = ShapeFunction(x_element, gauss_point);

        dX = gauss_point(1)*j;

        m = ElementData.dof_p;
        n = ElementData.nodes;

        D  = ModelData.D;
        C  = ModelData.C;

        % form B matrix for theta
        B_theta = zeros(3, m*n);
        B_theta(1, 2:m:end) = dN(:,1)';
        B_theta(2, 3:m:end) = dN(:,2)';
        B_theta(3, 2:m:end) = dN(:,2)';
        B_theta(3, 3:m:end) = dN(:,1)';

        % form N matrix for w
        N_w = zeros(1, m*n);
        N_w(1, 1:m:end) = N(:,1)';

        % form N matrix for theta
        N_theta = zeros(2, m*n);
        N_theta(1, 2:m:end) = N(:,1)';
        N_theta(2, 3:m:end) = N(:,1)';

        % form B matrix for w
        B_w = zeros(2, m*n);
        B_w(1, 1:m:end) = dN(:,1)';
        B_w(2, 1:m:end) = dN(:,2)';

        % various contributions to the element stiffness matrix
        ktt=B_theta'*D*B_theta;
        kee=N_theta'*C*N_theta;
        kvv=B_w'*C*B_w;
        ktv=- N_theta'*C*B_w;
        kvt=-B_w'*C*N_theta;

        % compute element stiffness matrix
        A_int = (ktt+kvt+ktv+kvv+kee)*dX;

        A_e = A_e + A_int;

    end

    % compute dof map
```

```
    dof_p = DofMap(ElementData.dof_p, connect_element, ElementData.nodes);

    % assemble
    A( dof_p, dof_p ) = A( dof_p, dof_p ) + A_e;

    % reset element matrix
    A_e(:) = 0.0;

end


function [N, dN, j] = ShapeFunction(x, gp)

xi  = gp(2);
eta = gp(3);

% shapefunctions Quad9
N = zeros(9,1); dN = zeros(9,2); J = zeros(2,2);

N(1) = 1/4*xi*eta*(1-xi)*(1-eta);
N(2) = -1/4*xi*eta*(1+xi)*(1-eta);
N(3) = 1/4*xi*eta*(1+xi)*(1+eta);
N(4) = -1/4*xi*eta*(1-xi)*(1+eta);
N(5) = -1/2*eta*(1-eta)*(1-xi^2);
N(6) = 1/2*xi*(1+xi)*(1-eta^2);
N(7) = 1/2*eta*(1+eta)*(1-xi^2);
N(8) = -1/2*xi*(1-xi)*(1-eta^2);
N(9) = (1-xi^2)*(1-eta^2);

dN(1,1)=1/4*eta*(1-xi)*(1-eta)-1/4*xi*eta*(1-eta);
dN(2,1)=-1/4*eta*(1+xi)*(1-eta)-1/4*xi*eta*(1-eta);
dN(3,1)=1/4*eta*(1+xi)*(1+eta)+1/4*xi*eta*(1+eta);
dN(4,1)=-1/4*eta*(1-xi)*(1+eta)+1/4*xi*eta*(1+eta);
dN(5,1)=xi*eta*(1-eta);
dN(6,1)=1/2*(1+xi)*(1-eta^2)+1/2*xi*(1-eta^2);
dN(7,1)= -xi*eta*(1+eta);
dN(8,1)=-1/2*(1-xi)*(1-eta^2)+1/2*xi*(1-eta^2);
dN(9,1)=-2*xi*(1-eta^2);

dN(1,2)=1/4*xi*(1-xi)*(1-eta)-1/4*xi*eta*(1-xi);
dN(2,2)=-1/4*xi*(1+xi)*(1-eta)+1/4*xi*eta*(1+xi);
dN(3,2)=1/4*xi*(1+xi)*(1+eta)+1/4*xi*eta*(1+xi);
dN(4,2)=-1/4*xi*(1-xi)*(1+eta)-1/4*xi*eta*(1-xi);
dN(5,2)=-1/2*(1-eta)*(1-xi^2)+1/2*eta*(1-xi^2);
dN(6,2)=-xi*eta*(1+xi);
dN(7,2)= 1/2*(1+eta)*(1-xi^2)+1/2*eta*(1-xi^2);
dN(8,2)=xi*eta*(1-xi);
dN(9,2)=-2*eta*(1-xi^2);

% jacobian matrix
J = x * dN;

% determinant of Jacobain matrix
j = det(J);

% transform normal system derivatives to carthesian system derivatives
dN = dN * inv(J);
```

```
end
```

D.2.2 Assemble matrix for membrane

```
function A = assembleMatrixMembrane(Mesh, ElementData,  ModelData)

N = Mesh.noNodes * ElementData.dof_m;
n = ElementData.nodes * ElementData.dof_m;

% allocate system matrix
A=zeros(N,N);

% allocate element matrix
A_e   = zeros(n);

% compute Gauss point locations and weights
gp = IntegrationScheme(ElementData.noInt);

% loop over all elements to assemble matrix
for element = Mesh.connect

    % set-up local node coordinates and connectivity
    connect_element = element(1:ElementData.nodes);
    x_element       = Mesh.x(:,connect_element);

    % loop over integration points
    for gauss_point = gp

        % compute shape functions
        [N, dN, j] = ShapeFunction(x_element, gauss_point);

        dX = gauss_point(1)*j;

        n = ElementData.nodes;
        m = ElementData.dof_m;

        Dmem = ModelData.Dmem;

        % form B matrix
        B = zeros(3, m*n);
        B(1, 1:m:end) = dN(:,1)';
        B(2, 2:m:end) = dN(:,2)';
        B(3, 1:m:end) = dN(:,2)';
        B(3, 2:m:end) = dN(:,1)';

        % compute element stiffness matrix
        A_int = B'*Dmem*B*dX;

        A_e = A_e + A_int;

    end

    % compute dof map
    dof_m = DofMap(ElementData.dof_m, connect_element, ElementData.nodes);
```

```
    % assemble
    A( dof_m, dof_m ) = A( dof_m, dof_m ) + A_e;

    % reset element matrix
    A_e(:) = 0.0;

end
```

D.2.3 Apply boundary conditions

```
function [K, F] = applyBoundaryConditions(K, F, BcData, ElementData)

for bc = BcData

    dof = ElementData*(bc(1)-1)+bc(3);
    K(dof,:)   = 0.0;
    K(dof,dof) = 1.0;
    F(dof) = bc(4);

end

end
```

D.2.4 Assemble extra strain vector

```
function f=computeA_star(Mesh,u_p)

for i=1:Mesh.noNodes
    f(i*3-2,1)=1/2*(u_p(i*3-1,1))^2;
    f(i*3-1,1)=1/2*u_p(i*3,1)^2;
    f(i*3,1)=u_p(i*3-1,1)*u_p(i*3,1);
end

end
```

D.2.5 Compute extra force vector

```
function f = computeF_t_star(Mesh, ElementData,  ModelData, A_star)

N = Mesh.noNodes * ElementData.dof_m;
n = ElementData.nodes * ElementData.dof_m;

% allocate system vector
f = zeros(N,1);

% allocate element vector
f_e    = zeros(n,1);

% compute Gauss point locations and weights
gp = IntegrationScheme(ElementData.noInt);

% loop over all elements to assemble vector
```

```
for element = Mesh.connect

    % set-up local node coordinates and connectivity
    connect_element = element(1:ElementData.nodes);
    x_element       = Mesh.x(:,connect_element);
    dof_p = DofMap(ElementData.dof_p, connect_element, ElementData.nodes);
    A_star_element = A_star(dof_p);

    % Loop over integration points
    for gauss_point = gp

        % compute shape functions
        [N, dN, j] = ShapeFunction(x_element, gauss_point);
        dX = gauss_point(1)*j;

        n = ElementData.nodes;
        m = ElementData.dof_m;
        k = ElementData.dof_p;

        Dmem = ModelData.Dmem;

        % form B matrix
        B = zeros(3, m*n);
        B(1, 1:m:end) = dN(:,1)';
        B(2, 2:m:end) = dN(:,2)';
        B(3, 1:m:end) = dN(:,2)';
        B(3, 2:m:end) = dN(:,1)';

        % form N_star matrix
        N_star=zeros(3,k*n);
        N_star(1, 1:k:end) = N(:);
        N_star(2, 2:k:end) = N(:);
        N_star(3, 3:k:end) = N(:);

        f_int = B'*Dmem*N_star*A_star_element*dX;

        f_e = f_e + f_int;

    end

    % compute dof map
    dof_m = DofMap(ElementData.dof_m, connect_element, ElementData.nodes);

    % assemble
    f( dof_m ) = f( dof_m ) + f_e;

    % reset element vector
    f_e = 0.0;

end
```

D.2.6 Compute membrane stresses

```
function f = computeMembraneStress(Mesh, ElementData,  u_m, f_t, ModelData, EpData)

N1 = Mesh.noNodes * ElementData.dof_p;
```

```
n = ElementData.nodes * ElementData.dof_m;
m = ElementData.dof_m;
k = ElementData.dof_p;

% allocate system vector
f=zeros(N1,1);

Dmem = ModelData.Dmem;

% compute Gauss point locations and weights
gp = IntegrationScheme(ElementData.noInt);
gp1=gp(:,1:4);
gp2=gp(:,5:8);
gp3=gp(:,9);

for element = Mesh.connect

    % set-up local node coordinates and connectivity
    connect_element = element(1:ElementData.nodes);
    x_element       = Mesh.x(:,connect_element);
    dof_m = DofMap(ElementData.dof_m, connect_element, ElementData.nodes);
    dof_p = DofMap(ElementData.dof_p, connect_element, ElementData.nodes);
    u_element = u_m( dof_m );
    f_t_element = f_t( dof_p);

    % loop over integration points to compute stress
    jj = 1;
    for gauss_point= gp1

        % compute shape functions
        [N, dN, j] = ShapeFunction(x_element, gauss_point);
        B = zeros(3, 27);
        B(1, 1:3:end) = N(:);
        B(2, 2:3:end) =N(:);
        B(3,3:3:end) = N(:);

        B1 = zeros(3, 18);
        B1(1, 1:m:end) = dN(:,1)';
        B1(2, 2:m:end) = dN(:,2)';
        B1(3,1:m:end) = dN(:,2)';
        B1(3,2:m:end) = dN(:,1)';

        f_int1(jj*3-2:jj*3,1) =Dmem*(B1*u_element+B*f_t_element)*1/4;
        jj=jj+1;
    end
    jj=1;
    for gauss_point= gp2

        % compute shape functions
        [N, dN, j] = ShapeFunction(x_element, gauss_point);
        B = zeros(3, 27);
        B(1, 1:3:end) = N(:);
        B(2, 2:3:end) =N(:);
        B(3,3:3:end) = N(:);

        B2 = zeros(3, 18);
        B2(1, 1:m:end) = dN(:,1)';
```

```
        B2(2, 2:m:end) = dN(:,2)';
        B2(3,1:m:end) = dN(:,2)';
        B2(3,2:m:end) = dN(:,1)';

        f_int2(jj*3-2:jj*3,1) =Dmem*(B2*u_element+B*f_t_element)*1/2;
        jj=jj+1;
    end
    jj=1;
    for gauss_point= gp3

        % compute shape functions
        [N, dN, j] = ShapeFunction(x_element, gauss_point);
        B = zeros(3, 27);
        B(1, 1:3:end) = N(:);
        B(2, 2:3:end) =N(:);
        B(3,3:3:end) = N(:);

        B3 = zeros(3, 18);
        B3(1, 1:m:end) = dN(:,1)';
        B3(2, 2:m:end) = dN(:,2)';
        B3(3,1:m:end) = dN(:,2)';
        B3(3,2:m:end) = dN(:,1)';

        f_int3(jj*3-2:jj*3,1) = Dmem*(B3*u_element+B*f_t_element);
        jj=jj+1;
    end

    dof_p = DofMap(ElementData.dof_p, connect_element, ElementData.nodes);

    f(dof_p(1:12))=f(dof_p(1:12))+f_int1;
    f(dof_p(13:24))=f(dof_p(13:24))+f_int2;
    f(dof_p(25:27))=f(dof_p(25:27))+f_int3;
end

% extrapolate
f(EpData.ring1*3-2)=f(EpData.ring2*3-2)+(f(EpData.ring2*3-2)-f(EpData.ring3*3-2));
f(EpData.ring1*3-1)=f(EpData.ring2*3-1)+(f(EpData.ring2*3-1)-f(EpData.ring3*3-1));
f(EpData.ring1*3)=f(EpData.ring2*3)+(f(EpData.ring2*3)-f(EpData.ring3*3));

f=f/ModelData.t;
end
```

D.2.7 Compute curvature

```
function f = computeCurvature(Mesh, ElementData, EpData, a_p)

N = Mesh.noNodes * ElementData.dof_p;
n = ElementData.nodes;
m = ElementData.dof_p;

% allocate system vector
f=zeros(N,1);

% compute Gauss point locations and weights
gp = IntegrationScheme(ElementData.noInt);
gp1=gp(:,1:4);
```

```
gp2=gp(:,5:8);
gp3=gp(:,9);

for element = Mesh.connect

    % set-up local node coordinates and connectivity
    connect_element = element(1:ElementData.nodes);
    x_element       = Mesh.x(:,connect_element);
    dof_p = DofMap(ElementData.dof_p, connect_element, ElementData.nodes);
    a_p_element = a_p( dof_p);

    % loop over integration points to compute curvature
    jj = 1;
    for gauss_point= gp1

        % compute shape functions
        [N, dN, j] = ShapeFunction(x_element, gauss_point);

        % form B matrix
        B = zeros(3, m*n);
        B(1, 2:m:end) = dN(:,1)';
        B(2, 3:m:end) = dN(:,2)';
        B(3, 2:m:end) = dN(:,2)';
        B(3, 3:m:end) = dN(:,1)';

        k_int1(jj*3-2:jj*3,1) = B*a_p_element*1/4;
        jj=jj+1;
    end
    jj=1;
    for gauss_point= gp2

        % compute shape functions
        [N, dN, j] = ShapeFunction(x_element, gauss_point);

        % form B matrix
        B = zeros(3, m*n);
        B(1, 2:m:end) = dN(:,1)';
        B(2, 3:m:end) = dN(:,2)';
        B(3, 2:m:end) = dN(:,2)';
        B(3, 3:m:end) = dN(:,1)';

        k_int2(jj*3-2:jj*3,1) = B*a_p_element*1/2;
        jj=jj+1;
    end
    jj=1;
    for gauss_point= gp3

        % compute shape functions
        [N, dN, j] = ShapeFunction(x_element, gauss_point);

        % form B matrix
        B = zeros(3, m*n);
        B(1, 2:m:end) = dN(:,1)';
        B(2, 3:m:end) = dN(:,2)';
        B(3, 2:m:end) = dN(:,2)';
        B(3, 3:m:end) = dN(:,1)';
```

```
        k_int3(jj*3-2:jj*3,1) = B*a_p_element;
        jj=jj+1;
    end

    % assemble
    f(dof_p(1:12))=f(dof_p(1:12))+k_int1;
    f(dof_p(13:24))=f(dof_p(13:24))+k_int2;
    f(dof_p(25:27))=f(dof_p(25:27))+k_int3;
end

% extrapolate
f(EpData.ring1*3-2)=f(EpData.ring2*3-2)+(f(EpData.ring2*3-2)-f(EpData.ring3*3-2));
f(EpData.ring1*3-1)=f(EpData.ring2*3-1)+(f(EpData.ring2*3-1)-f(EpData.ring3*3-1));
f(EpData.ring1*3)  =f(EpData.ring2*3)+(f(EpData.ring2*3)-f(EpData.ring3*3));

f(EpData.corner*3-2)=2*f(EpData.corner*3-2);
f(EpData.corner*3-1)=2*f(EpData.corner*3-1);
f(EpData.corner*3)  =2*f(EpData.corner*3);

end
```

D.2.8 Compute membrane forces in z direction

```
function f = computeF_t_star(Mesh, ElementData,  ModelData, A_star)

N = Mesh.noNodes * ElementData.dof_m;
n = ElementData.nodes * ElementData.dof_m;

% allocate system vector
f = zeros(N,1);

% allocate element vector
f_e    = zeros(n,1);

% compute Gauss point locations and weights
gp = IntegrationScheme(ElementData.noInt);

% loop over all elements to assemble vector
for element = Mesh.connect

    % set-up local node coordinates and connectivity
    connect_element = element(1:ElementData.nodes);
    x_element       = Mesh.x(:,connect_element);
    dof_p = DofMap(ElementData.dof_p, connect_element, ElementData.nodes);
    A_star_element = A_star(dof_p);

    % loop over integration points
    for gauss_point = gp

        % compute shape functions
        [N, dN, j] = ShapeFunction(x_element, gauss_point);
        dX = gauss_point(1)*j;

        n = ElementData.nodes;
        m = ElementData.dof_m;
        k = ElementData.dof_p;
```

```
        Dmem = ModelData.Dmem;

        % form B matrix
        B = zeros(3, m*n);
        B(1, 1:m:end) = dN(:,1)';
        B(2, 2:m:end) = dN(:,2)';
        B(3, 1:m:end) = dN(:,2)';
        B(3, 2:m:end) = dN(:,1)';

        % form N_star matrix
        N_star=zeros(3,k*n);
        N_star(1, 1:k:end) = N(:);
        N_star(2, 2:k:end) = N(:);
        N_star(3, 3:k:end) = N(:);

        f_int = B'*Dmem*N_star*A_star_element*dX;

        f_e = f_e + f_int;

    end

    % compute dof map
    dof_m = DofMap(ElementData.dof_m, connect_element, ElementData.nodes);

    % assemble
    f( dof_m ) = f( dof_m ) + f_e;

    % reset element vector
    f_e = 0.0;

end
```

D.3 Postprocessor

D.3.1 Compute reaction forces

```
function Reactions = computeReactions(Mesh, K_p, BcData, u_p)

for i=1:BcData.noBc
    Reactions.value(i)=K_p(BcData.plate(1,i)'*3-2,:)*u_p;
    Reactions.x(i,:)=Mesh.x(:,BcData.plate(1,i)');
end
end
```

D.3.2 Compute bending stresses

```
function Sigma_b = computeBendingStress(Mesh, ModelData, Kappa)

for i=1:Mesh.noNodes
    Sigma_b(3*i-2:3*i,1)=ModelData.Dmem*-0.5*Kappa(3*i-2:3*i,1);
end
end
```

D.3.3 Compute total stresses

```
function Sigma_t = computeTotalStress(Sigma_m, Sigma_b)

Sigma_t.SxxL1=Sigma_b(1:3:end)+Sigma_m(1:3:end);
Sigma_t.SyyL1=Sigma_b(2:3:end)+Sigma_m(2:3:end);
Sigma_t.SxyL1=Sigma_b(3:3:end)+Sigma_m(3:3:end);

Sigma_t.SxxL2=Sigma_m(1:3:end);
Sigma_t.SyyL2=Sigma_m(2:3:end);
Sigma_t.SxyL2=Sigma_m(3:3:end);

Sigma_t.SxxL3=-Sigma_b(1:3:end)+Sigma_m(1:3:end);
Sigma_t.SyyL3=-Sigma_b(2:3:end)+Sigma_m(2:3:end);
Sigma_t.SxyL3=-Sigma_b(3:3:end)+Sigma_m(3:3:end);

end
```

D.3.4 Compute principal stresses

```
function Sigma_p = computePrincipalStress(Mesh, Sigma_m, Sigma_b)

for i=1:Mesh.noNodes
    Sigma_p.P1L1(i,1)=(Sigma_m(i*3-2)+Sigma_b(i*3-2)+Sigma_m(i*3-1)+Sigma_b(i*3-1))/2 +
    sqrt(((Sigma_m(i*3-2)+Sigma_b(i*3-2)-Sigma_m(i*3-1)+Sigma_b(i*3-1))/2)^2 +
    (Sigma_m(i*3)+Sigma_b(i*3))^2);

    Sigma_p.P2L1(i,1)=(Sigma_m(i*3-2)+Sigma_b(i*3-2)+Sigma_m(i*3-1)+Sigma_b(i*3-1))/2 -
    sqrt(((Sigma_m(i*3-2)+Sigma_b(i*3-2)-Sigma_m(i*3-1)+Sigma_b(i*3-1))/2)^2+
    (Sigma_m(i*3)+Sigma_b(i*3))^2   );

    Sigma_p.P1L2(i,1)=(Sigma_m(i*3-2)+Sigma_m(i*3-1))/2 +
    sqrt(((Sigma_m(i*3-2)-Sigma_m(i*3-1))/2)^2 + Sigma_m(i*3)^2);

    Sigma_p.P2L2(i,1)=(Sigma_m(i*3-2)+Sigma_m(i*3-1))/2 -
    sqrt(((Sigma_m(i*3-2)-Sigma_m(i*3-1))/2)^2 + Sigma_m(i*3)^2);

    Sigma_p.P1L3(i,1)=(Sigma_m(i*3-2)-Sigma_b(i*3-2)+Sigma_m(i*3-1)-Sigma_b(i*3-1))/2 +
    sqrt(((Sigma_m(i*3-2)-Sigma_b(i*3-2)-Sigma_m(i*3-1)-Sigma_b(i*3-1))/2)^2 +
    (Sigma_m(i*3)-Sigma_b(i*3))^2);

    Sigma_p.P2L3(i,1)=(Sigma_m(i*3-2)-Sigma_b(i*3-2)+Sigma_m(i*3-1)-Sigma_b(i*3-1))/2 -
    sqrt(((Sigma_m(i*3-2)-Sigma_b(i*3-2)-Sigma_m(i*3-1)-Sigma_b(i*3-1))/2)^2+
    (Sigma_m(i*3)-Sigma_b(i*3))^2);
end
end
```