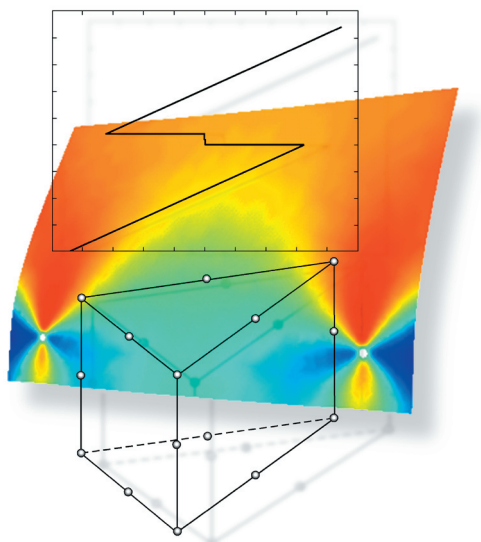
LUND UNIVERSITY



A FINITE ELEMENT BASED DESIGN TOOL FOR POINT FIXED LAMINATED GLASS

JENS MALMBORG

Structural Mechanics

Master´s Dissertation

Department of Construction Sciences
Structural Mechanics

ISRN LUTVDG/TVSM--06/5144--SE (1-79)
ISSN 0281-6679

A FINITE ELEMENT BASED DESIGN TOOL FOR POINT FIXED LAMINATED GLASS

Master's Dissertation by

JENS MALMBORG

Supervisors:

Kent Persson,PhD, and Jonas Lindemann, PhD.,
Div. of Structural Mechanics

Printed by KFS I Lund AB, Lund, Sweden, June, 2006.

For information, address:
Division of Structural Mechanics, LTH, Lund University, Box 118, SE-221 00 Lund, Sweden.
Homepage: http://www.byggmek.lth.se

Abstract

In this master's project, a design tool for bolt fixed laminated glass was developed, based on linear-elastic finite element theory. The program produces images of the deflections and stress distributions in bolted glass panes. A two-dimensional triangular mesh, created by an external mesh generator, was extended to a three-dimensional mesh comprising 15-node wedge elements. A least-square method was adopted for extrapolating stresses from Gaussian sample points to the element nodes. A simple graphical user interface was developed in the programming language Python, as well as a simple post-processor employing the graphics library OpenGL.

Keywords: FEM, finite element analysis, bolt fixed glass, laminated glass, toughened glass, Python, Fortran, wedge element, 15-node wedge element, stress extrapolation, discontinuous stress distribution, sparse matrix

Summary

The work presented in this master's thesis concerns the development of a finite element based design tool for balustrades and facades of laminated, toughened, bolt-fixed glass.

The objective of this thesis was to implement a finite element solver using composite material shell elements in order to determine stresses in rectangular, bolt-fixed glass panes, subjected to a distributed load or a line load. Merely one free mesh generator was found, yielding two-dimensional triangular elements. There are several commercial mesh generators, but they are rather expensive and did not fit the budget of this project. An effort was made to model the behavior of a glass structure subjected to bending by means of triangular composite material shell elements. The shell elements failed to represent the discontinuous stress distribution that arises in the thickness direction of a laminated glass pane under certain loads and boundary conditions. A routine was written for extending an arbitrary two-dimensional triangular mesh to a multi-layered three-dimensional mesh comprising 15-node wedge elements. The 15-node wedge elements proved to be as capable as 20-node hexagonal elements in representing both deflections and stress distributions, but with a slower convergence.

Two different sparse solvers were tested for solving the global system of equations; a direct solver from the *Intel Math Kernel Library*, and an iterative solver from *SPARSKIT*. The iterative solver converged rather quickly when using shell elements, but did not converge at all when using the 15-node wedge elements. This might be due to the higher nodal connectivity, resulting in a less sparse system matrix. A nodal-reordering scheme, the nested dissection method, was used for speeding up the convergence of the iterative solver. This improved the convergence speed significantly for the shell elements, but had no effect at all on the wedge elements. The direct solver employs the very same nodal-reordering scheme as the one tested with the iterative solver. The direct solver was not tested with the shell elements, but proved to be very stable with the wedge elements that were finally implemented in the design tool.

In order to obtain the stresses at the structure boundaries, a least-square extrapolation method with local stress smoothing was adopted.

A simple post-processor, showing structural displacements and nodal stresses in different sections of the analyzed glass pane, was developed in Python and OpenGL.

Preface

The work presented in this master's thesis was carried out at the Division of Structural Mechanics at Lund University, Sweden, during the period November 2005 - May 2006.

I would first and foremost like to express my deepest gratitude to my supervisor Dr. Kent Persson for invaluable guidance and support throughout the course of this work. Not once was I asked to come back at a later time when I was in need of advice. I am also very grateful to my supervisor Dr. Jonas Lindemann, whose invaluable programming and technical skills I could not have been without. Help from Mr. Bo Zadig with graphics and printing is also gratefully acknowledged.

Lund, May 2006

Jens Malmborg

Contents

Chapter 1

Introduction

1.1 Background

During the last few years architects have become more interested in using glass in supporting parts of building structures, and in these structures minimizing the amount of other materials. The fixings of glass structures raise interesting problems, since this is where other types of materials usually need to be employed. Glass balustrades attached to railings, and glass facades attached to the building structure, are two examples where the glass may be fixed through cylindrical or countersunk holes that are cut out during the manufacturing process. Through these holes the glass may be connected onto the supporting structure by metal bolts, see Figure 1.1, where the glass needs to carry itself and due loads.

Laminated glass usually consists of two layers of very brittle hardened glass with a thin intermediate foil of PVB. PVB is a highly elastic material that keeps the glass in place in the event of failure, thereby preventing people from getting hurt.

Figure 1.1: *Bolts for cylindrical holes and countersunk holes, respectively. [1]*

1.2 Objective

In the building industry, there is little knowledge about how do design glass structures in a safe and efficient way. The objective of this master's thesis is to develop a design tool where different parameters concerning laminated glass panes, fixed with bolts, can be determined. These parameters include distance between bolts and edges, glass thickness, etc. Material relationships established in an earlier master's thesis, [1], as well as some building codes, will be implemented. The tool will be based on the finite element method and should have a user-friendly, intuitive graphical interface, that enables persons unacquainted to the finite element method to use the tool.

1.3 Project Tasks

The major task is to implement a finite element solver that uses shell elements with composite material formulation. An external mesh generator for triangular or quadrilateral elements will be implemented. A graphical interface capable of handling different geometries, support and load types is to be connected to the computational code. The geometries are confined to rectangular shapes.

Relations between calculated stresses/strains and design parameters should be implemented. The results will be verified by comparison to ABAQUS, in order to assure that the computational routines have been correctly implemented. To the extent time allows, the results will also be compared to experimental results obtained in an earlier master's thesis where the bolts in Figure 1.1 were tested.

1.4 Extended Project Tasks

It was shown that the shell elements fail to represent the characteristic stress distribution that may arise in a laminated glass pane subjected to bending. Therefore, the project is extended to include the implementation of a three dimensional solid element that may represent this stress distribution. Only one free mesh generator has been found, which yields two-dimensional triangular elements. Therefore, the project also involves creating an algorithm that extends a two-dimensional triangular mesh to a three-dimensional mesh comprising wedge elements.

1.5 Limitations

The geometries are confined to rectangular shapes. Two different loading cases are considered; a line load acting along the top edge of a balustrade, and a distributed load acting on a facade, see Figure 1.2. The program developed within this project will be completely based on linear elastic theory. There will be no consideration of

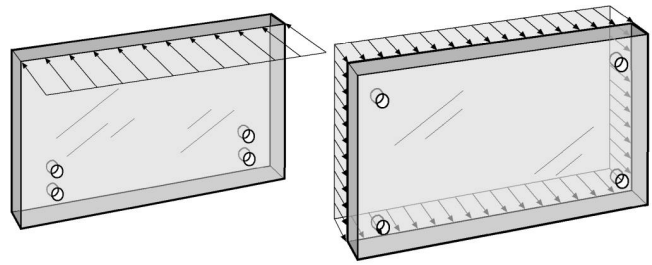


Figure 1.2: *A balustrade subjected to a line load, and a facade subjected to a distributed load.*

nonlinear effects, neither geometrical nor material. The actual bolt fixings will be excluded from the model, meaning that an attempt will be made to resemble the behavior of the glass structure by means of proper boundary conditions around the holes.

1.6 Intended Audience

In order to grasp the work presented in this master's thesis, good knowledge in finite element analysis is required. It is also assumed that the reader has some experience in programming, especially applied to finite element analysis, so that the basic steps in solving a finite element problem are clear. The text presented does not contain any program code.

1.7 Report Outline

In Chapter 2, an overview of the mechanical properties of laminated glass is given.

Chapter 3 explains what kind of tools the modelling of laminated glass requires, or more specifically, what types of finite elements are appropriate for implementation in a glass design program.

Chapter 4 is the main chapter of this report. It deals with the implementation of the glass design program, ClearSight, created in this project. A more detailed disposition of Chapter 4 is given in Section 4.1.

A few examples of structures analyzed with the program, are shown in Chapter 5.

Chapter 6 contains some general concluding remarks and a few proposals for future work.

Chapter 2

Mechanical Properties of Laminated Glass

2.1 Toughened Glass

Glass is an isotropic extremely brittle material. Under normal temperatures, it displays a linear elastic behavior with almost no plastic strains. According to the theory of Fracture Mechanics, plastic strains have to develop to some extent, though, since an infinitesimal load increment leads to infinite stress concentrations around a crack tip. Considering glass in normal temperatures, the development of these plastic strains is normally negligible, and the stress-strain curve is almost perfectly linear up to failure. Hence, it is treated as an ideal elastic-plastic material.

Toughened glass is made from annealed float glass that undergoes a thermal tempering process, in which the the annealed glass is heated to its softening point ($\sim 650°$). The glass is then rapidly cooled by air jets, which causes the glass surfaces to contract whereas the inner region continues to float a while longer. When the inner region finally contracts, the surfaces are subjected to compressive stresses that are balanced by tensile stresses in the inner region. [1]. A principal figure of the residual stress variation through the glass thickness is shown in Figure 2.1.

Glass is especially sensitive to tension, and it is in most real cases tensile stresses that ultimately causes glass to fail. The toughened glass can resist larger bending induced deformations than ordinary glass, when the residual compressive stresses near the glass surface allow for larger bending induced tensile stresses. When toughened glass fails, it shatters in small harmless pieces.

2.2 Laminated Glass

Laminated glass usually consists of two layers of toughened glass, with a thin intermediate foil of PVB.

PVB is a highly elastic material, allowing for very large deformations. It behaves non-linearly when subjected to large deformations, but since it is only subjected to very small deformations in the applications considered in this project, it is modelled as a linear elastic material. The creep properties, however, may be important for the long-term mechanical behavior, but they are neglected in this study.

When the toughened glass shatters, the pieces will stick to the PVB foil. This is the sole purpose of the PVB foil.

Both glass and PVB are regarded as linear-elastic materials. The following stiffness parameters were employed throughout this study

$$\begin{aligned} E_{glass} &= 78\ GPa \\ v_{glass} &= 0.23 \\ \\ E_{pvb} &= 9\ MPa \\ v_{pvb} &= 0.43 \end{aligned} \tag{2.1}$$

2.3 Fracture Criterion

A proposal to the new Eurocode on the design of glass structures, [2], suggests that a permissible tensile strength $f_{gk} = 50\ MPa$ be used for thermally toughened safety glass.

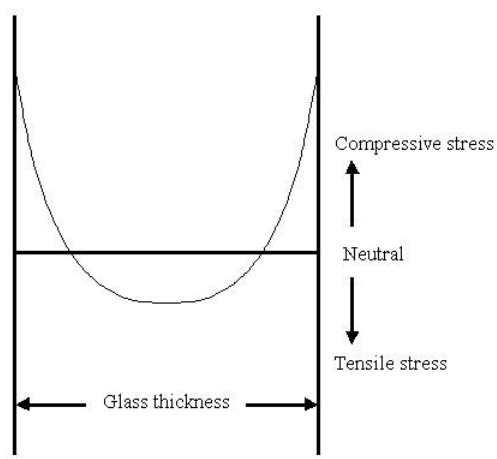


Figure 2.1: *The residual stress profile in a toughened glass pane. (Reproduced from [1])*

A principal stress is a normal stress acting on a plane where the shear stresses are zero. For a certain point, the maximum positive principal stress is therefore the maximum tensile stress acting in that point. Too high a tensile stress is what causes glass to fail, and hence the maximum positive principal stresses are used to investigate whether the permissible stress is exceeded in any point of the analyzed structure.

The residual stresses in the toughened glass, shown in Figure 2.1, are not considered in the stress calculations. It is merely their beneficial effect on the permissible stress that is taken into account.

Chapter 3

Analyzing Laminated Toughened Glass

3.1 General

From a previous master's thesis [1], laminated glass is known to display very complex stress distributions in the thickness direction, when subjected to certain loads and boundary conditions. This complex stress distribution is due to the large differences in stiffness for the glass versus the PVB. In order to calculate the stresses that arise in different parts of a laminated glass structure, where it is impossible to use analytic formulas, it is necessary to resort to numerical methods such as the Finite Element Method. Different element types are appropriate for different types of problems, and even though it is always possible to find a solution, the obtained solution is not always in the vicinity of the true solution.

Since merely one free, reliable mesh generator has been found − Triangle, yielding triangular elements only − the scope of element types appropriate for implementation in this project is essentially narrowed. Three types of elements, all based on triangle shapes, were tested.

3.2 The Composite Material Shell Element

3.2.1 General

Many different shell finite elements have been proposed during the last few decades. These kinds of elements are usually employed when shell-like structures are to be analyzed. The behavior of a shell structure depends on the geometry as well as the boundary conditions, placing the structure in a category of membrane-dominated action, bending-dominated action or mixed action. An isotropic triangular shell element as proposed by Phill-Seung Lee and Klaus-Jürgen Bathe [3] was tested in MATLAB. The test results were compared to an identical structure analyzed with

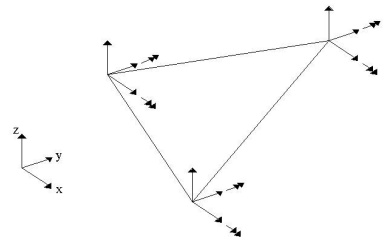


Figure 3.1: *A shell element with 15 degrees of freedom.*

solid elements in ABAQUS. The shell element tested has three nodes, with five degrees of freedom at each node; three translational and two rotational. See Figure 3.1.

Every element is given a certain number of integration layers, typically one integration layer per material layer. Thickness and material parameters are set for each integration layer. Every integration layer has two integration points located at the center of each triangle, equally distanced from the top and bottom of the layer. Using a larger number of integration layers results in a larger number of integration points, presumably giving a more accurate result of the stress and strain distribution in the thickness direction.

3.2.2 Testing the shell elements

To investigate whether the shell elements are able to represent the stress distribution that arises in a laminated glass structure, a simple beam test was performed. Figure 3.2 shows a cantilever beam of length L = 5 m, composed of 800 structured triangles. The beam consists of three material layers; two glass layers, each of thickness $t_g = 8mm$, and one intermediate PVB layer of thickness $t_{pvb} = 0.76mm$. The material parameters from Equation 2.1 are used. One end of the beam is completely locked in all directions, i.e no translation or rotation is allowed, whereas the free end is subjected to two point loads P = 50 N.

Figure 3.3 shows the stress distribution in the direction of the beam, in a section located 2.5 meters away from the load, as obtained with the shell elements. The normal stress varies almost linearly through the whole thickness, the only discrepancy occurring in the soft PVB foil where the normal stress is close to zero throughout the foil.

Figure 3.4 shows the same stress distribution as given by an analysis with quadratic hexagonal solid elements in ABAQUS. This element type proved to be accurate when used for simulating a few experimental set-ups in an earlier master's thesis [1]. Every material layer has two finite elements in the thickness direction. In this simple

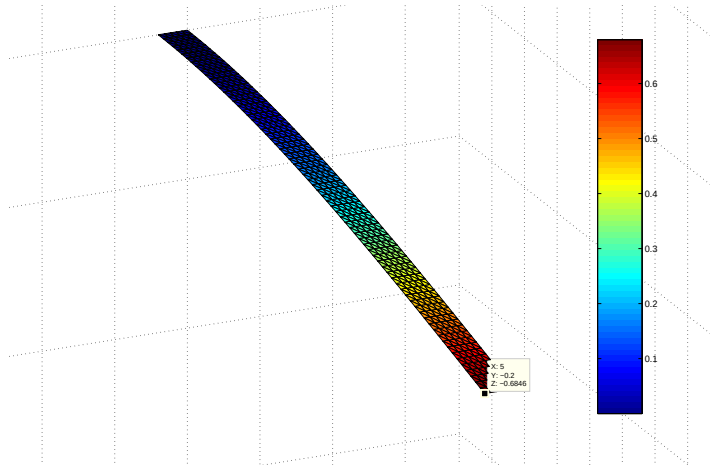


Figure 3.2: *A beam composed of 800 shell elements.*

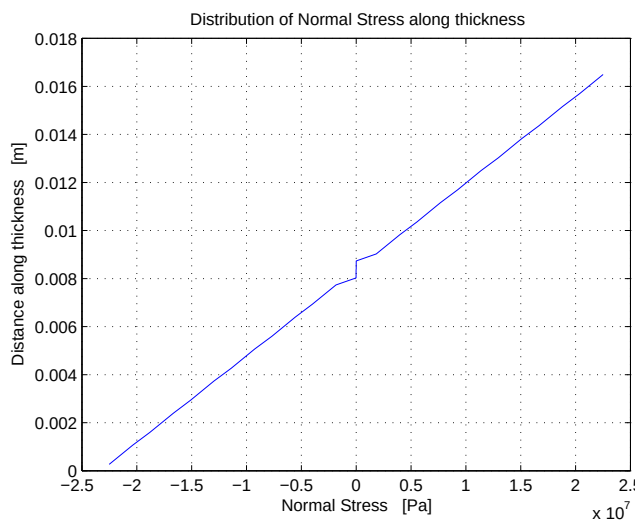


Figure 3.3: *Stress distribution in the thickness direction of the shell beam.*

model, the shell elements yield about the same result as the hexagonal elements. Although this first, simple check seems promising, it is necessary to investigate further whether the shell element can represent other, more complex stress distributions. As mentioned, laminated glass is known to show much more complex stress distributions when subjected to certain loads and boundary conditions. Such a stress distribution may for instance occur around holes in a balustrade subjected to bending. Without digging too deep into the theory of these specific shell elements, their capability to represent more complex stress distributions can easily be tested

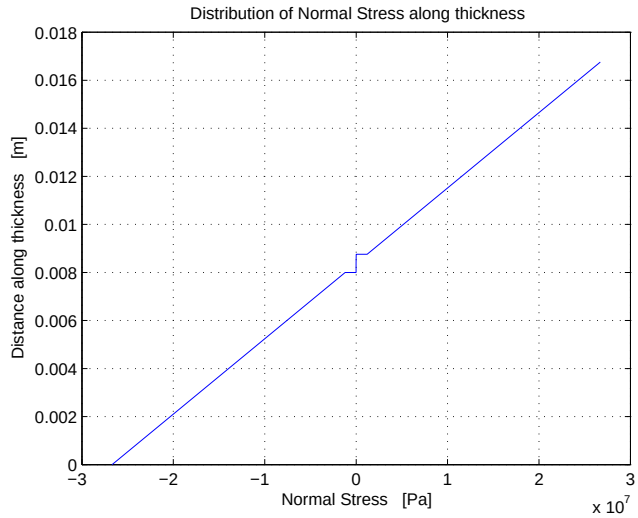


Figure 3.4: *Stress distribution in the thickness direction of the beam composed of quadratic hexagonal solid elements.*

by using the same beam model with a lower Young's modulus for the intermediate PVB foil.

Figure 3.5 shows the stress distribution in the same section of the beam, obtained with the same ABAQUS-model as above, where Young's modulus of the intermediate foil is given a value of $E = 9kPa$.

There appears to be almost no transmission of shear stresses between the different material layers. This lack of interaction between the layers is due to the very large differences in stiffness. It is emphasized that this kind of discontinuous stress distribution is known to arise in some situations when the intermediate foil is given its true value, and it is therefore utterly important that the element type implemented in a glass design program is capable of representing such a stress distribution.

Figure 3.6 shows the result obtained with the shell elements, where again the normal stress varies almost linearly through the whole thickness.
It is concluded that, due to the incapability to resemble a discontinuous stress distribution as shown in Figure 3.5, the shell element is not a proper choice for implementation in this project.

Another reason for not using shell elements in this project is that they are two-dimensional, thus making realistic modelling of the fixings hard.

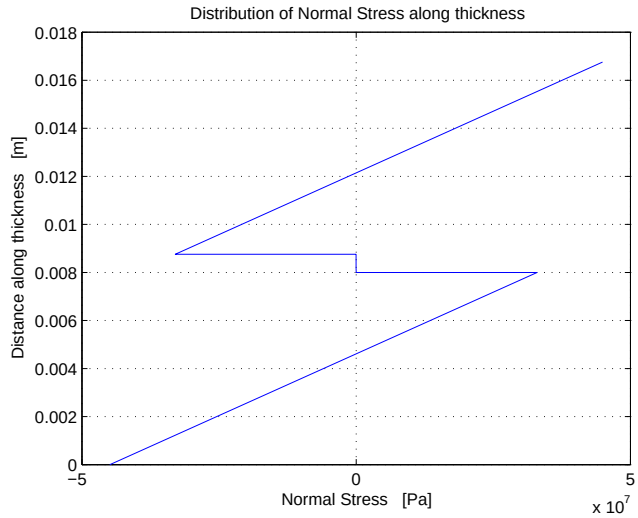


Figure 3.5: *Stress distribution in the thickness direction of the beam composed of hexagonal solid elements, where Young's modulus of the intermediate foil is $E = 9$ kPa.*

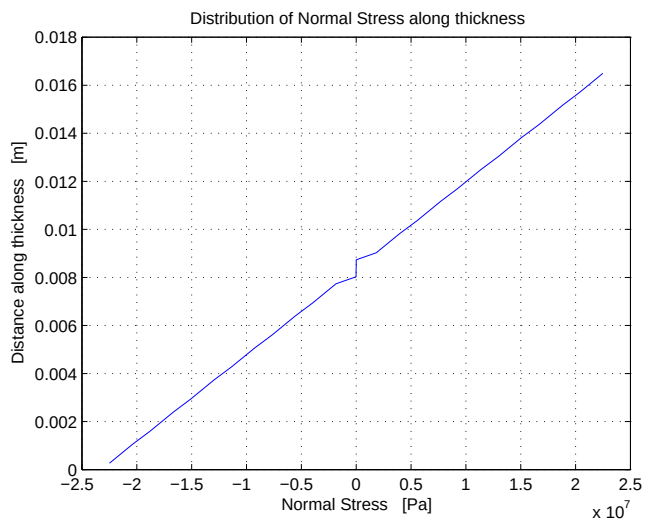


Figure 3.6: *Stress distribution in the thickness direction of the beam composed of shell elements, where Young's modulus of the intermediate foil is $E = 9$ kPa.*

3.3 Solid Continuum Elements

3.3.1 The 6-node wedge element

A six node wedge element was also tested. This element type, shown in Figure 3.7, has 3 translational degrees of freedom at each node.

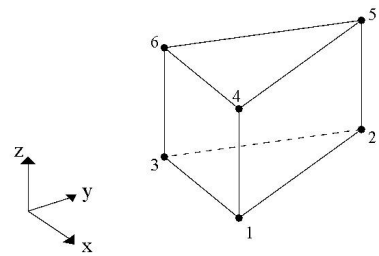


Figure 3.7: *The 6-node wedge element.*

The element was implemented and tested in MATLAB, using the same beam model used for testing the shell element. The results showed that this element type is extremely stiff and reluctant to bend, even when a very large number of elements is used. The ABAQUS manual [4] also discourages users from employing the 6-node wedge element in structures subjected to bending.

3.3.2 The 15-node wedge element

The 15-node wedge element, shown in Figure 3.8, is an isoparametric solid continuum element. It has three translational degrees of freedom in each node.

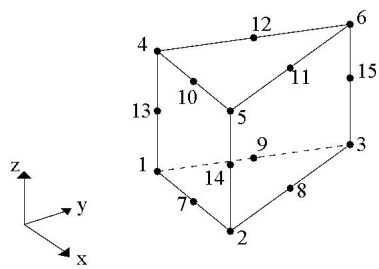


Figure 3.8: *The 15-node wedge element.*

The beam test used for evaluating the shell element, as well as the 6-node wedge element, was also used with the 15-node wedge elements. Every material layer has two finite elements in the thickness direction. The stress distribution is shown in

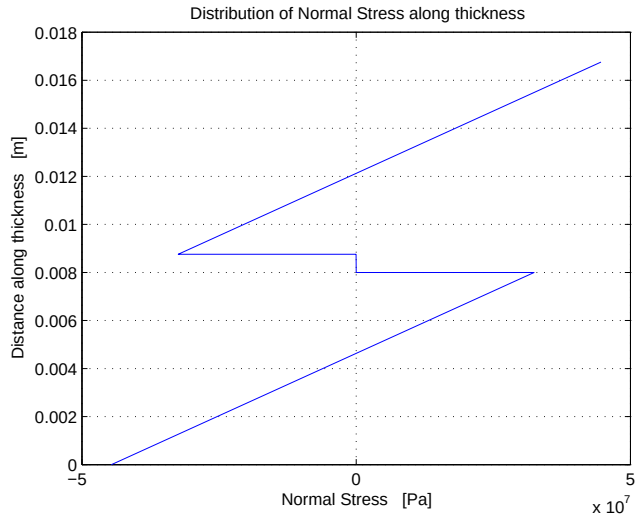


Figure 3.9: *Stress distribution in the thickness direction of the beam composed of 15-node wedge elements, where Young's modulus of the intermediate foil is E = 9 kPa.*

Figure 3.9. This stress distribution is almost identical to the one obtained with the hexagonal elements, see Figure 3.5.

3.4 Conclusions

Due to the large differences in Young's moduli of glass and PVB, the stress distribution in a laminated glass structure subjected to bending becomes highly discontinuous in certain areas. The shell elements fail to represent such a discontinuous stress distribution. The 6-node wedge element proved to be very stiff and reluctant to bend. The stress distribution was not therefore calculated.

The results show that the 15-node wedge element is obviously much more capable of representing bending than the 6-node wedge element. It can also represent discontinuous stress distributions when several element layers are used. Of the three element types tested, the 15-node wedge element is superior for applications involving structures subjected to bending, when the structure is composed of different materials whose Young's moduli differ significantly.

Further tests, not shown here, indicate that the 15-node wedge element converges to the same solution as the hexagonal element, but a greater number of elements

is required. The hexagonal element would therefore be preferable, but since no free mesh generator was found for this type of element, there are really no options. This 15-node wedge element was the one finally implemented in the glass design program, and it is therefore more thoroughly described in Chapter 4.

Chapter 4

Development and Implementation of a Glass Design Program

4.1 Chapter Outline

This chapter aims at explaining some of the theoretical and practical aspects of the development of a glass design program. Every major topic has been given an own section.

In Section 4.2 the basic program structure is explained. The major steps in the computational process are presented.

In order to allow for persons unacquainted to finite elements to use the program, a simple graphical user interface was developed in the programming language Python. In Section 4.3 the details about the implementation are explained.

Section 4.4 deals with the generation of finite element meshes. In this project an external mesh generator, Triangle [15], is used to produce a two dimensional mesh composed of triangular elements. This 2D mesh is then extended to a 3D mesh comprising wedge elements.

A 15-node wedge element was implemented in this project. It is discussed in detail in Section 4.5.

In Section 4.6 the evaluation of stresses and strains is discussed. A method for extrapolating the stresses to the nodes was implemented, and some of the theory behind that method is presented.

Section 4.7 explains how the program deals with bolt fixings. The assumptions and simplifications are presented and discussed.

Section 4.8 explains how the global system of equations that arise in finite element

analysis may be solved. First, two different formats for storing large matrices where most of the indices are zeroes, so called *sparse* matrices, are presented. Large finite element models with many degrees of freedom, result in very large matrices that are always more or less sparse. In order to store these matrices in an efficient way, advantage is taken of the fact that zeroes do not need to be stored. Two different families of solvers, direct and iterative, are briefly introduced. Focus has been put on one type of a direct solver that performs a Cholesky factorization of the stiffness matrix. The important concept of *fill-ins* as well as its implications on computer memory consumption and solve time are explained.

Section 4.9 deals with the visualization of the results from the finite element calculations. The graphics library OpenGL was used in order to create nice and illustrative images of stress distributions in a simple post-processor.

4.2 Program Structure

The design tool created in this project is intended for users with little or no experience in finite element analysis, and hence it has been a major objective to make it as user-friendly and simplistic as possible. This means that the user should not really have to know anything about how the solution is obtained, but merely how to give the program proper input.

In order to minimize the steps where the user may go astray, the part of the program where inputs are entered has been given a very simple Graphical User Interface *(GUI)* where detailed instructions are given for every input. The glass design program is actually three separate programs that are controlled by one of them. The GUI was developed in the programming language Python, with the extension module wxPython. This is further presented in section 4.3.

When the user has given the proper inputs and decides to run the analysis, essentially three things are carried out from the Python code.

- A material file that contains information about the number of material layers in the structure, the number of finite element layers and material properties of each layer, is written. A load file that contains information about what loads are imposed on the structure, is also written.
- A mesh generator is executed, creating a finite element mesh of the structure defined by the user. The finite element mesh is stored in a number of files, which are read by a routine in the GUI. The routine creates a file (a so called area file, see further section 4.4) that tells the mesh generator where the mesh needs to be refined. The mesh generator is then executed again, and new mesh files are created.

- The computational code is executed, with the mesh-, material-, and load-files given as input.

All computational code was written in Fortran95, which is a language traditionally used for finite element computations. When the structural displacements and stresses have been calculated, they are written to text files. These text files are then read by the post-processor that is called from the GUI when the computations have been carried out. The post-processor is written in Python with the graphics library OpenGL.

Figure 4.1 shows the principal structure of the design program. Figure 4.2 shows in principle how the computations are carried out.

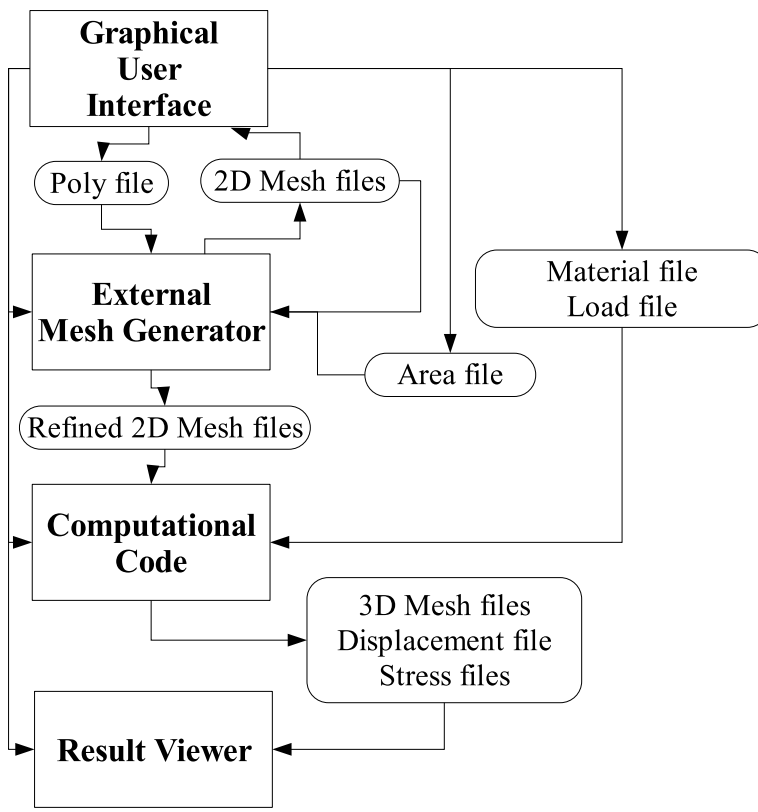


Figure 4.1: *Principal structure of ClearSight*

The Fortran code uses some external routines. For inverting matrices, determining eigenvalues and solving a linear system of equations, routines from the *Intel Math Kernel Library* [11] are used, whereas routines for handling sparse matrices are taken from *SPARSKIT* [9].

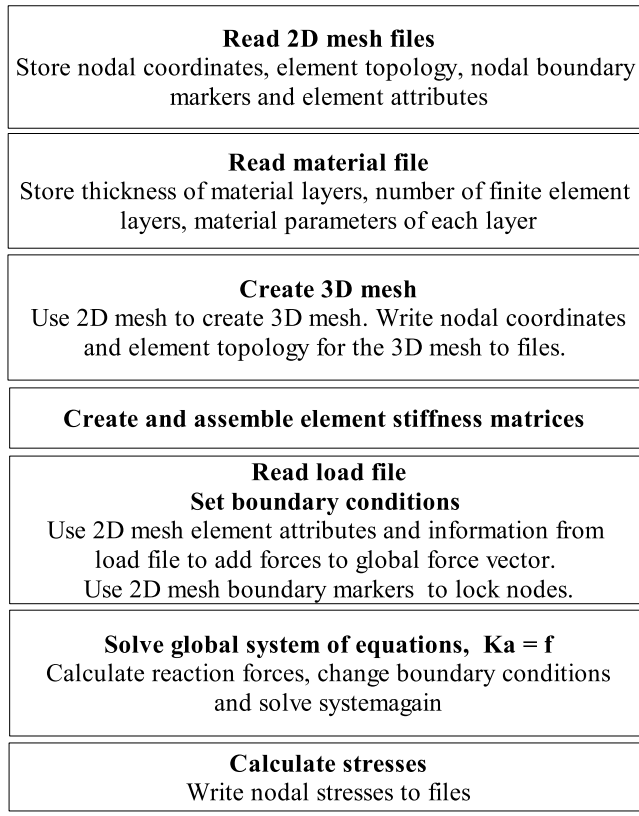


Figure 4.2: *The different steps in obtaining the solution*

4.3 Graphical User Interface

4.3.1 General remarks

The graphical user interface was developed in Python, with the extension module wxPython [21]. wxPython is a cross-platform GUI toolkit, enabling the use of the exact same code on several different platforms, such as Microsoft Windows, Mac OS X and Linux. The basic outline of the GUI was created using the GUI designer wxGlade [22].

The graphical user interface was designed to be as user-friendly as possible. Alas, this user-friendliness also implicates that fewer options are available to the user. The first apparent restriction is on the geometry of the structure to be analyzed. It would not be too hard to program a feature where the user independently draws the 2D shape of the structure, but it would require more inputs from the user. (The development of such a feature was actually commenced but abandoned for the sake

of user-friendliness.) Instead, the user is now confined to analyze rectangular glass panes only. This is not too disturbing a restriction, since most glass panes in use for balustrades and building facades are certainly rectangular. The entire GUI is based on tabs, where a certain type of input is entered under a certain tab. In a sense, the tabs represent a route for the user to follow. When the fields in the first tab have been filled, the user clicks the next tab and fills those input fields, and so on. When all inputs have been entered, it is timely to run the analysis. The analysis is started by pressing the "!"-icon.

4.3.2 User inputs

Structure geometry

The size of the rectangular pane is given in terms of a width and a height. The user then chooses whether the analysis concerns laminated glass, consisting of two glass plates glued together by an intermediate foil of PVB, or a single-layered glass pane. The thickness of the involved material layers are specified. This is entered under the first tab that appears when the program is started, as shown in Figure 4.3.
The bolt positions are specified under the second tab, shown in Figure 4.4. An arbitrary number of bolts may be specified. Possibilities to specify different sizes on

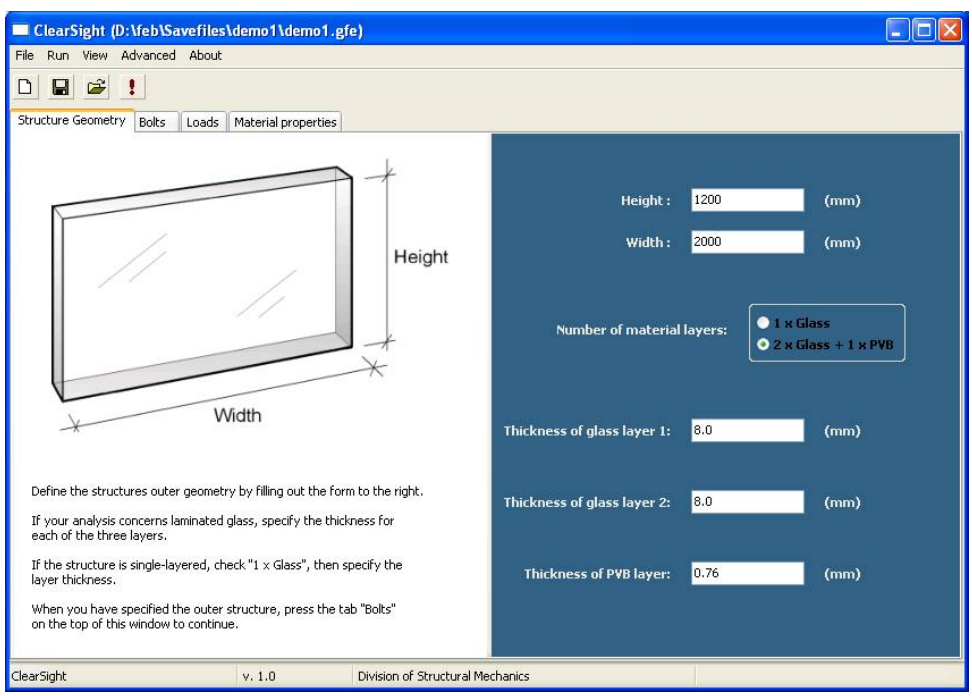


Figure 4.3: *The first tab where the structures outer geometry is entered*

bolts would certainly be an useful option, but in this version only one type of bolt has been implemented, namely the cylindrical bolt with an inner diameter $d = 30mm$ and an outer diameter $D = 50mm$. The positions of the bolts are entered in a table, where the coordinates refer to an origin located in the bottom left corner of the structure.

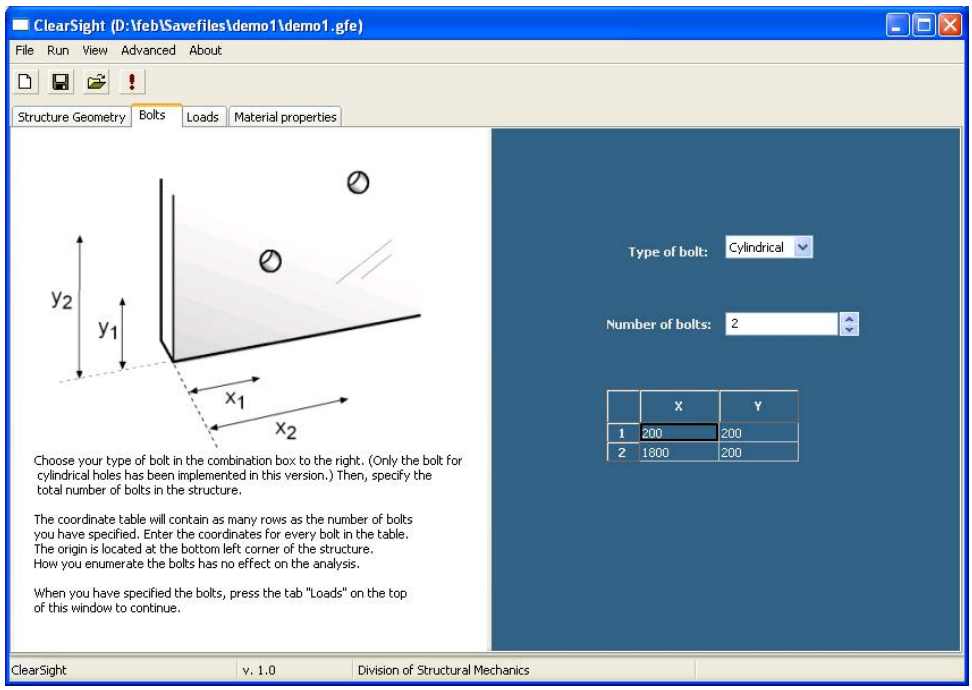


Figure 4.4: *The second tab where the number of bolts and their positions are defined*

Loads

Building codes prescribe what type of loads different structures should be designed to carry, as well as the size of the loads in different situations. Balustrades should be designed to carry a line load acting along the top edge, whereas facades should be designed to carry a distributed load. Both these load types have been implemented, and the size of the loads are entered under the third tab. See Figure 4.5.

Material properties

The fourth and last tab allows the user to specify material parameters, such as Young's modulus, Poisson's ratio and density, for the glass layers as well as the

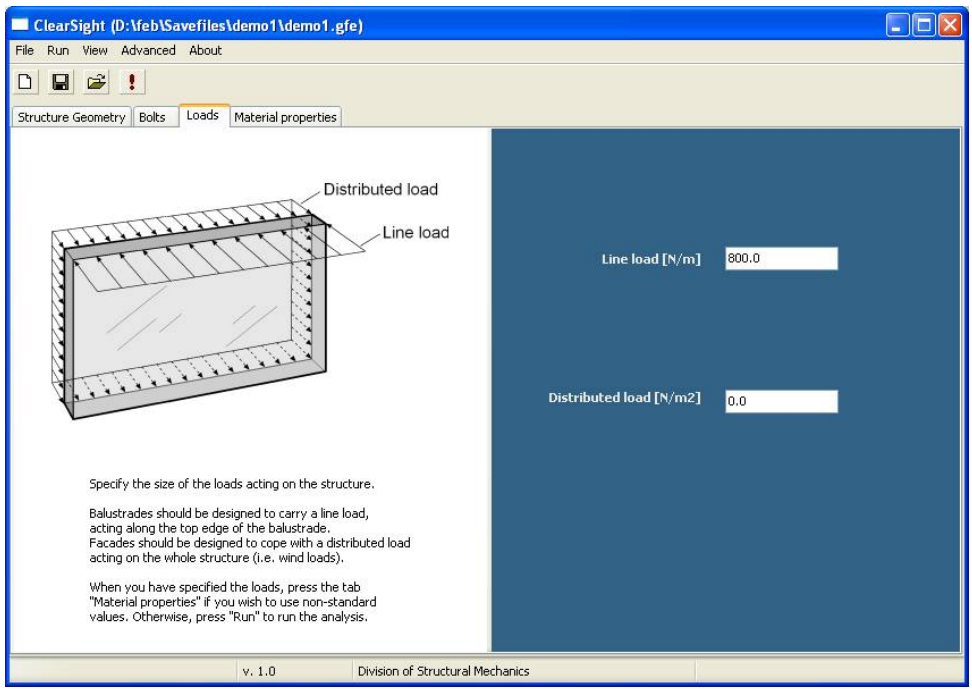


Figure 4.5: *The third tab where the loads are defined. A line load acting on the top edge of the glass or/and a distributed load acting on the whole surface is entered.*

PVB layer. In addition, the glass' permissible stress may also be specified. The permissible stress is not used in the calculations, but in the post-processor where the user can choose to view areas where the stresses exceed the permissible stress.

Meshing properties

Apart from the structure data entered in the four tabs, there is also a couple of mesh related properties that may be changed. As mentioned, three dimensional wedge elements are used in the calculations. The 3D mesh is extended from a two-dimensional triangular mesh. The triangular mesh is created by an external mesh generator, as discussed in Section 4.4. The properties that may be changed affect the accuracy of the solution, and they are therefore accessed through the "Advanced"-menu in order to keep novice users away.

The first option deals with the coarseness of the two dimensional mesh. A maximum element area that applies to the whole mesh may be given, as well as a maximum area that only applies to areas within a given distance from holes (=bolts). The number of element nodes on the hole boundary may also be specified. All these

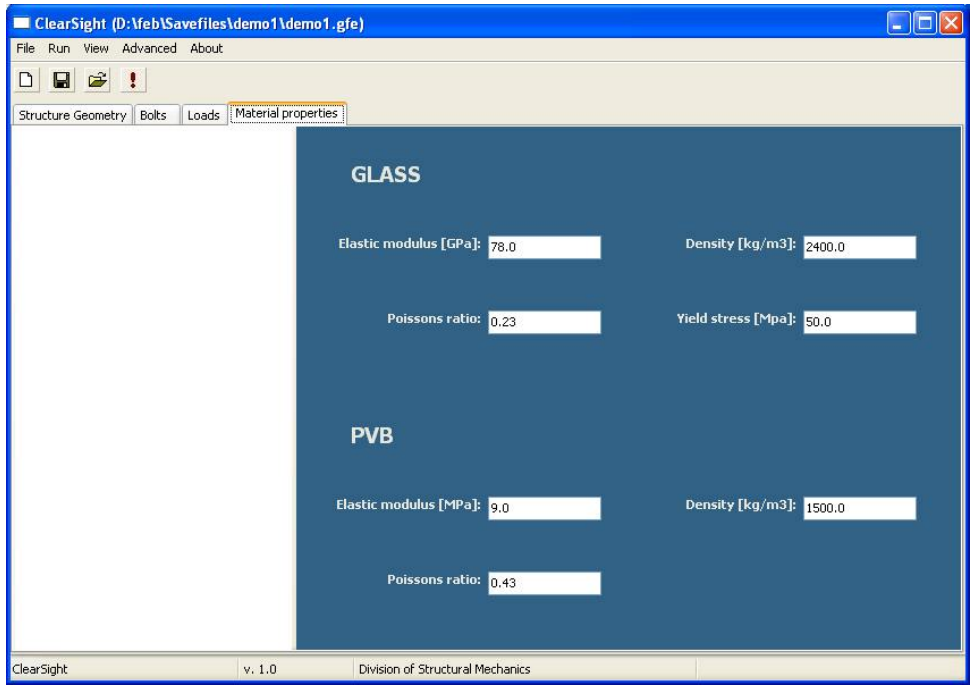


Figure 4.6: *The fourth tab where the material parameters are set*

parameters have tuned standard values in order to give satisfactory results in most cases.

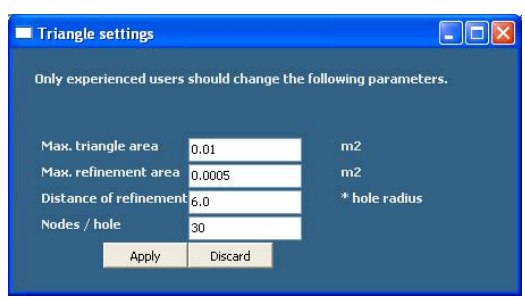


Figure 4.7: *Parameters affecting the 2D triangular mesh*

The second option concerns the three dimensional mesh, namely the number of finite element layers of each material layer. A greater number of element layers gives better accuracy, but it also implicates a longer solution time. Standard values are set.

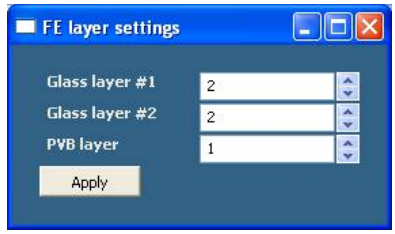


Figure 4.8: *The option where the number of elements in the thickness direction is set*

Saving and loading files

Functions for saving and loading work files have also been added. When the user chooses to save a file, all the inputs that have been entered are written to one single text file that can be opened and read at any time.

4.3.3 Running the analysis

When the analysis is executed, all the inputs are written to a number of text files. From the geometry inputs, a so called poly-file is written, which is basically a list of vertices and segments that define the structure. This file is used by the external mesh generator. Every segment may be given an attribute, which is passed on to the nodes created on that segment by the mesh generator. The attributes are used to keep track of which nodes and elements are subjected to certain boundary conditions, such as a load or a prescribed displacement. This is further described in Section 4.4.

A load file is written where the magnitudes of the two different load types are specified. A material file is written, where the material parameters for the different layers are specified. These material parameters are used when the element stiffness matrices are calculated, as well as when the stresses are evaluated.

When these files have been created, an external mesh generator is executed. The mesh generator reads the poly-file and creates a two-dimensional triangular mesh, which is stored in a couple of files. These files are read, and an area-file is created. The area file is, along with the original mesh files, used by the mesh generator to create a new, refined mesh. These different files are described in Section 4.4.

Finally, the computational code is executed. When the computations have been carried out, the post-processor described in Section 4.9 is executed.

4.4 Mesh Generation

4.4.1 Introduction

Given a certain structure, generating a mesh manually is necessarily not a very difficult, if yet tedious, task. Constructing algorithms capable of generating quality meshes from arbitrary regions, suitable for use in FE-analysis, is on the contrary a truly intricate problem that goes well outside the scope of this project. Therefore, the purpose was never to write such an algorithm, but merely to employ any fast and stable mesh generator available in the public domain.

There are only a few mesh generators available online for free, of which even fewer are capable of creating meshes of high quality. A great number of commercial mesh generators exist, that can create both two- and three-dimensional meshes comprising different element types. They are often rather expensive, and hence they do not fit the budget of this project. Triangle, written by Jonathan Shewchuk [15], though, is a free, almost inconceivably fast, stable and easy-to-use program that constructs a quality triangular mesh from a region defined by vertices and segments. An attempt was made to use the free program CQmesh [16], which takes a triangular mesh as input, an tries to convert it to a quadrilateral mesh. This worked very poorly, and was therefore not implemented.

4.4.2 Creating a 2D mesh using Triangle

Input/Output Files

Triangle is capable of creating meshes given different inputs and constraints. Here, though, only the features used in this project will be described.

Triangle reads so called poly-files, representing a Planar Straight Line Graph (PSLG) which, by definition, is simply a list of vertices and segments. Vertices may be located on the structure boundary, or inside the structure boundary. A vertex located on the boundary is a point that connects segments. The outer region of the structure is entirely defined by segments, see Figure 4.9 below.

Triangle creates a so called *constrained conforming Delaunay triangulation* by inserting nodes on the segments and in the region interior. The maximum triangle size may be passed as an argument when calling Triangle, and will apply to all the triangles in the mesh. It is also possible to specify the maximum triangle area within a region, thereby giving different subregions different maximum triangle areas, as discussed below. Constraints on triangle angles can also be enforced; for example one might not want to allow smaller angles than 30 degrees in a mesh for finite element purposes.

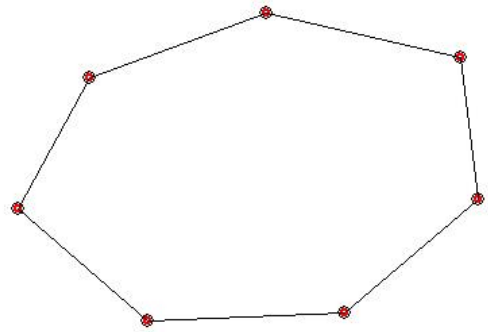


Figure 4.9: *A closed polygon*

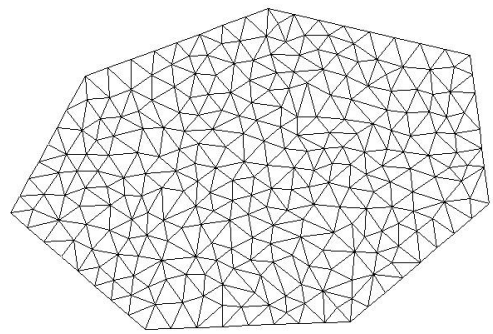


Figure 4.10: *A triangular mesh generated by Triangle*

For easy identification of nodes, a boundary marker can be assigned to any vertex or segment in the input poly-file. The boundary marker assigned to a segment, will be passed on to nodes inserted on the segment. This is used in order to identify which nodes are subjected to the line load, when analyzing a balustrade. Every region inside a closed curve, may be given attributes that only apply to the triangle elements within that region. This is used when identifying elements subjected to certain conditions, see further Section 4.7 about modelling the bolt fixings. In the input poly-file, a regional attribute is given a x- and a y-coordinate where the attribute first applies. Triangle will then pass that regional attribute to all the elements in the entire region or subregion enclosed by segments.

Triangle can also create holes in the mesh. A starting coordinate for the hole is defined, from which Triangle will start deleting triangles in all directions until it reaches a segment. Creating intrinsically non-linear shapes, such as a circular hole, is naturally not possible since a segment is linear. The remedy is to use as many of the piecewise linear segments as needed to decently represent the non-linear shape. This might result in very small triangles around the curved region.

What Triangle most importantly produces, is a node- and an element-file. The node file is a list of all the nodes, their coordinates and boundary markers. The element-file is a list of the topology and the regional attribute assigned to the region in which the triangle belongs. The nodes are enumerated counter clockwise.

Apart from creating linear triangular elements as discussed above, Triangle can also create subparametric quadratic elements consisting of six nodes rather than three, where the term subparametric means that the triangle edges are still straight lines. A subparametric quadratic triangle is geometrically identical to a linear triangle, the only difference being the number of nodes. The three extra nodes in a quadratic triangle are inserted at the midpoint of each edge in the linear triangle, as shown below, enabling the use of quadratic shape functions. In following sections, the three nodes located in the corners of a triangle will be referred to as corner nodes.

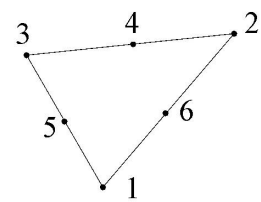


Figure 4.11: *A subparametric triangle element.*

Mesh refinement

There are supposedly several ways to refine an existing mesh in Triangle. One way is to create a so called area-file, which is simply a list of all the triangles, and a maximum area for every triangle. If the maximum area for a certain triangle is set to a smaller value than its actual size, additional nodes will be inserted, and the triangle will be divided so that the requirement on maximum triangle size is met. When refining a mesh using this method, the area-file is passed to Triangle along with the already created node- and element-files. New node- and element-files will then be created.

This way of imposing a smaller triangle size is very useful when one wishes to refine the mesh around a hole. In this project, the holes represent bolt fixings that give raise to large stress and strain gradients around the hole. By running Triangle once, the elements situated within a given distance from the holes can be identified. By means of an area-file, these elements are assigned a smaller maximum area that Triangle enforces when executed a second time. Thus, a mesh better capable of dealing with large gradients is obtained.

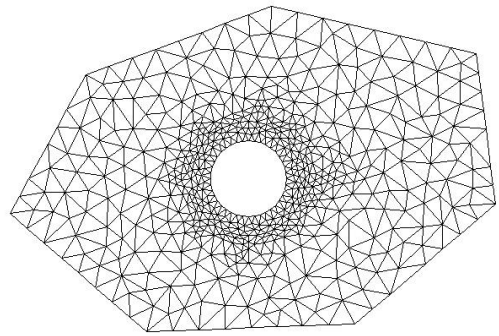


Figure 4.12: *A mesh with a hole. The mesh is refined close to the hole.*

4.4.3 Implementation

When the user chooses to run the analysis, a routine in the GUI creates a poly-file that corresponds to the geometry input. Two polygons are used at each hole; the inner polygon defines the hole, and the outer polygon is used so that the elements inside it can be given a regional attribute. This regional attribute is used when finding the elements where certain boundary conditions apply, see further Section 4.7. Figure 4.13 shows the polygons for a balustrade with two bolts, specified in the GUI according to Section 4.3. Figure 4.14 shows the resulting mesh. Note that no area constraints have been given in this example, and that only 10 nodes are used to define each hole. The region near the hole, that has been given a regional attribute, is drawn with red color. The poly-file, as well as parts of the node- and element-files for this example can be found in Appendix 1.

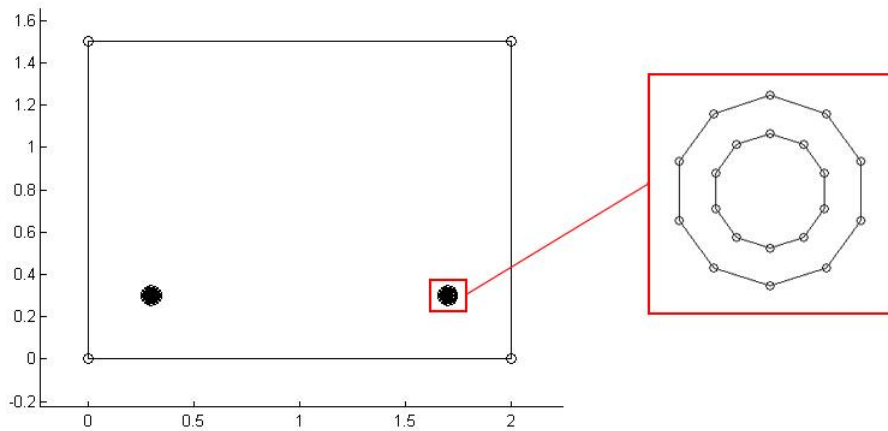


Figure 4.13: *The vertices and segments specified in the poly-file.*

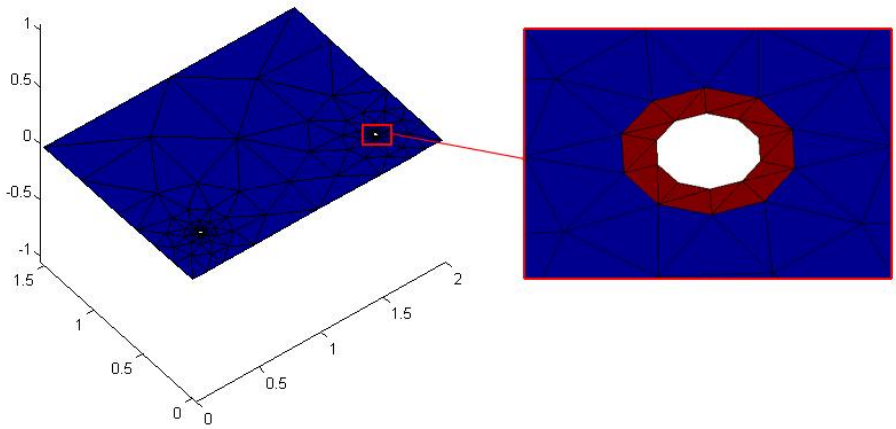


Figure 4.14: *The resulting triangular mesh, with the elements having a regional attribute drawn in red.*

4.4.4 Extending a 2D triangular mesh to a 3D prismatic mesh

Even in some cases of plane, equally thick structures as dealt with in this project, one might have to resort to three dimensional solid elements when two dimensional elements fail to properly model the structure's behavior. Nevertheless, a two dimensional mesh might still be useful. By giving every node in the 2D mesh a depth coordinate, and 'copying' these nodes to another depth level, a mesh comprising three dimensional wedge elements can be obtained. By 'copying a node', it is meant that a new node is created with the same planar coordinates as the node being 'copied'.

For the purpose of creating a mesh composed of 15-node wedge elements, the feature in Triangle yielding subparametric quadratic triangles is used. The 15-node wedge element has the structure and nodal labelling according to Figure 4.15.

A mesh comprising this kind of elements may be created by the following steps:

1. Find out how many triangle corner nodes the two-dimensional mesh comprises. Use is taken of the topology list given by Triangle. Triangle puts every element's three corner nodes first in topology list, as in Figure 4.11.

   If $triNnd$ is the number of nodes in the 2D mesh, $triNel$ the number of elements in the 2D mesh, $corNnd$ the number of triangle corner nodes in the 2D mesh, and $nLayers$ is the number of element layers in the final mesh, then

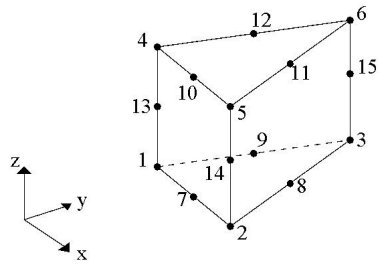


Figure 4.15: *A 15-node wedge element.*

the final mesh will consist of

$totNnd = (nLayers + 1) * triNnd + nLayers * corNnd$ nodes
and
$totNel = nLayers * triNel$ elements.

2. All the nodes from the two-dimensional mesh are copied to new levels of depth. Every layer of nodes now represent one side, top and/or bottom, of a set of wedge elements. These nodes are simply labelled after their parent node.

3. All the triangle corner nodes are copied to the midpoint at each layer. The labels of the midpoint nodes are stored in a matrix, with the row index corresponding to the parent node label, and the column index corresponding to the element layer in which the node is located. The topology for the wedge mesh is defined using the topology for the triangular mesh.

In this project, the GUI creates a file that specifies the number of material layers, the thickness of each material layer, as well as the number of finite elements in the thickness direction of each material layer. A routine written in Fortran then creates the wedge mesh according to the steps above.

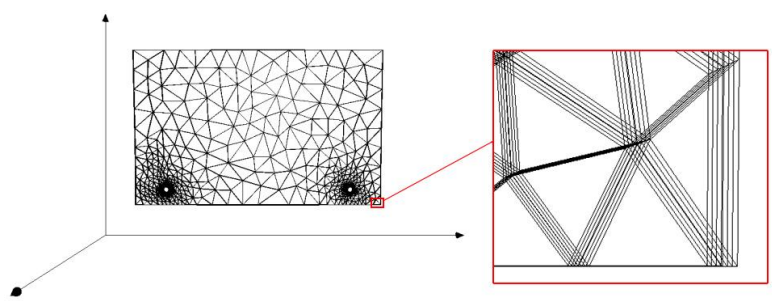


Figure 4.16: *A mesh composed of wedge elements, created according to the procedure described above.*

4.5 The 15-node Wedge Element

4.5.1 Evaluating the element stiffness

The wedge element may take an infinite number of configurations, and it is not possible to establish a general expression of the shape functions valid for any configuration. Therefore, one uses shape functions determined for the element in one special configuration, the *parent domain*. Employing isoparametric mapping, the shape functions are then used to map the element into an arbitrary configuration in the *global domain*, see Figure 4.17.

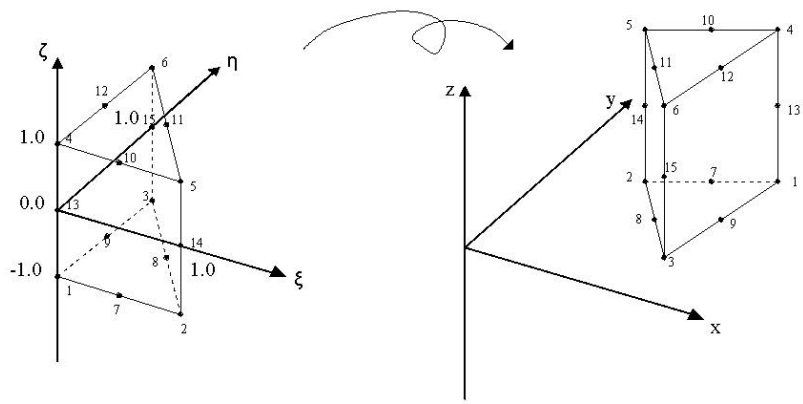


Figure 4.17: *Mapping from parent to global domain.*

$$x = x(\xi,\eta,\zeta) = \mathbf{N^e x^e}; y = y(\xi,\eta,\zeta) = \mathbf{N^e y^e}; z = z(\xi,\eta,\zeta) = \mathbf{N^e z^e}; \tag{4.1}$$

where $\mathbf{x^e}$,$\mathbf{y^e}$ and $\mathbf{z^e}$ are vectors with the nodal coordinates for the element in the global domain. $\mathbf{N^e}$ is a vector containing the 15 element shape functions, expressed in ξ,η,ζ-coordinates. To obtain a unique mapping, 4.1 is differentiated using the the chain rule

$$\begin{bmatrix} dx \\ dy \\ dz \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \zeta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} & \frac{\partial y}{\partial \zeta} \\ \frac{\partial z}{\partial \xi} & \frac{\partial z}{\partial \eta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} \begin{bmatrix} d\xi \\ d\eta \\ d\zeta \end{bmatrix} \tag{4.2}$$

The matrix above is called the Jacobian matrix, and its determinant is called the Jacobian. With the isoparametric mapping, 4.1, the Jacobian matrix is written

as

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{N^e}}{\partial \xi}\mathbf{x^e} & \frac{\partial \mathbf{N^e}}{\partial \eta}\mathbf{x^e} & \frac{\partial \mathbf{N^e}}{\partial \zeta}\mathbf{x^e} \\ \frac{\partial \mathbf{N^e}}{\partial \xi}\mathbf{y^e} & \frac{\partial \mathbf{N^e}}{\partial \eta}\mathbf{y^e} & \frac{\partial \mathbf{N^e}}{\partial \zeta}\mathbf{y^e} \\ \frac{\partial \mathbf{N^e}}{\partial \xi}\mathbf{z^e} & \frac{\partial \mathbf{N^e}}{\partial \eta}\mathbf{z^e} & \frac{\partial \mathbf{N^e}}{\partial \zeta}\mathbf{z^e} \end{bmatrix} \tag{4.3}$$

Generally, the element stiffness matrix is calculated as

$$\mathbf{K^e} = \int_V \mathbf{B^T D B} dV \tag{4.4}$$

where

$$\mathbf{B} = \tilde{\nabla}\mathbf{N^e} \tag{4.5}$$

$$\tilde{\nabla} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 \\ 0 & \frac{\partial}{\partial y} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{\partial y} \end{bmatrix} \mathbf{N^e} = \begin{bmatrix} N_1 & 0 & 0 & N_2 & 0 & 0 & .. & .. & N_{15} & 0 & 0 \\ 0 & N_1 & 0 & 0 & N_2 & 0 & .. & .. & 0 & N_{15} & 0 \\ 0 & 0 & N_1 & 0 & 0 & N_2 & .. & .. & 0 & 0 & N_{15} \end{bmatrix} \tag{4.6}$$

i.e.

$$\mathbf{B} = \begin{bmatrix} \frac{\partial N_1}{\partial x} & 0 & 0 & \frac{\partial N_2}{\partial x} & 0 & 0 & .. & .. & \frac{\partial N_{15}}{\partial x} & 0 & 0 \\ 0 & \frac{\partial N_1}{\partial y} & 0 & 0 & \frac{\partial N_2}{\partial y} & 0 & .. & .. & 0 & \frac{\partial N_{15}}{\partial y} & 0 \\ 0 & 0 & \frac{\partial N_1}{\partial y} & 0 & 0 & \frac{\partial N_2}{\partial y} & .. & .. & 0 & 0 & \frac{\partial N_{15}}{\partial y} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & 0 & \frac{\partial N_2}{\partial y} & \frac{\partial N_2}{\partial x} & 0 & .. & .. & \frac{\partial N_{15}}{\partial y} & \frac{\partial N_{15}}{\partial x} & 0 \\ \frac{\partial N_1}{\partial z} & 0 & \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial z} & 0 & \frac{\partial N_2}{\partial x} & .. & .. & \frac{\partial N_{15}}{\partial z} & 0 & \frac{\partial N_{15}}{\partial x} \\ 0 & \frac{\partial N_1}{\partial z} & \frac{\partial N_1}{\partial y} & 0 & \frac{\partial N_2}{\partial z} & \frac{\partial N_2}{\partial y} & .. & .. & 0 & \frac{\partial N_{15}}{\partial z} & \frac{\partial N_{15}}{\partial y} \end{bmatrix} \tag{4.7}$$

$N_1, N_2, ..., N_{15}$ are the element shape functions given in terms of ξ, η and ζ coordinates. The problem is that $\mathbf{B}$ contains derivatives of the shape functions with

respect to x,y and z.

The derivatives $\frac{\partial N_i}{\partial x}$, $\frac{\partial N_i}{\partial y}$ and $\frac{\partial N_i}{\partial z}$ in Equation 4.7 can be determined by first differentiating the shape functions with respect to ξ, η and ζ.

$$\begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \\ \frac{\partial N_i}{\partial \zeta} \end{bmatrix} = \begin{bmatrix} \frac{\partial N_i}{\partial x}\frac{\partial x}{\partial \xi} + \frac{\partial N_i}{\partial y}\frac{\partial y}{\partial \xi} + \frac{\partial N_i}{\partial z}\frac{\partial z}{\partial \xi} \\ \frac{\partial N_i}{\partial x}\frac{\partial x}{\partial \eta} + \frac{\partial N_i}{\partial y}\frac{\partial y}{\partial \eta} + \frac{\partial N_i}{\partial z}\frac{\partial z}{\partial \eta} \\ \frac{\partial N_i}{\partial x}\frac{\partial x}{\partial \zeta} + \frac{\partial N_i}{\partial y}\frac{\partial y}{\partial \zeta} + \frac{\partial N_i}{\partial z}\frac{\partial z}{\partial \zeta} \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{bmatrix} \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix} \tag{4.8}$$

i.e.

$$\begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix} = (\mathbf{J^T})^{-1} \begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \\ \frac{\partial N_i}{\partial \zeta} \end{bmatrix} \tag{4.9}$$

Due to the complexity of $\mathbf{B}$, a simple expression of how to evaluate the stiffness matrix cannot be given, but at this point it is apparent that all the components of $\mathbf{B}$ can be determined for any point (ξ_j, η_j, ζ_j), by means of Equations 4.7, 4.9 and 4.3. The integral 4.4 can be evaluated by means of the transformation described by 4.10.

The integration of an arbitrary function $f(x, y, z)$ can be transformed and evaluated as

$$\int_V f(x,y,z)dV = \int_{-1}^{1}\int_{0}^{1-\eta}\int_{0}^{1} f(x(\xi,\eta,\zeta), y(\xi,\eta,\zeta), z(\xi,\eta,\zeta))det(\mathbf{J})d\xi d\eta d\zeta \tag{4.10}$$

See for example Ottosen [5].

The element shape functions, reproduced below, were taken from the ABAQUS manual.

$$
\begin{aligned}
N_1 &= \tfrac{1}{2}((1-\xi-\eta)(2(1-\xi-\eta)-1)(1-\zeta)-(1-\xi-\eta)(1-\zeta^2)) \\
N_2 &= \tfrac{1}{2}(\xi(2\xi-1)(1-\zeta)-\xi(1-\zeta^2)) \\
N_3 &= \tfrac{1}{2}(\eta(2\eta-1)(1-\zeta)-\eta(1-\zeta^2)) \\
N_4 &= \tfrac{1}{2}((1-\xi-\eta)(2(1-\xi-\eta)-1)(1+\zeta)-(1-\xi-\eta)(1-\zeta^2)) \\
N_5 &= \tfrac{1}{2}(\xi(2\xi-1)(1+\zeta)-\xi(1-\zeta^2)) \\
N_6 &= \tfrac{1}{2}(\eta(2\eta-1)(1+\zeta)-\eta(1-\zeta^2)) \\
N_7 &= 2(1-\xi-\eta)\xi(1-\zeta) \\
N_8 &= 2\xi\eta(1-\zeta) \\
N_9 &= 2\eta(1-\xi-\eta)(1-\zeta) \\
N_{10} &= 2(1-\xi-\eta)\xi(1+\zeta) \\
N_{11} &= 2\xi\eta(1+\zeta) \\
N_{12} &= 2\eta(1-\xi-\eta)(1+\zeta) \\
N_{13} &= (1-\xi-\eta)(1-\zeta^2) \\
N_{14} &= \xi(1-\zeta^2) \\
N_{15} &= \eta(1-\zeta^2)
\end{aligned}
\tag{4.11}
$$

The differentiation of the shape functions with respect to ξ,η and ζ were carried out in Maple. The stiffness of the 15 node wedge element cannot, due to the complex shape functions, be evaluated by exact integration and hence numerical integration is required.

There is no Gaussian integration formula for integrating a function over the entire wedge region at once. The integration is therefore carried out in the triangle ξ, η-plane and in the ζ-direction separately. $3x3$ sample points are used for the integration of the element stiffness; three sets of three sample points in three different ξ, η-planes. The stress and strain evaluation of this element is treated in section 4.6,

and it will be shown that 18 sample points are required for that integration in order to use the least-square extrapolation technique. For now, it is sufficient to consider the element with 9 sample points. The numerical integration of a function $f(\xi, \eta, \zeta)$ over the wedge region can be written as

$$\int_{-1}^{1}\int_{0}^{1-\eta}\int_{0}^{1} f(\xi,\eta,\zeta)d\xi d\eta d\zeta = \sum_{i=1}^{3}\sum_{j=1}^{3} f(\xi_j,\eta_j,\zeta_i)wpr_i wps_j \tag{4.12}$$

where the summation over j represents the integral over a triangle area in the ξ, η-plane, and the summation over i represents the integral in the thickness direction, ζ. Every Gauss point has one weight depending on its ξ, η-coordinates, wps, and one weight depending on its ζ-coordinate, wpr. The locations and weights of the Gauss points can be found in [6]. They are shown in Table 4.1 and 4.2. With Equations 4.4, 4.10 and 4.12, the element stiffness can be calculated as

$$\mathbf{K^e} = \int_{-1}^{1}\int_{0}^{1-\eta}\int_{0}^{1} \mathbf{B^T D B} det\mathbf{J} d\xi d\eta d\zeta = \sum_{i=1}^{3}\sum_{j=1}^{3} \mathbf{B^T D B} det\mathbf{J} wpr_i wps_j \tag{4.13}$$

where $\mathbf{B}$, $\mathbf{D}$ and $\mathbf{J}$ are evaluated at the Gauss point with the coordinates (ξ_j, η_j, ζ_i) and the corresponding planar weight wps_j and thickness weight wpr_i.

A few simple tests were performed and compared to identical examples in ABAQUS, in order to confirm the correct implementation of the element stiffness calculation. These tests are not shown here.

Table 4.1: *Location of Gauss points in the (ξ, η)-plane with the planar Gauss weight wps*

j	ξ_j	η_j	Weight wps_j
1	0.16666666666667	0.16666666666667	0.16666666666667
2	0.66666666666667	0.16666666666667	0.16666666666667
3	0.16666666666667	0.66666666666667	0.16666666666667

Table 4.2: *Location of Gauss points in (ζ)-direction with the Gauss weight wpr*

i	ζ_i	Weight wpr_i
1	-0.774596669241483	0.55555555555555556
2	0.0	0.8888888888888889
3	0.774596669241483	0.55555555555555556

4.5.2 Nodal forces for the 15-node wedge element

Body forces

The effect on the element nodal load vector due to body forces can be expressed as

$$\mathbf{f_l^e} = \int_V \tilde{\mathbf{N}}^{\mathbf{eT}} \mathbf{b} dV \tag{4.14}$$

where $\tilde{\mathbf{N}}^{\mathbf{e}}$ is the element shape functions in x, y, z-coordinates, V is the volume of the element and $\mathbf{b}$ is the body force vector [5]. dV denotes an incremental volume in the xyz-space.

For the isoparametric wedge element with the shape functions expressed in ξ, η, ζ-coordinates, by using Equations 4.1 and 4.10, and the same numerical integration used for evaluating the element stiffness, Equation 4.14 can be evaluated as

$$\mathbf{f_l^e} = \sum_{i=1}^{3}\sum_{j=1}^{3} \mathbf{N}^{\mathbf{eT}} \mathbf{b} det\mathbf{J} wpr_i wps_j \tag{4.15}$$

where $\mathbf{N}^{\mathbf{eT}}$, $\mathbf{b}$ and $\mathbf{J}$ are evaluated at the Gauss points with the coordinates (ξ_j, η_j, ζ_i) and the corresponding planar weights wps_j and thickness weights wpr_i.

The body force vector $\mathbf{b}$ is

$$\mathbf{b} = \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} \tag{4.16}$$

which for the configuration shown in Figure 4.16, where the gravitational force acts in the negative y-direction, is

$$\mathbf{b} = \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} 0 \\ -\gamma * g \\ 0 \end{bmatrix} \tag{4.17}$$

where γ is the density $[kg/m^3]$ of the material, and g is the gravitational acceleration $g \approx 9.81\ m/s^2$.

The nodal forces due to body forces are calculated in the same routine as the stiffness matrix.

Distributed loads

Facades should be designed to carry a distributed load, i.e. wind load, acting on the outside surface of the glass pane. The size of this load depends on which elevation above ground the facade is located.

The effect on the element nodal load vector of a distributed load is expressed as

$$\mathbf{f_b^e} = \int_{S_\alpha} \tilde{\mathbf{N}}^{\mathbf{eT}} \mathbf{h} dS \tag{4.18}$$

where $\tilde{\mathbf{N}}^{\mathbf{e}}$ is the element shape functions in x, y, z-coordinates, S_α is the surface of the element and $\mathbf{h}$ is a known traction vector [5]. dS denotes the incremental area of the element boundary surface located in the xyz-space.

Consider a distributed load acting on the top surface of the element in Figure 4.18. The element is shown both in the parent domain and in the global domain. For this surface, $\zeta = 1$, i.e. $d\zeta = 0$. Let S_α in Equation 4.18 denote this surface, whereas $\mathbf{h}$ is the traction vector due to the distributed load. Consider the two straight lines given by $\xi = C_1$ and $\eta = C_2$, shown in Figure 4.19. Now, an incremental vector $d\xi \neq 0$ along $\eta = C_2$ (where $d\eta = 0$ and $d\zeta = 0$), transforms into the xyz-space according to

$$\mathbf{a} = \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix} = \mathbf{a_1} d\xi \tag{4.19}$$

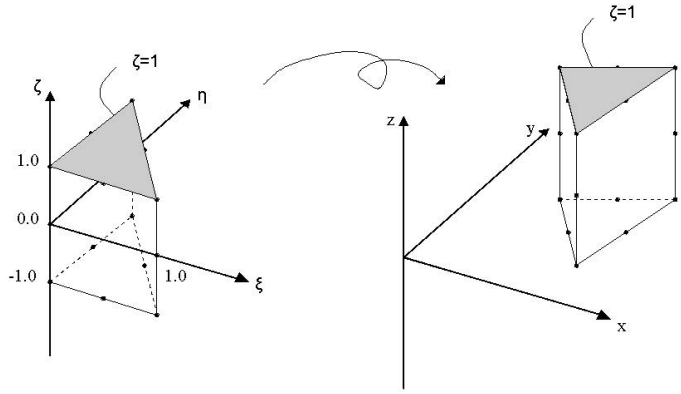


Figure 4.18: *A wedge element with a distributed load acting on its top surface, shown in the parent domain and in the global domain.*

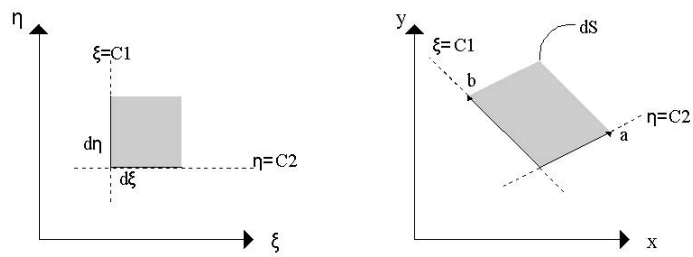


Figure 4.19: *The mapping of an incremental area on the wedge's top surface, from the parent domain to the global domain.*

where

$$\mathbf{a_1} = \begin{bmatrix} \frac{\partial x}{\partial \xi} \\ \frac{\partial y}{\partial \xi} \\ \frac{\partial z}{\partial \xi} \end{bmatrix} \tag{4.20}$$

Similarly, an incremental vector given by $d\eta \neq 0$ (where $d\xi = 0$ and $d\zeta = 0$) transforms into the xyz-space according to

$$\mathbf{b} = \begin{bmatrix} dx \\ dy \\ dz \end{bmatrix} = \mathbf{b_1} d\eta \tag{4.21}$$

where

$$\mathbf{b_1} = \begin{bmatrix} \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \eta} \\ \frac{\partial z}{\partial \eta} \end{bmatrix} \tag{4.22}$$

Notice that $\mathbf{a_1}$ and $\mathbf{b_1}$ are the first and second columns respectively, of the Jacobian matrix $\mathbf{J}$ in Equation 4.3. See also Figure 4.19. The area spanned by the two vectors $\mathbf{a}$ and $\mathbf{b}$ can be expressed with the cross-product as

$$dS = |\mathbf{a} \times \mathbf{b}| \tag{4.23}$$

and by using Equations 4.19-4.23, we get

$$dS = |\mathbf{a_1} \times \mathbf{b_1}||d\xi||d\eta| \tag{4.24}$$

An arbitrary function $f = f(x, y, z)$ can now be integrated over the wedge's top surface as

$$\int_{S_\alpha} f(x, y, z)dS = \int_0^{1-\eta} \int_0^1 f(x(\xi, \eta, \zeta), y(\xi, \eta, \zeta), z(\xi, \eta, \zeta))|\mathbf{a_1} \times \mathbf{b_1}||d\xi||d\eta| \tag{4.25}$$

and for Equation 4.18 we get

$$\mathbf{f_b^e} = \int_{S_{h\alpha}} \tilde{\mathbf{N}}^{\mathbf{eT}} \mathbf{h} dS = \int_0^{1-\eta} \int_0^1 \mathbf{N^{eT}} \mathbf{h} |\mathbf{a_1} \times \mathbf{b_1}||d\xi||d\eta| \tag{4.26}$$

where $\mathbf{N^e}$ is a matrix with the shape functions in ξ, η, ζ-coordinates. As in the case of the stiffness matrix, Equation 4.26 also needs to be evaluated numerically. This is carried out in the same manner, using Gaussian integration. Three integration points are used. They are located at $\zeta = 1$ with the ξ, η-coordinates according to Table 4.1.

$$\mathbf{f_b^e} = \sum_{j=1}^{3} \mathbf{N^{eT}} \mathbf{h} |\mathbf{a_1} \times \mathbf{b_1}| wps_j \tag{4.27}$$

where $\mathbf{N^{eT}}$, $\mathbf{a_1}$ and $\mathbf{b_1}$ are to be evaluated at the Gauss points with the coordinates $(\xi_j, \eta_j, 1)$ and the corresponding planar weights wps_j. See Table 4.1.

With the elements configured as in Figure 4.16, the vector $\mathbf{h}$ is

$$\mathbf{h} = \begin{bmatrix} h_x \\ h_y \\ h_z \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -q \end{bmatrix} \tag{4.28}$$

where q $[N/m^2]$ is the distributed wind load acting on the back side of the glass pane.

Line loads

Building codes prescribe that balustrades should be designed to carry a line load. The line load acts along the top edge of a balustrade. The line load's contribution to the nodal load vector may be evaluated by integrating an expression similar to 4.18, but in this project the loads are simply lumped to the nodes subjected to the load.

If the boundary is subjected to a line load Q $[N/m]$, we want to know the equivalent nodal loads. If the distance between all the nodes on the boundary is d, then the equivalent nodal load would be $d * Q$ for all nodes except one located on the edge of the boundary. For such a node, the equivalent nodal load would be $0.5 * d * Q$. When using an unstructured mesh, the distance between the nodes is not uniform, so the the nodal coordinates are used for calculating the distances.

4.6 Stress and Strain Evaluation

4.6.1 Stresses and strains at Gauss points

In order to evaluate the strains in an element, use is once again taken of the derivatives of the shape functions, or more specifically, the matrix **B** (4.7). The strains are calculated as $\varepsilon_\mathbf{i} = \mathbf{B_i u}$ where $\mathbf{B_i}$ is the matrix **B** evaluated at sample point i, and **u** is the element nodal displacements. The strains are calculated at each of the 9 Gauss points in Tables 4.1 and 4.2.

$$\begin{bmatrix} \varepsilon^i_{xx} \\ \varepsilon^i_{yy} \\ \varepsilon^i_{zz} \\ \varepsilon^i_{xy} \\ \varepsilon^i_{xz} \\ \varepsilon^i_{yz} \end{bmatrix} = \begin{bmatrix} \frac{\partial N_1}{\partial x} & 0 & 0 & \frac{\partial N_2}{\partial x} & 0 & 0 & .. & .. & \frac{\partial N_{15}}{\partial x} & 0 & 0 \\ 0 & \frac{\partial N_1}{\partial y} & 0 & 0 & \frac{\partial N_2}{\partial y} & 0 & .. & .. & 0 & \frac{\partial N_{15}}{\partial y} & 0 \\ 0 & 0 & \frac{\partial N_1}{\partial y} & 0 & 0 & \frac{\partial N_2}{\partial y} & .. & .. & 0 & 0 & \frac{\partial N_{15}}{\partial y} \\ \frac{\partial N_1}{\partial y} & \frac{\partial N_1}{\partial x} & 0 & \frac{\partial N_2}{\partial y} & \frac{\partial N_2}{\partial x} & 0 & .. & .. & \frac{\partial N_{15}}{\partial y} & \frac{\partial N_{15}}{\partial x} & 0 \\ \frac{\partial N_1}{\partial z} & 0 & \frac{\partial N_1}{\partial x} & \frac{\partial N_2}{\partial z} & 0 & \frac{\partial N_2}{\partial x} & .. & .. & \frac{\partial N_{15}}{\partial z} & 0 & \frac{\partial N_{15}}{\partial x} \\ 0 & \frac{\partial N_1}{\partial z} & \frac{\partial N_1}{\partial y} & 0 & \frac{\partial N_2}{\partial z} & \frac{\partial N_2}{\partial y} & .. & .. & 0 & \frac{\partial N_{15}}{\partial z} & \frac{\partial N_{15}}{\partial y} \end{bmatrix} \begin{bmatrix} u_{1x} \\ u_{1y} \\ u_{1z} \\ u_{2x} \\ u_{2y} \\ u_{2z} \\ : \\ : \\ u_{15z} \end{bmatrix} \tag{4.29}$$

When the strains have been determined, the stresses are obtained merely by multiplying with the constitutive matrix, **D**. This method of obtaining stresses and strains at the element integration points is purely textbook; the procedure is independent of element type. See for example [5]. It is often less convenient to have the stresses obtained at the integration points, since the stresses on the structure boundary are in many cases larger than those in its interior. Methods have been developed to extrapolate the stresses to the nodes. This matter is discussed below.

4.6.2 Extrapolating the stresses to the nodes

The Gauss points are the best sampling points when evaluating stresses in elements where numerical integration is required. It is often more useful to know the stresses at the nodes, i.e. on the element boundary, instead of in the element interior. Unfortunately, the nodes are the worst choice of sampling points. By using a quadratic least-square fit, it is possible to extrapolate the stresses, evaluated at the Gauss points, to the nodes. Chen et al [7] suggests a method for estimating interfacial stresses in laminated composites, using a least-square extrapolation method and local stress smoothing. In order to use a quadratic least-square fit, three integration points per axes direction are needed. For the 15-node wedge element, this means that 18 Gauss points are employed for the stress evaluation, instead of the 9 Gauss points used for the element stiffness integration. If $g(\xi, \eta, \zeta)$ is the assumed extrap-

olation function, and $\sigma(\xi, \eta, \zeta)$ is the discrete Gaussian stress distribution, the error at any point in the element may be written as

$$e(\xi, \eta, \zeta) = \sigma(\xi, \eta, \zeta) - g(\xi, \eta, \zeta) \tag{4.30}$$

Now, if the assumed extrapolation stress $g(\xi_j, \eta_j, \zeta_j)$ is expressed as a combination of the element shape functions $\tilde{N}_i(\xi_j, \eta_j, \zeta_j)$ and the nodal stresses $\tilde{\sigma}_i$, the problem is to find the nodal stresses that minimize the functional

$$\chi = \sum [\sigma_j(\xi_j, \eta_j, \zeta_j) - \tilde{N}_i(\xi_j, \eta_j, \zeta_j)\tilde{\sigma}_i]^2, i = 1, nnd \tag{4.31}$$

where nnd are the number of nodes, and nsp are the number of integration points.

Differentiating the functional with respect to the unknown nodal stresses $\tilde{\sigma}_i$ and setting it to zero, yields

$$\chi = \sum [\sigma_j(\xi_j, \eta_j, \zeta_j) - \tilde{N}_k(\xi_j, \eta_j, \zeta_j)\tilde{\sigma}_k][-\tilde{N}_i(\xi_j, \eta_j, \zeta_j)], i, k = 1, nnd \tag{4.32}$$

See further [7] or [8].

In matrix form, this can be written as

$$\int_V \mathbf{N^T N} dV \tilde{\sigma} = \int_V \mathbf{N^T} \sigma dV \tag{4.33}$$

where $\mathbf{N}$ are the element shape functions, $\tilde{\sigma}$ are the nodal stresses, and σ are the element stresses according to the Gaussian integration. The normal and shear stresses can be extrapolated separately, by letting $\tilde{\sigma}$ be a vector with the nodal normal stress components or the nodal shear stress components.

$$\tilde{\sigma}_{\mathbf{normal}} = \begin{bmatrix} \tilde{\sigma}^1_{xx} \\ \tilde{\sigma}^1_{yy} \\ \tilde{\sigma}^1_{zz} \\ \tilde{\sigma}^2_{xx} \\ \tilde{\sigma}^2_{yy} \\ \tilde{\sigma}^2_{zz} \\ :: \\ :: \\ \tilde{\sigma}^{15}_{zz} \end{bmatrix} \tilde{\sigma}_{\mathbf{shear}} = \begin{bmatrix} \tilde{\sigma}^1_{xy} \\ \tilde{\sigma}^1_{xz} \\ \tilde{\sigma}^1_{yz} \\ \tilde{\sigma}^2_{xy} \\ \tilde{\sigma}^2_{xz} \\ \tilde{\sigma}^2_{yz} \\ :: \\ :: \\ \tilde{\sigma}^{15}_{yz} \end{bmatrix} \tag{4.34}$$

Define the matrix $\mathbf{N}$ as

$$\mathbf{N} = \begin{bmatrix} N_1 & 0 & 0 & N_2 & 0 & 0 & .. & .. & N_{15} & 0 & 0 \\ 0 & N_1 & 0 & 0 & N_2 & 0 & .. & .. & 0 & N_{15} & 0 \\ 0 & 0 & N_1 & 0 & 0 & N_2 & .. & .. & 0 & 0 & N_{15} \end{bmatrix} \tag{4.35}$$

With isoparametric elements, Gaussian integration and the following definitions,

$$\mathbf{M} = \sum_{i=1}^{3}\sum_{j=1}^{6} \mathbf{N^T N} det\mathbf{J} wpr_i wps_j \tag{4.36}$$

$$\mathbf{P}_{normal} = \sum_{i=1}^{3}\sum_{j=1}^{6} \mathbf{N^T} \sigma^{\mathbf{normal}} det\mathbf{J} wpr_i wps_j \tag{4.37}$$

$$\mathbf{P}_{shear} = \sum_{i=1}^{3}\sum_{j=1}^{6} \mathbf{N^T} \sigma^{\mathbf{shear}} det\mathbf{J} wpr_i wps_j \tag{4.38}$$

4.33 can be evaluated as

$$\tilde{\sigma}_{\mathbf{normal}} = \mathbf{M^{-1}P}_{normal} \tag{4.39}$$

$$\tilde{\sigma}_{\mathbf{shear}} = \mathbf{M^{-1}P}_{shear} \tag{4.40}$$

$\mathbf{N}$, $\mathbf{J}$, $\sigma^{\mathbf{normal}}$ and $\sigma^{\mathbf{shear}}$ in 4.36-4.38 should be evaluated at every Gauss point with the coordinates (ξ_j, η_j, ζ_i) and the corresponding planar weights wps_j and thickness

weights wpr_i. The coordinates of the Gauss points are given in table 4.3-4.4 [6].

The procedure above is performed element by element. Since elements share nodes, several different stresses are obtained for each node. The nodal stresses obtained from each element are assembled using the topology matrix, to a global nodal stress vector. Each time a stress component is assembled to a node i in the global stress vector, a 1 is assembled to position i in another vector. When all the element nodal stresses have been calculated and assembled, the number of elements that have contributed to the total stress in each node, is known. By dividing the total stress at each node by the number of elements that have contributed, a smoothed stress value is obtained at the nodes.

If the mesh is coarse, a stress component at a certain node evaluated in one element, may differ significantly from the same stress component, at the same node, evaluated in a neighboring element. Hence, it would perhaps be more appropriate to weigh each of the stress components to the contributing element's size, i.e. letting the stress influence from a larger element be greater than that from a smaller element. But since the mesh has been generated so that all elements are small where the stress gradients are large, this should not be an issue. It is important though, that the stress smoothing is carried out separately for each material layer. Two material layers may share a node, and the strains in this node are the same for both materials. The stresses, though, are discontinuous.

When the smoothed stress components have been obtained at all nodes, the principal stresses at each node may be calculated. This is performed by setting up the stress tensor $\mathbf{S}$ for each node, and determining its eigenvalues. The directions of the principal stresses are the corresponding eigenvectors. They are not of interest here.

$$\mathbf{S} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{bmatrix} \tag{4.41}$$

In order to determine the eigenvalues of $\mathbf{S}$, the functions *dsytrd* and *dsteqr* from the *Intel Math Kernel Library* [11] were used. This library also contains the functions $dgetrf$ and *dgetri* that were used for inverting the matrix $\mathbf{M}$ in Equations 4.39-4.40. The nodal stress components σ_{ij}, as well as every node's largest principal stress, are written to files that can be read by the post-processor, discussed in Section 4.9.

Table 4.3: *Location of Gauss points in the ξ, η-plane, with the planar Gauss weight wpi*

j	ξ_j	η_j	wps_j
1	0.09157621350977	0.09157621350977	0.05497587182766
2	0.44594849091597	0.10810301816807	0.11169079483901
3	0.81684757298046	0.09157621350977	0.05497587182766
4	0.10810301816807	0.44594849091597	0.11169079483901
5	0.44594849091597	0.44594849091597	0.11169079483901
6	0.09157621350977	0.81684757298046	0.05497587182766

Table 4.4: *Location of Gauss points in the ζ-direction, with the corresponding Gauss weight wpj*

i	ζ_i	Weight wpr_i
1	-0.77459666924148	0.55555555555556
2	0	0.88888888888889
3	0.77459666924148	0.55555555555556

4.7 Modelling of Bolt Fixings

4.7.1 General remarks

As discussed in Chapter 1, two distinctive type of bolt fixings for laminated glass panes were analyzed in an earlier master's thesis [1], namely the cylindrical bolt and the countersunk bolt. From a finite element modelling perspective, they both have their advantages and disadvantages. The countersunk bolt does not involve materials that behave non-linearly to a great extent, which constitutes a simpler constitutive model. The shape of the hole, though, makes the meshing procedure perplexing. The meshing methods used in this project, as discussed in Section 4.4, does not allow for these kinds of holes without special consideration. The cylindrical bolt, on the other hand, does not require any special treatment with respect to meshing. The difficulty lies in one of its components, rubber, which with its incompressibility and non-linear behavior necessitates special consideration.

Clearly, none of the two bolts are straight forward to model, and due to lack of time one of them has to be prioritized. Since the cylindrical bolt can be used together with the available meshing facilities, focus has been put on that type. The scope of this project does not allow for developing a tool that determines stresses and strains with perfect accuracy. Such a tool would require non-linear material

models, and would hence require an iterative solution procedure. This would not only imply longer solution times annoying for the user, but it would also require more work. An effort has been made to model the approximate behavior, based on simplifications believed decently justified.

4.7.2 The cylindrical bolt

The cylindrical bolt implemented in ClearSight, requires a cylindrical hole of diameter $d = 30mm$. The rubber ring that is fastened to the glass by means of the metal bolt, thereby forming the friction joint, has an outer diameter of $d = 50mm$. The bolt is tightened with a torque $T = 15 - 20\ Nm$, which squeezes the rubber and causes it to flatten out. When the bolt is fixed to a glass pane that is subjected to bending, a part of the rubber will be squeezed extensively on one side of the glass pane, whereas the rubber may actually lose contact with the glass on the other side of the bolt, on the same side of the glass pane. On the other side of the glass pane, the behavior is exactly the opposite.

In an attempt simulate this behavior by means of a rather simple finite element model, two computations are carried out for each model. In both computations, the actual bolt is excluded from the model. In the first computation, the nodes on the glass area facing the rubber ring are completely locked in all directions. These nodes are found by means of the element attribute assigned to the region when creating the poly-file, as discussed in Subsection 4.4.3. When the reaction forces have been obtained, the nodes subjected to tensile reaction forces are released. This corresponds well with the bolt fixing being unable to transfer tensile forces. The boundary conditions used in the second computation are hence based on the results of the first computation, enabling the nodes that are not subjected to compressive reaction forces to displace.

4.8 Solving the Global System of Equations

4.8.1 Introduction

Solving large systems of equations is a problem to which there are several different approaches, suitable for different kinds of problem types. Two major branches of solution methods are the direct methods and the iterative methods. In this project, a direct solver - *PARDISO* from the *Intel Math Kernel Library* [11] - was eventually chosen. An iterative solver - *PGMRES* from *SPARSKIT* [9] - was also tested. The latter did not converge when using solid elements, but worked fine with the shell elements. To introduce any of those, it is necessary to shortly present some theory from linear algebra. Note that the following sections are intended as brief introductions only; see the references at each section for further reading.

When dealing with large finite element problems it is often not feasible to store the full stiffness matrix. The stiffness matrix of a typical finite element problem is always *sparse*, i.e. most of the matrix elements are zeroes. The size of the stiffness matrix, in terms of number of matrix elements, increases very fast when increasing the number of degrees of freedom *(DOFs)*. By going from *(n)* to *(n+1)* DOFs, the number of matrix elements has increased by *(2n+1)*. Even when actually possible, storing the full stiffness matrix would, because of its inherently sparse nature, implicate an unnecessary waste of computer memory. Therefore, one resorts to sparse matrix storage, which means that all the zeroes are omitted and only non-zero indices are stored. When solving the global system of equations, it is therefore required that the solver is a so called sparse-solver.

There are several different methods to apply the sparse storage, using various formats with different advantages. The matrix *number of non-zeroes (NNZ)* is an important property that for a given format conveniently indicates the computer storage cost. Figure 4.20 shows the sparsity pattern of a stiffness matrix typically dealt with in this project. In this particular example, more than 99.5 per cent of the indices contain zeroes. Neglecting to explicitly store all those zeroes, consumes a lot less memory than storing the full matrix.

4.8.2 Coordinate Sparse Format

The simplest format in which to store sparse matrices is called Coordinate Sparse Format *(COO)*, where three arrays are used to describe the matrix **A** of size *(n*m)*.

One array, **AS**, containing the non-zero values of the matrix **A**. The length of **AS** is *NNZ*.

Two integer arrays, **AI** and **AJ**, contain the row and column indices of the corresponding element in **A**. **AI** and **AJ** both have the length *NNZ*.

See the example below. An asterisk (*) in A represents a zero index, i.e. an index that is not stored.

$$A = \begin{bmatrix} 5 & * & 2 & * & * \\ * & 3 & * & 8 & * \\ 2 & * & 1 & 5 & 6 \\ * & 8 & 5 & 1 & * \\ * & * & 6 & * & 5 \end{bmatrix} \Leftrightarrow AS = \begin{bmatrix} 5 \\ 2 \\ 3 \\ 8 \\ 2 \\ 1 \\ 5 \\ 6 \\ 8 \\ 5 \\ 1 \\ 6 \\ 5 \end{bmatrix}, AI = \begin{bmatrix} 1 \\ 3 \\ 2 \\ 4 \\ 1 \\ 3 \\ 4 \\ 5 \\ 2 \\ 3 \\ 4 \\ 3 \\ 5 \end{bmatrix}, AJ = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 2 \\ 3 \\ 3 \\ 3 \\ 3 \\ 4 \\ 4 \\ 4 \\ 5 \\ 5 \end{bmatrix} \tag{4.42}$$

It is obvious that storing a dense matrix, where the number of non-zeroes is large compared to the matrix size *(m*n)*, in this format is not beneficial since three arrays of size *NNZ* must be allocated. For dense matrices, *3*NNZ* is larger than *m*n*.

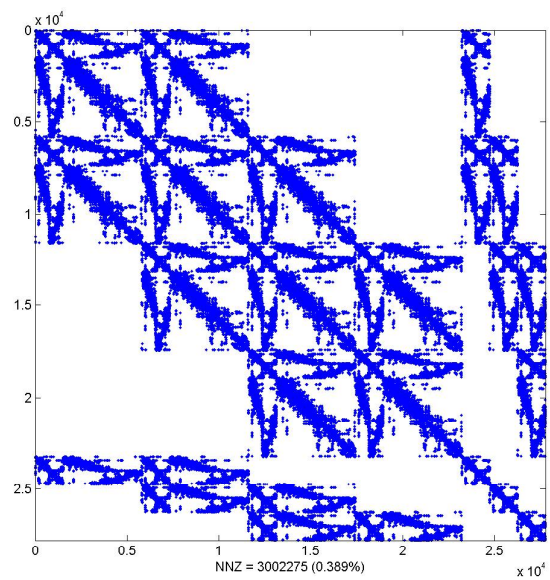


Figure 4.20: *Typical sparsity pattern of a FE stiffness matrix. Only 0.389 per cent of the indices contain non-zeroes.*

An advantage of this format, besides memory savings in cases of sparse matrices, is that it is rather straight forward to assemble a full matrix in to this format. When dealing with very large matrices, though, it might be worth while to use a format that is even cheaper in terms of memory usage.

One drawback of using any sparse format is that simple matrix operations such as addition and multiplication are made a bit trickier. Luckily, the excellent package *SPARSKIT* by Yousef Saad [9], contains heaps of routines for performing simple and advanced matrix operations and manipulations on sparse matrices, as well as routines for converting matrices between different sparse formats.

4.8.3 Compressed Sparse Row Format

An even more efficient sparse format is the *Compressed Sparse Row format (CSR)*, where the matrix **A** of size *(n*m)* is again described by three arrays.

An array, **AS**, contains the non-zero values of the matrix **A**, stored row by row. The length of **AS** is *NNZ*.

An integer array, **AI**, contains the column indices of the corresponding element in **AS**. The length of **AI** is *NNZ*.

A second integer array, **AJ**, contains row pointers. The pointers indicate at which element in **AS** and **AI** every new row in **A** begins. The length of **AJ** is *(n+1)*, where the position *(n+1)* points to a fictitious element in **AS** and **AI** .

See the example below.

$$A = \begin{bmatrix} 5 & * & 2 & * & * \\ * & 3 & * & 8 & * \\ 2 & * & 1 & 5 & 6 \\ * & 8 & 5 & 1 & * \\ * & * & 6 & * & 5 \end{bmatrix} \Leftrightarrow AS = \begin{bmatrix} 5 \\ 2 \\ 3 \\ 8 \\ 2 \\ 1 \\ 5 \\ 6 \\ 8 \\ 5 \\ 1 \\ 6 \\ 5 \end{bmatrix}, AI = \begin{bmatrix} 1 \\ 3 \\ 2 \\ 5 \\ 1 \\ 3 \\ 4 \\ 5 \\ 2 \\ 3 \\ 4 \\ 3 \\ 5 \end{bmatrix}, AJ = \begin{bmatrix} 1 \\ 3 \\ 5 \\ 9 \\ 12 \\ 14 \end{bmatrix} \tag{4.43}$$

For very large, sparse matrices the Compressed Sparse Row format is preferable, since a considerable amount of computer memory may be saved. In this project

though, the strongest reason for using the CSR format is that some of the employed routines require matrix inputs in this format.

4.8.4 Assembling element stiffness matrices

When assembling element stiffness matrices into a global stiffness matrix, a simple approach is to first employ a variant of the COO format. One float array (**KS**) and two integer arrays (**KI**,**KJ**) of size $Nel * ElDof^2$ are allocated, where Nel are the total number of finite elements, and $ElDof$ are the number of degrees of freedom per finite element. Every non-zero index in the element stiffness matrix is added row by row to the array **KS**, with its row and column indices according to the topology stored in **KI** and **KJ** respectively.

The global stiffness matrix, described by the three arrays **KS**, **KI** and **KJ**, now has indices that occur several times due to the connectivity between nodes. By summing up the number of indices that occur more than once, and subtracting those from the length of the array **KS**, the number of non-zeroes in the matrix has been obtained. Three new vectors may now be allocated, exactly according to the description of *COO* format above. If the duplicate indices are merged, and all the elements are ordered row by row, the result obtained is exactly that described by the *COO* format. It is now a rather simple task to convert the matrix to *CSR* format. SPARSKIT [9] contains routines for performing both the merge and the conversion. This way of assembling the global stiffness matrix is certainly not the most memory efficient, since the first three vectors allocated have the size $Nel * ElDof^2$, which is often a lot larger than the number of non-zeroes in the final stiffness matrix. Its advantage is that it is probably the simplest way.

The amount of memory saved by going from *COO* format to *CSR* format is often not that significant, but since the employed linear equation system solver accepts matrices stored in *CSR* format only, the conversion is well motivated.

4.8.5 Matrix partitioning due to prescribed DOFs

The system of equations that need to be solved in a linear finite element problem has the form

$$\mathbf{Ka} = \mathbf{f} \tag{4.44}$$

where $\mathbf{K}$ is the global stiffness matrix, $\mathbf{a}$ is the displacement vector and $\mathbf{f}$ is the load vector. Since this system contains both known (prescribed) displacements, a^*, and unknown displacements, $\tilde{a}$, the first thing that needs to be done is to partition the system so that the displacement vector contains unknown displacements only. Consider the system of Equations in 4.45. The system has $ndof$ equations, with m

prescribed displacements.

$$\begin{bmatrix} & & \\ & \mathbf{K} & \\ & & \end{bmatrix} \begin{bmatrix} a_1^* \\ \tilde{a}_2 \\ a_3^* \\ \tilde{a}_4 \\ : \\ \tilde{a}_{ndof-1} \\ \tilde{a}_{ndof} \end{bmatrix} = \begin{bmatrix} 0 \\ \tilde{f}_2 \\ 0 \\ \tilde{f}_4 \\ : \\ \tilde{f}_{ndof-1} \\ \tilde{f}_{ndof} \end{bmatrix} \tag{4.45}$$

The m known displacements are permuted, so that they appear last in the displacement vector. The rows and columns of $\mathbf{K}$ are permuted accordingly, so that the resulting system takes the form of Equation 4.46, where $\mathbf{K}$ has been partitioned into submatrices.

$$\begin{bmatrix} \mathbf{K_{aa}} & \mathbf{K_{ab}^*} \\ \mathbf{K_{ba}} & \mathbf{K_{bb}^*} \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{a}} \\ \mathbf{a}^* \end{bmatrix} = \begin{bmatrix} \tilde{\mathbf{f}} \\ \mathbf{f}^* = \mathbf{0} \end{bmatrix} \tag{4.46}$$

The last m columns of the first $ndof - m$ equations in $\mathbf{K}$ can now be multiplied with the m known displacements, yielding additional forces to the first $ndof - m$ equations. The submatrix K_{aa} is extracted, yielding the following system of equations,

$$\mathbf{K_{aa}}\tilde{\mathbf{a}} = \tilde{\mathbf{f}} - \mathbf{K_{ab}^*}\mathbf{a}^* \tag{4.47}$$

which is a system of *(ndof-m)* equations, where everything is fully known but $\tilde{\mathbf{a}}$, which is fully unknown. The original system in Equation 4.45, involving both known and unknown displacements, has thus been reduced to 4.47, where the left hand vector contains unknown displacements only. The problem is now reduced to a linear system of equations, $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ and $\mathbf{b}$ are fully known, whereas $\mathbf{b}$ is fully unknown. Solving the latter is a classical Linear algebra problem, to which there are several solution methods suitable for computer implementation.

4.8.6 LU- and Cholesky factorization

Consider the system of linear equations

$$\mathbf{Ax} = \mathbf{b} \tag{4.48}$$

where $\mathbf{A}$ is a quadratic, real matrix of size *(m*m)*, and where $\mathbf{x}$ and $\mathbf{b}$ are vectors of length m. $\mathbf{A}$ and $\mathbf{b}$ are both known, whereas $\mathbf{x}$ is to be determined. Finding the inverse of the matrix $\mathbf{A}$, thus obtaining the solution as $\mathbf{x} = \mathbf{A^{-1}b}$, is often hard and inefficient. The idea of LU factorization is to decompose the matrix $\mathbf{A}$ into a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$, with the property $\mathbf{LU} = \mathbf{A}$. A lower triangular matrix is a matrix in which all the non-zeroes are located below the main diagonal. Similarly, an upper triangular matrix is a matrix in which all the non-zeroes are located above the main diagonal. With $\mathbf{A} = \mathbf{LU}$, Equation 4.48 now takes the form

$$\mathbf{LUx} = \mathbf{b} \tag{4.49}$$

which with

$$\mathbf{Ux} = \mathbf{y} \tag{4.50}$$

takes the form

$$\mathbf{Ly} = \mathbf{b} \tag{4.51}$$

Since $\mathbf{L}$ is lower triangular, the unknown vector $\mathbf{y}$ can easily be determined by forward substitution. Finally, the vector $\mathbf{x}$ can be obtained by backward substitution from Equation 4.50. Solving the Equations 4.50 and 4.51 is trivial, and the problem lies in finding the matrices $\mathbf{L}$ and $\mathbf{U}$. This can be performed by means of Gaussian Elimination.

If the matrix $\mathbf{A}$ is symmetric, i.e A(i,j) = A(j,i), and positive definite, i.e. $\mathbf{x^T Ax} > \mathbf{0}$ for any vector $\mathbf{x} \neq 0$, $\mathbf{A}$ can be factored as $\mathbf{LL^T} = \mathbf{A}$, where $\mathbf{L}$ is lower triangular. This factorization is called the Cholesky factorization. From a storing efficiency viewpoint, the benefit of the Cholesky factorization is apparent, since only one matrix $\mathbf{L}$ needs to be stored, as opposed to two matrices $\mathbf{L}$ and $\mathbf{U}$ in the LU decomposition. Some solvers, such as the *PARDISO* solver from the Intel Math Kernel Library [11] employed in this project, exploit this. When the matrix has been factored, i.e. when the matrix $\mathbf{L}$ has been determined, the same back- and forward substitution strategy as for LU is used. The algorithm for performing the Cholesky factorization can be found in for example [12].

An important concept when dealing with LU and Cholesky decompositions of sparse matrices is the *fill-in*. A fill-in represents a non-zero element in the matrix **L** or **U** at a position (i, j) that in the matrix **A** contains a zero. A larger number of fill-ins results in more computational work, as well as higher memory consumption. The number of fill-ins can be significantly reduced by permuting the rows and columns of **A** according to some fill-reducing scheme as discussed later on. The example below, taken from the Intel Math Kernel Library manual [11], illustrates the concept of fill-ins as well as the effect of row and column permutation.

Consider the system of equations

$$\begin{bmatrix} 9 & \frac{3}{2} & 6 & \frac{3}{4} & 3 \\ \frac{3}{2} & \frac{1}{2} & * & * & * \\ 6 & * & 12 & * & * \\ \frac{3}{4} & * & * & \frac{5}{8} & * \\ 3 & * & * & * & 16 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \tag{4.52}$$

where **A** is symmetric and positive definite. An asterisk (*) in **A** represents a zero index. Obviously, **A** contains 12 zeroes and 13 non-zeroes. The Cholesky factorization of **A**, i.e. $\mathbf{A} = \mathbf{L}\mathbf{L}^{\mathbf{T}}$, is

$$L = \begin{bmatrix} 3 & * & * & * & * \\ \frac{1}{2} & \frac{1}{2} & * & * & * \\ 2 & -2 & 2 & * & * \\ \frac{1}{4} & -\frac{1}{4} & -\frac{1}{2} & \frac{1}{2} & * \\ 1 & -1 & -2 & -3 & 1 \end{bmatrix} \tag{4.53}$$

Even though **A** is rather sparse, **L** does not contain any non-zeroes below the main diagonal. Now, consider the effect of letting x_1 and x_5 switch places in the system 4.52. The rows and columns of **A** are permuted accordingly, i.e. the first and fifth rows switch places, as well as the first and fifth columns.

$$\begin{bmatrix} 16 & * & * & * & 3 \\ * & \frac{1}{2} & * & * & \frac{3}{2} \\ * & * & 12 & * & 6 \\ * & * & * & \frac{5}{8} & \frac{3}{4} \\ 3 & \frac{3}{2} & 6 & \frac{3}{4} & 9 \end{bmatrix} \begin{bmatrix} x_5 \\ x_2 \\ x_3 \\ x_4 \\ x_1 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \\ 3 \\ 4 \\ 1 \end{bmatrix} \tag{4.54}$$

If the new matrix is called $\hat{\mathbf{A}}$, then the corresponding Cholesky factorization yields

$$\hat{L} = \begin{bmatrix} 4 & * & * & * & * \\ * & \frac{1}{\sqrt{2}} & * & * & * \\ * & * & 2(\sqrt{3}) & * & * \\ * & * & * & \frac{\sqrt{10}}{4} & * \\ \frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{\frac{3}{5}}}{4} \end{bmatrix} \tag{4.55}$$

$\hat{\mathbf{L}}$ obviously has a lower level of fill-in than do $\mathbf{L}$, due to the permutation. For small systems like the one in the example above, the effect on solve time and memory savings is insignificant, but for larger system such permutations may have a great effect.

4.8.7 Nodal reordering

As mentioned, the number of fill-ins in the LU- or Cholesky decomposition of a matrix $\mathbf{A}$ can be immensely reduced by permuting its rows and columns. Permuting the rows and columns of the stiffness matrix in a finite element problem, is equivalent to relabelling the degrees of freedom in the element mesh. How the DOFs are labelled determines the sparsity pattern of the stiffness matrix, and this pattern has a huge effect on how fast the system of equations may be solved, given a specific solving routine. Therefore, several schemes have been developed that relabel the nodes with the objective of obtaining some desirable property of the sparsity pattern.

For banded solvers, the desirable property of the matrix structure is a small bandwidth, i.e all the non-zeros are located near the main diagonal. The Reverse Cuthill-McKee algorithm is probably the best renown scheme for this purpose.

When using a general sparse LU- or Cholesky solver, the desirable property is that

of a low level of fill-in, as discussed in Subsection 4.8.6. In order to obtain this low fill property, the *PARDISO* solver employs a scheme called *the Nested Dissection Method* [20] that permutes the rows and columns of the stiffness matrix, as well as the corresponding indices in the right hand vector. As discussed above, this permutation can be interpreted as a reordering of nodes. When using *PARDISO*, though, one never notices this permutation. Since its sole purpose is to speed up the solve procedure, the solution vector is automatically back permuted to correspond to the original nodal labelling, once the solution is obtained.

4.8.8 An iterative solver

Before finally settling with the direct solver *PARDISO* [11], an iterative sparse solver from SPARSKIT [9] was tested. This iterative solver employs a Krylov subspace method called *PGMRES*, with a preconditioning technique called *Dual Threshold Incomplete LU factorization (ILUT)*, see further [10]. The theory of this iterative solver is far more complex than that for the direct solver, and it is therefore not described here at all. See [10] for details.

The *PGMRES* solver converged rather quickly when using shell elements. This solver requires that an (incomplete) LU factorization be made of the matrix, where the number of fill-ins in the **L** and **U** factors have a very significant effect on solve-time and convergence. In order to reduce the number of fill-ins, a nodal reordering scheme was tested. This is exactly the same reordering scheme used by the direct solver, i.e. the nested dissection method. The algorithm is implemented in an application suite called *METIS*, see [20], which was used to reorder the nodes. For the shell elements, the reordering reduced the solve-time significantly, as well as the memory use. For the wedge elements though, the iterative solver did not converge at all, not even with the nodal reordering. This might be due to the higher nodal connectivity, resulting in a less sparse system matrix. It is possible that convergence can be obtained by tweaking some input parameters to the *ILUT*- and *PGMRES* routines, but the direct solver was finally used since it does not require any such tweaking of parameters to obtain speed and stability. It is also much more convenient to have the solver do all the steps automatically, from nodal relabelling to factorization and final solving.

MATLAB also uses a direct method for solving linear systems of equations where the matrix is symmetric and positive definite.

4.9 Visualization of Results

4.9.1 Used tools

In order to visualize the results in a nice and illustrative manner, a simple post-processor has been developed. The post-processor uses the graphics library OpenGL. OpenGL is the most widely used 2D and 3D graphics application programming interface (API), providing hundreds of routines for drawing complex scenes from geometric primitives, performing modelling transformations, adding lighting and textures, etc. [17]. GLU (OpenGL Utility library) is distributed with OpenGL and consists of a number of higher-level drawing routines that uses the more primitive OpenGL routines for certain features such as positioning the camera, etc. Neither OpenGL nor GLU contains routines for defining a window in which to draw, and therefore GLUT (OpenGL Utility Toolkit) was used for this purpose. GLUT also contains functions for monitoring mouse and keyboard events and creating simple pop-up menus. The Python OpenGL binding PyOpenGL [18] was used as well as PyGLUT [19], so that all the code for the post-processor could be written in Python.

4.9.2 Drawing nodal stresses

It would certainly be possible to draw a scene containing the analyzed glass pane fully in 3D, but for the purpose of visualizing the stresses in different sections along the thickness direction of the glass pane, it is easier to just draw the section of interest. Hence, a function has been included in the post-processor that allows the user to specify what section to view, and this section is then drawn in 3D, but without a thickness. The post-processor allows for viewing four different sections in a laminated glass pane, irrespective of how many finite element-layers are used for each material. These sections are shown in Figure 4.21. For the sections shared by PVB and glass, stresses in both materials can be viewed. For a single-layered glass pane, only stresses on the free surfaces of the glass can be viewed.

As mentioned, OpenGL contains routines for drawing primitives, and one of those primitives is a 3-node triangle. The triangle can be drawn in 3D, i.e. every node of the triangle is given an (x, y, z)-coordinate. Every corner is also assigned a color, and OpenGL will then smooth the nodal colors over the whole triangle as shown in Figure 4.22. This feature is used when drawing the nodal stresses that were obtained according to Subsection 4.6.2. The triangular top and bottom surface of the wedge element has 6 nodes, but there is no such primitive in OpenGL. Instead, every 6-node triangle is divided into four triangles according to Figure 4.23, and hence the stresses at the midpoint nodes can be taken into account as well.

Every color in OpenGL is specified as a mixture of three base colors; red, green and blue (RGB). Every base color enters the mix-color with a value from 0.0 to 1.0, so the color $(1.0, 0.0, 0.0)$ represents red, $(0.0, 1.0, 0.0)$ represents green and $(0.0, 0.0, 1.0)$ represents blue, [13]. In order to map a certain nodal stress into a

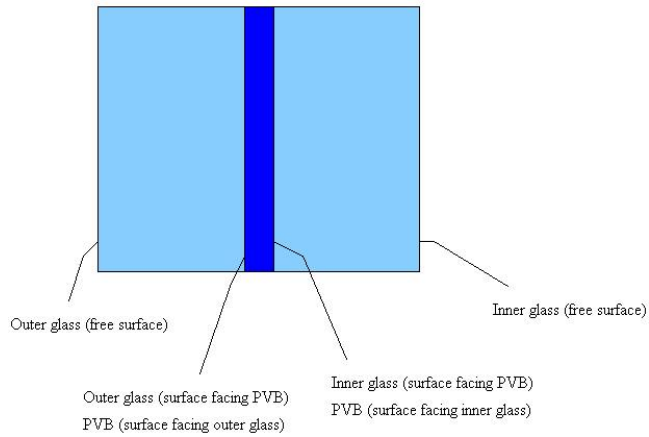


Figure 4.21: *Figure indicating which sections of a laminated glass pane can be viewed in the post-processor.*

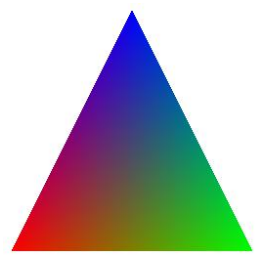


Figure 4.22: *A triangle with the colors red, green and blue assigned to each of the three corners. OpenGL smoothes the colors over the triangle.*

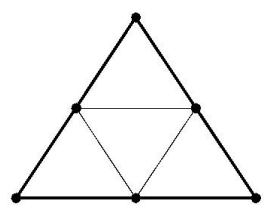


Figure 4.23: *A 6-node triangle divided into four 3-node triangles.*

Figure 4.24: *The color scale used for mapping stresses to colors.*

certain color, use was taken of the function JET in MATLAB [14]. This function produces a list of vectors, a colortable, of which each vector represents an RGB color. The first vector in the list represents blue, the last vector represents red, and the vector in the middle of the list represents green. All the other vectors are linear interpolations of these colors, so that a smooth colortable is achieved. An arbitrary number of vectors can be retrieved by the function JET; a larger number of vectors gives a smoother colortable. A colortable comprising 64 colors, i.e. 64 RGB-vectors, is used in the post-processor. These colors are shown in Figure 4.24.

When the user chooses to view a certain stress component, all the nodal stresses of this component are read from a text file. The maximum and minimum nodal stresses, together with the colortable, define the color mapping. A certain nodal stress S is mapped to its corresponding color by

$$Max\ absolute\ stress = max(|max\ nodal\ stress|, |min\ nodal\ stress|)$$

$$Normalized\ stress = \frac{S + (Max\ absolute\ stress)}{2(Max\ absolute\ stress)}$$

$$Index = integer(63(Normalized\ stress)) + 1$$

where $Index$ is an integer $1 - 64$ that indicates which RGB vector in the colortable that corresponds to the nodal stress S. With this mapping, a zero stress will always be drawn with green color, which feels intuitive. There is also an option that allows the user to view the dangerous stresses, i.e. stresses exceeding the maximum allowed stress that was entered in the GUI. All nodes where the maximum principal stress exceeds the allowed value are drawn with a red color, whereas all the other nodes are drawn green.

In order to let the user choose what stress component and section to view, a simple pop-up menu was created using GLUT. Tools for moving around the scene, i.e. pan, rotate and zoom, using the mouse are also accessible from this menu, which appears on a right-click with the mouse. When one of these tools are selected, GLUT monitors the mouse movement. These mouse movements are then mapped into proper changes to the transformation matrix that determines what to draw on the screen. Figure 4.25 shows the menu.

Nodal coordinates and displacements are read from text files produced by the Fortran code. When a section is chosen by the user, only nodes in that section are drawn. The pop-up menu also allows the user to choose a deformation scaling, so that the displacements can be exaggerated and the deformed shape of the structure becomes clear. In Chapter 5 a few examples are shown.

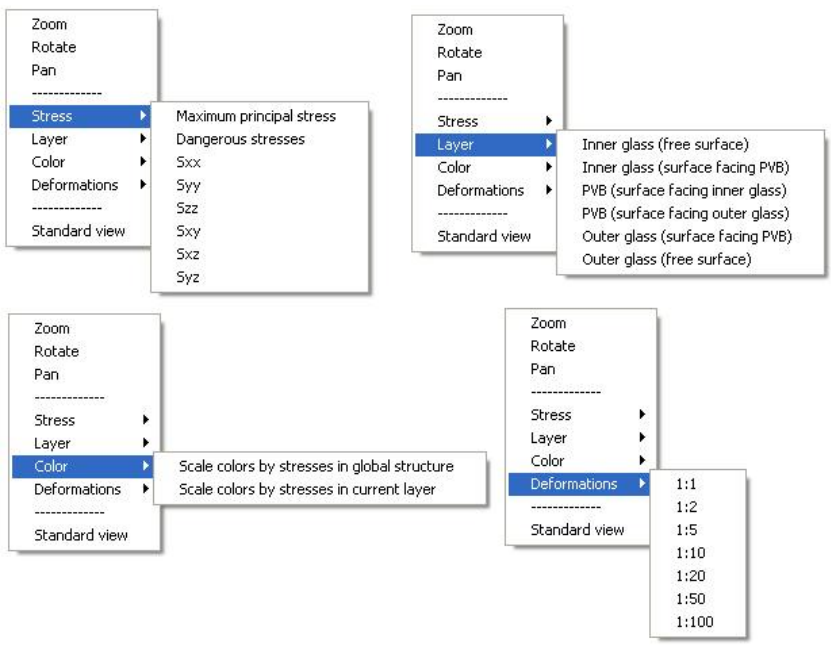


Figure 4.25: *The pop-up menu*

Chapter 5

Examples

5.1 General

In this chapter a few examples solved with the program developed in this project, are shown. The simple post-processor developed in this project, described in Section 4.9, is used for visualizing the results.

5.2 Balustrade

Figures 5.1-5.2 show stresses in a $2 * 1.2m$ large balustrade, composed of two $8mm$ glass panes with a $0.76mm$ thick intermediate foil of PVB. The material parameters are set according to Equation 2.1. The two bolts are located $200mm$ from the right and left edge, respectively, $200mm$ above the lower edge. The balustrade is subjected to a line load $Q = 800\ N/m$ along the top edge.

Two elements were used in the thickness direction for each of the glass layers, whereas merely one was used for the PVB foil. Each hole comprises 30 piecewise linear segments, i.e. 30 vertices were used in the poly-file. The model contains 7230 elements, and 66030 degrees of freedom.

The maximum principal stresses emerge on the inner free glass surface, from where the load is applied.

The figure of interest for someone using ClearSight as a design tool would be Figure 5.2b, where regions with principal stresses exceeding the permissible stress are drawn in red color.

5.3 Facade

Figure 5.3 shows the maximum principal stresses in a $1.4 * 1.4m$ large facade, fixed with four bolts. The facade is composed of two $8mm$ glass panes with a $0.76mm$ thick intermediate foil of PVB. The material parameters are set according to Equation 2.1. Each bolt is located $50mm$ from the edges. The entire facade is subjected to a wind load $q = 1000\ N/m^2$.

As in the balustrade example above, two elements were used in the thickness direction for each of the glass layers, and one for the PVB foil.

The model contains 9230 elements, and 86373 degrees of freedom. The maximum principal stresses emerge on the outer free glass surface, from where the load is applied. Figure 5.4a show regions where the permissible stress is exceeded, drawn in red.

Figure 5.4b shows the maximum principal stresses in a similar facade, where the bolts have been moved further away from the edges, more specifically $200mm$ away from the edges. This model contains 12155 elements, and 110520 degrees of freedom. The mesh was generated with the same meshing properties as the former example; the reason this model contains more elements and degrees of freedom is that the holes are located further away from the edges, so a larger region is refined. The maximum principle stresses are considerably smaller, but yet they exceed the permissible stress 50 MPa.

5.4 Discussion

Since all the nodes inside the area facing the bolt's rubber ring are locked in the first computation, the results are marred by some errors. The locked area form a rigid surface that does not arise in reality. This prevents the maximum stresses from emerging directly around the hole edges, where they in reality do emerge [1]. See for example Figure 5.3b.

The bolt is actually a friction joint, and hence friction forces will act over the ring to different degrees. The method used here, where nodes are locked, yields very high stress concentrations. A more appropriate method for modelling the bolt fixings, still using linear-elastic theory, is suggested in Chapter 6. To gain really accurate results though, non-linear material models are required.

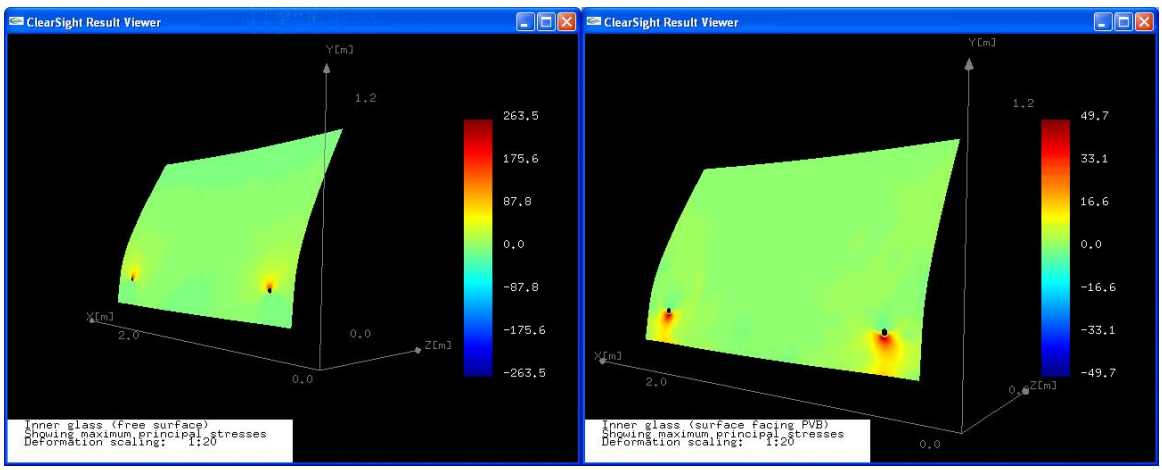


Figure 5.1: *The left figure shows the maximum principal stresses on the inner free glass surface. The right figure shows the maximum principal stresses on the inner glass surface facing the PVB foil.*

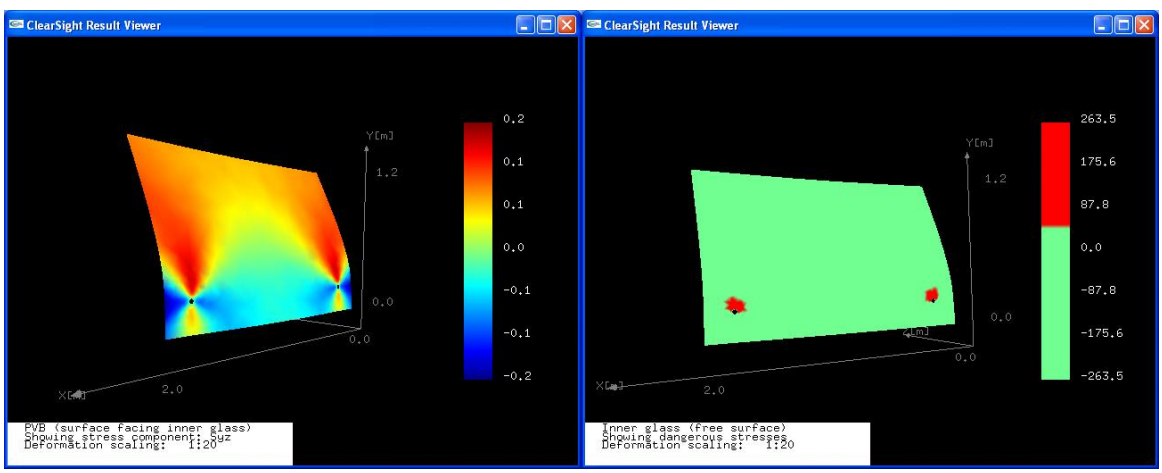


Figure 5.2: *The left figure shows the shear stress component σ_{yz} in the PVB foil. In the right figure, principal stresses exceeding $50MPa$ on the inner free glass surface are drawn in red color.*

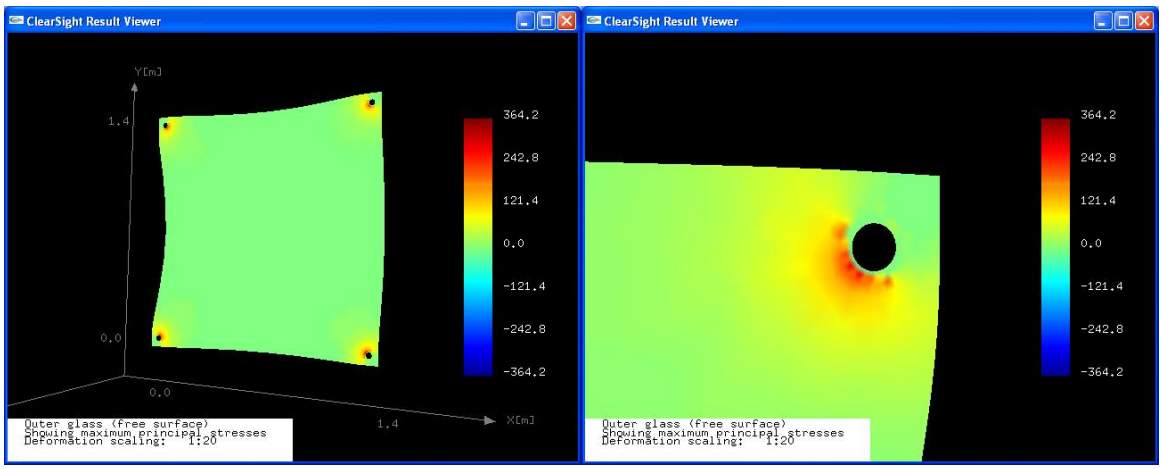


Figure 5.3: *Maximum principal stresses in a point-fixed facade subjected to a wind load. The right figure shows a close-up of one of the holes.*

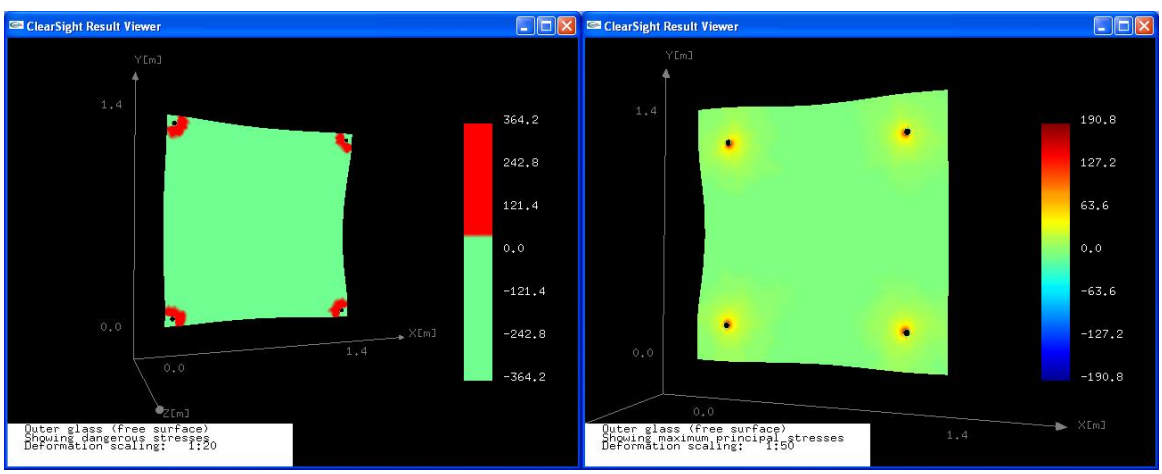


Figure 5.4: *The left figure shows principal stresses exceeding* $50MPa$ *drawn in red color. The right figure shows the maximum principal stresses in a similar facade, where the bolts have been moved further from the edges.*

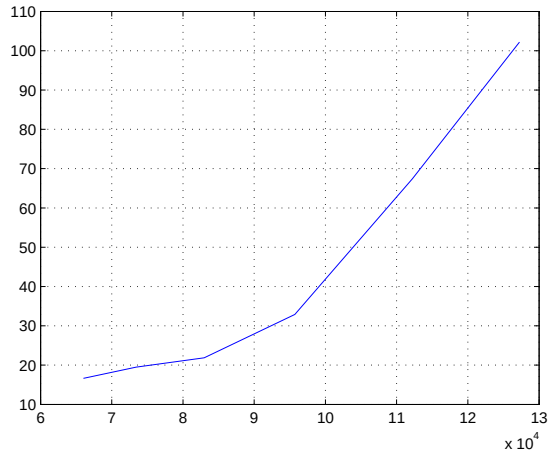


Figure 5.5: *The solve time increases quickly with increasing number of DOFs*

5.5 Solve time

Figure 5.5 shows how the time for solving a linear system of equations increases with increasing number of degrees of freedom, using the direct solver PARDISO [11]. A *Dell Optiplex GX620* with a 3.00GHz Pentium 4 processor and 1Gb of memory was used.

Solving the global system of equations is by far the most time-consuming procedure. Calculating the element stiffness matrices and evaluating the element stresses goes relatively quickly.

Chapter 6

Concluding Remarks

6.1 General

In this master's project, a finite element-based program for simulating the behavior of point-fixed laminated glass panes has been developed. In an earlier master's thesis [1], 20-node hexagonal elements were used for simulating a few experimental set-ups concerning bolt-fixed laminated glass. These simulations yielded rather accurate results. The 15-node wedge elements used in this project converge to the same solution, but more elements are required.

For large structures the system of equations gets very large. The solve-time grows super-linearly, which means that doubling the number of DOFs more than doubles the solve-time. For linear analysis, this is only a minor problem since the solve time stays at a few minutes. Elements located far away from the bolts have a rather large top and bottom surface area, which is unadvantageous due to the modest element thickness. But since the stress gradients are very small at these locations, the error stays small. The mesh is generated so that the elements near the bolts, where the stress gradients are high, have small surface areas.

A simple post-processor was developed employing the graphics library OpenGL. This enables the program to draw different stress components and displacements in a very illustrative manner. The feature where areas with principal stresses exceeding the permissible stress are drawn red, ought to be very useful for persons using ClearSight as a design tool.

All computational routines written in this project, have been verified to ABAQUS to ensure correct implementations. The technique used for modelling the bolt-fixings do not seem to yield accurate results. It is not adequate to lock nodes completely in order to simulate the behavior of a friction joint. The stresses obtained with the program are overestimated, due to the node locking. A more appropriate, but still rather simple, approach for modelling the bolts is proposed in section 6.2. When a proper technique for modelling the bolts has been established, ClearSight will con-

stitute a very powerful tool for designing point-fixed glass balustrades and facades.

There are several things that would improve the performance and the accuracy of the computations. Some of those would not require too much effort, while others would require some work. Most of the routines written could and should certainly be re-used with little or no changes when the tool is further developed.

6.2 Proposals For Future Work

- The boundary conditions used in order to simulate the bolt fixings have to be developed further. In this work, the nodes facing the cylindrical bolt's rubber plate are completely locked which results in unrealistical stress concentrations around the bolts. A more appropriate approach would be to use elastic fixings, where every node facing the bolt's rubber ring is connected to three spring elements; two shear springs and one normal spring. All springs connected to a node where the normal spring is subjected to tension, would be released in order to simulate the behavior of the bolt as discussed in Section 4.7. This feature would require some testing of the rubber to determine proper stiffness of the springs in different directions. When a proper spring stiffness has been determined, the actual implementation would be simple. A more accurate stress distribution around the bolts would probably be obtained. Possibilities to analyze glass panes with other fixings would also be very useful.

- At this point, there is no consideration of structural symmetries. Most balustrades are supposedly symmetrical, and hence it would be very advantageous to exploit this fact in the calculations. This would shorten the solve time significantly, since the system would only need to contain half the amount of degrees of freedom. For facades, two symmetry lines could probably be used in most cases, and the computational savings would be immense. Adding this kind of feature would not require much work.

- The 15-node wedge element is not as accurate as the hexagonal element. The sole reason the wedge element prevails in this project is due to the fact that Triangle, yielding triangular elements only, was the only free mesh generator found. If a free quadrilateral/hexagonal mesh generator emerges, this would be preferable, provided that similar functions for identifying nodes and elements by means of boundary markers and element attributes, also be included.

- Further, it could be useful if the tool could handle more complex geometries. To preserve the user-friendliness, a number of geometry templates could be implemented so that the user does not have to draw the geometry. A feature where the user draws the geometry would be easy to implement, but would also be less user-friendly.

- Other load cases could perhaps be useful, even though the line loads and the distributed loads are the governing load cases for dimensioning balustrades and facades, respectively.

Bibliography

[1] Bength C., (2006) *Bolt Fixings in Toughened Glass*, Division of Structural Mechanics, Lund University, Sweden.

[2] Sedlacek, G., (2005) *Proposal for the content of new Eurocode on design of glass structures. Part 1: Basis of design - design of glass panes*

[3] Lee P-S., Bathe K-J., (2004) *Development of MITC isotropic triangular shell finite elements*

[4] ABAQUS Inc., (2005) *ABAQUS/Standard manuals, version 6.5*, Pawtucket, RI, USA.

[5] Ottosen N-S., Pettersson H., (1992) *Introduction to the Finite Element Method*, Prentice Hall Europe, Great Britain.

[6] Šolín P., et al, (2004) *Higher-Order Finite Element Methods*, Chapman and Hall/CRC, USA.

[7] Chen D.J. et al, (1996) *Interfacial stress estimation using least-square extrapolation and local stress smoothing in laminated composites*, Computers Structures Vol.58, No.4, pp-765-744

[8] Zienkiewicz O.C., Taylor R.L., (1991) *The Finite Element Method, Volume 2*, McGraw-Hill, Berkshire, England.

[9] Saad Y., (1994) *SPARSKIT: a basic tool kit for sparse matrix computations*, USA.

[10] Saad Y., (2000) *Iterative methods for Sparse Linear Systems*, Second edition, USA.

[11] Intel, (2006) *Intel Math Kernel Library Manual*

[12] Heath M., (2002) *Scientific Computing - An Introductory Survery*, McGraw-Hill, New York, USA.

[13] Woo M., et al, (1999) *OpenGL Programming Guide*, Addison-Wesley, Reading, Massachusetts, USA.

[14] MathWorks Inc., (2004) *MATLAB Help - Functions*

[15] Shewschuk J., *Triangle - A Two-Dimensional Quality Mesh Generator and Delaunay Triangulator*. http://www.cs.cmu.edu/∼quake/triangle.html [as viewed 2005-11-10].

[16] Siqueira, M., *CQMesh - A Convex Quadrilateral Mesh Generator* http://www.seas.upenn.edu/∼marcelos/cqmesh.html [as viewed 2005-11-10].

[17] *OpenGL - The Industry Standard for High Performance Graphics*, http://www.opengl.org [as viewed 2006-02-20]

[18] *PyOpenGL - The Python OpenGL Binding*, http://pyopengl.sourceforge.net/ [as viewed 2006-02-20]

[19] *PyGLUT*, http://www.btinternet.com/ ahcox/PyGlut/ [as viewed 2006-02-20]

[20] *METIS - Family of Multilevel Partitioning Algorithms*, http://glaros.dtc.umn.edu/gkhome/views/metis [as viewed 2005-12-10]

[21] *wxPython*, http://www.wxpython.org [as viewed 2005-12-01]

[22] *wxGlade - a GUI builder for wxWidgets/wxPython*, http://wxglade.sourceforge.net [as viewed 2005-12-01]

Appendix

Poly-file for the example in Section 4.4.3

The poly-file has the following format:

First line: ♯ of vertices	dimension=2	♯ of attributes	♯ of boundary markers	
Following lines: vertex ♯	x	y	attributes	boundary marker
One line: ♯ of segments	♯ of boundary markers			
Following lines: segment ♯	endpoint	endpoint	boundary marker	
One line: ♯ of holes				
Following lines: hole ♯	x	y		
One line: ♯ of regional attributes				
Following lines: region ♯	x	y	attribute	

In the example in Figure 4.13, the top segment is given the boundary marker 555, which Triangle passes on to the nodes that are created on that segment. Two holes are created. The starting coordinates for the holes are $0.3, 0.3$ and $1.7, 0.3$. Two regions, with the starting coordinates $0.31515, 0.31515$ and $1.71515, 0.31515$ are given the regional attribute 1. This attribute is passed on to the elements that Triangle creates in those regions. They are drawn red in Figure 4.14. The poly-file is:

44 2 0 1
1 0 0 0
2 2 0 0
3 2 1.5 0
4 0 1.5 0
5 0.3 0.315 0
6 0.308817 0.312135 0
7 0.314266 0.304635 0
:
:
41 1.68531 0.279775 0

42 1.67622 0.292275 0
43 1.67622 0.307725 0
44 1.68531 0.320225 0
44 1
1 1 2 0
2 2 3 0
3 3 4 555
4 4 1 0
5 5 6 0
6 6 7 0
7 7 8 0
8 8 9 0
:
:
42 42 43 0
43 43 44 0
44 44 35 0
2
1 0.3 0.3
2 1.7 0.3
2
1 0.31515 0.31515 1
2 1.71515 0.31515 1

Node-file for the example in Section 4.4.3

The node-file Triangle creates has the following format:

First line: ♯ of vertices	dimension=2	♯ of attributes	♯ of boundary markers	
Remaining lines: vertex ♯	x	y	attributes	boundary markers

For the example in Figure 4.14, the node-file is:

497 2 0 1
1 0 0 0
2 2 0 0
3 2 1.5 555
4 0 1.5 555

5 0.29999999999999999 0.315 0
6 0.30881700000000001 0.312135 0
7 0.31426599999999999 0.30463499999999999 0
8 0.31426599999999999 0.29536499999999999 0
:
:
491 0.5 1.5 555
492 0.28879689868563829 1.3269770123864146 0
493 0.28879689868563829 1.1394770123864146 0
494 0 1.3125 0
495 1.8499999999999963 0.9316937444655472 0
496 1.5 1.5 555
497 2 0.9375 0

Element-file for the example in Section 4.4.3

The element-file Triangle creates has the following format:

First line: ♯ of triangles	nodes per triangle	♯ of attributes					
Remaining lines: triangle ♯	node	node	node	node	node	node	attributes

For the example in Figure 4.14, the element-file is:

228 6 1
1 89 108 99 136 137 135 0
2 22 54 21 139 140 138 0
3 22 11 12 142 143 141 1
4 11 22 21 140 144 141 1
5 91 58 88 146 147 145 0
6 22 12 13 148 149 143 1
7 21 10 11 151 144 150 1
8 70 24 63 153 154 152 0
:
:
226 3 125 132 441 349 496 0
227 120 46 133 497 495 414 0
228 131 134 123 407 474 493 0