ANALYSIS OF TIMBER STRUCTURES CREATED USING A GE-BASED ARCHITECTURAL DEISGN TOOL

SCHOOL OF ARCHITECTURE, LANDSCAPE
AND CIVIL ENGINEERING
UNIVERSITY COLLEGE DUBLIN

A project submitted
to University College Dublin
for the degree of

ME in Structural Engineering with Architecture

By
Michael Fenton
23rd April 2010

ABSTRACT

This project fuses together structural analysis methods with the Grammatical Evolutionary design program GEVA-Blender. A new code has been written in the Python programming language which allows the user to seamlessly analyse whichever designs they desire in GEVA-Blender, using the open-source structural analysis program SLFFEA, with usability and simplicity being key issues. A survey of analyses of various bridge types is carried out, and the possibility of using structural analysis as a fitness function for GEVA is explored based on these results. A number of recommendations are made with regards to further study in this area, including limitations and restrictions to the scope of this project.

TABLE OF CONTENTS

LIST OF FIGURES

NOTATION AND TERMINOLOGY

Bar - Any 1-dimensional element within Blender or an analysis package. In analysis programs, a bar can be assigned properties so that it represents a beam.

Beam - Any 3-dimensional beam of given material, e.g. timber. In analysis programs, a beam is represented by a bar.

Function - A sub-set of a program which runs a specific task. A function is defined and assigned a name, so that other functions can then call this function within themselves, leading to a complex network of functions. See Appendix 1 for examples of functions.

Generation - A collection of individuals. GEVA creates 15 new individuals in each generation, with each new generation being spawned from the "fittest" individuals of the previous generation.

GEVA - Grammatical Evolution in Java

GUI - Graphic User Interface

Individual - Any single individual structure (or otherwise) that GEVA creates. An individual is part of a generation, which in itself is a subset of a population.

Module - A specific task within a function, such as an "if" or "for" loop.

Node - A point in 3-dimensional x-y-z space. A bar joins two nodes together.

Open-Source - Software, generally freely distributed, for which the user can readily modify the source code

Plugin - An additional piece of software that attaches to a host program / application and extends its capabilities and / or functions

Population - The total number of individuals so far in the entire GEVA run. The population can be viewed as the over-binding list of generations, themselves a list of individuals.

Program - An entire file consisting of functions and definitions of methods. Examples of programs can be found in Appendices 1 & 2.

1. INTRODUCTION

Ever since Charles Darwin's seminal work on evolution [Darwin (1859)], man has sought to copy nature's methods in creating intelligent machines. Nature has a surprising ability to come up with novel solutions to problems, and genetic mutation and diversity allows for continual evolution of a species. A man-made application of this genetic evolution has yielded a form of computer programming called Genetic Programming (GP), whereby programs evolve autonomously to create "better" programs with each successive generation [Poli et al. (2008), Willis et al. (1997), Koza (1992), DeJong (2006)]. Within GP there are numerous sub-types of programming methods; of particular note is a new and powerful method of programming: Grammatical Evolution (GE) [O'Neill et al. (2008), O'Neill et al. (2009), O'Neill et al. (2010), O'Neill & Ryan (2001), O'Neill & Ryan (2003)].

Using GE as a base, the UCD Natural Computing Research & Applications Group (NCRA) has created GEVA, a Java implementation of GE. In conjunction with School of Architecture, Landscape and Civil Engineering, the NCRA has produced GEVA-Blender, using GEVA as a plugin for the Python-scripted 3D modelling software Blender. The GEVA-Blender program allows the user to iteratively design structures constructed out of standard timber elements [O'Neill et al. (2010)]. A problem with these structures, however, is that they invariably need post-evolutionary structural enhancements and modifications. In order to assess the suitability for using structural analysis results as a fitness function for GEVA, a survey of the various types of structures GEVA-Blender produces must be made, along with their stress characteristics. This has inherent difficulties in itself in that the random nature of the selection of designs that GEVA produces makes comparisons between similar structures difficult to achieve, necessitating a large sample size.

In order to analyze the produced structures, a suitable analysis program must be found. Once sourced, automatic communication between the host program (GEVA) and the analysis program must be enabled, along with a re-designed Blender GUI incorporating the new analysis methods. Automating Finite Element software to accept outputs from another program has been identified as being notoriously difficult [Zhang & Van der Werff (1998), Logg (2007)], with most users ending up creating their own finite element software and file types in order to solve both compatibility issues and specify nodal complexities in the finite element software. A survey

of recommended methods for creating finite element and matrix-approach structural analysis files from structural data will be completed and various analysis programs will be tested for suitability.

2. LITERATURE REVIEW

2.1. Genetic Programming

Genetic Programming (GP) is an evolutionary method of computer programming based on Charles Darwin's theory of natural selection [Darwin (1859), Kicinger et al. (2005), Poli et al. (2008)], in which initial programs are selectively bred and crossed with each other to produce more suitable or successful programs. The method mimics that of nature, with two genetic operators: crossover (whereby parts of two programs are randomly mixed to form an offspring program with characteristics of both parents), and mutation (whereby one program is randomly altered to produce a new program with similar, but noticeably different, characteristics from the original). This allows for both "better" and "worse" programs to be created, i.e. some programs will have strengths and some will have drawbacks. Stronger, "fitter" solutions are given a high fitness value, while weaker functions that do not perform to standard are given a low fitness value. With each successive population of new programs, the low-fitness programs are ignored, while the high-fitness programs are selected to seed the next iteration of programs, thus ensuring a more suitable solution (Figure 1) [Darwin (1859), Poli et al. (2008)].

One advantage of the GP method is its ability to come up with novel solutions, in the same manner that nature will sometimes produce an interesting solution to an arbitrary problem. The mutation operation allows for population diversity, with solutions not readily present in the original program being developed [Willis et al. (1997)]. This has on numerous occasions resulted in GP operations deriving solutions regarded as innovative, and being patentable in their own right [O'Neill et al. (2010), O'Neill & Ryan (2003)].

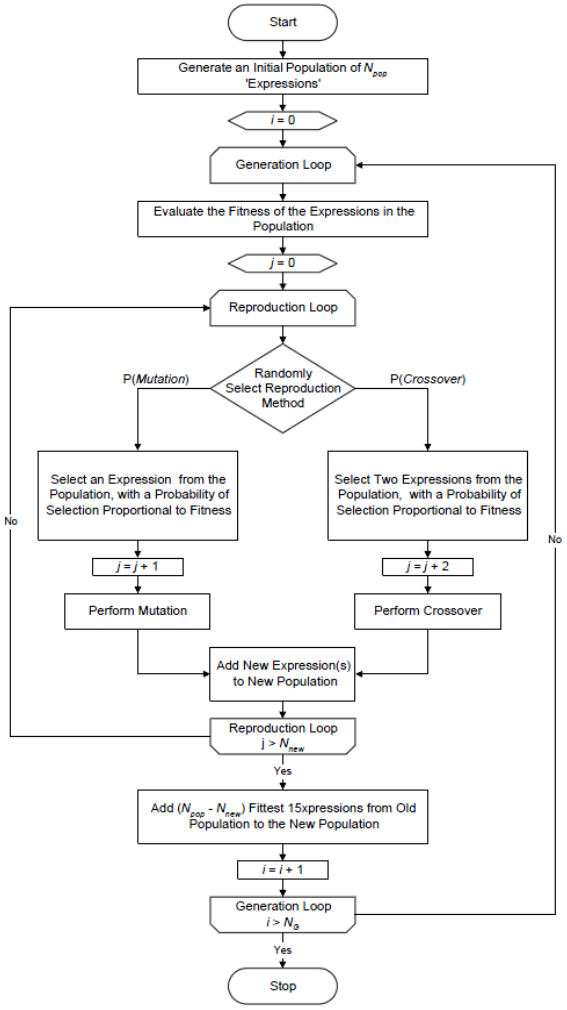


Figure 1: Typical Genetic Programming algorithm flowchart [Willis et al (1997)]

2.1.1 Grammatical Evolution

Grammatical Evolution is a grammar based form of Genetic Programming, which takes cues from the biological world in order to synthesise reproduction of two separate man-made codes (parents) to form an offspring code which is a mixture of both parent codes. GE differs from Genetic Programming in that it does not use a tree system to achieve this, but rather a user-defined grammar structure with sequences of integers representing tree nodes [O'Neill et al. (2008), O'Neill et al. (2010), Poli et al. (2008), O'Neill & Ryan (2003), Ryan et al. (1998)]. GE is essentially a mix of two sectors of programming – Evolutionary Algorithms and Grammatical Representation – to give an altogether more powerful type of program [O'Neill et al. (2010)].

2.1.2 Grammars

According to O'Neill & Ryan (2003), grammars dictate what *can* be done by a program, while a search algorithm within that program dictates what *should* be done. Thus GE does not tend to generate solutions which do not work entirely; rather it tends to generate a selection of potential solutions, each of which will differ in its particular fitness to fulfil a particular role defined by the program search. The grammars themselves contain a legal structure, or set of rules, which can be easily modified to trivially change the outputs of that grammar [O'Neill & Ryan (2003)]. It follows that the overall system can be set up to modify the grammars with response to the output in order to improve further outputs, and hence evolution of the grammars (Grammatical Evolution) is obtained.

What makes GE altogether more appealing than GP is that domain knowledge such as constraints or physical boundaries can be incorporated into the initial grammars to readily modify the output [O'Neill et al. (2010), O'Neill & Ryan (2003)]. The possibility of utilizing structural rules and criteria as boundary conditions for the grammatical representation is of particular interest and will be further explored in this paper.

2.1.3 An explanation of Shape Grammars and the I.E.C. Method

Shape Grammars are a method of encoding human domain knowledge into the evolutionary / generative process [O'Neill et al. (2010), Stiny & Gips (1971)]; a physical or visual representation of numeric problems and solutions. They differ from regular "phrase structure" grammars in that they are composed of an alphabet of shapes rather than one of symbols, and generate n-dimensional shapes rather than one-dimensional strings of symbols [Stiny & Gips (1971)]. They are particularly useful in interactive evolutionary computation (IEC), as their visual quality allows aesthetics, which are inherently difficult for a computer to judge, to easily be assessed on a subjective basis by human counterparts [O'Neill et al. (2010)]. The IEC method involves making the user the arbiter of the fitness function; with shape grammars, the user can simply set the fitness function (often a binary-style "yes-or-no" method) by viewing and judging the grammars. This method is used both widely and successfully as it is both easy to implement and lends itself well to user-judged designs.

2.1.4 GEVA

Recent work done by the UCD Natural Computing Research & Applications Group (NCRA) [O'Neill et al. (2008), O'Neill et al. (2010)] has culminated in the creation of GEVA (Grammatical Evolution in Java) [http://ncra.ucd.ie/geva/], an open-source implementation of GE in the Java programming language. Of particular note is that GEVA can be utilized as a plugin for 3d modelling programs such as Blender 3D [www.blender.org]. In this format the NCRA, in conjunction with members of the School of Architecture, Landscape and Civil Engineering, have created a specific program [O'Neill et al. (2010)] which, based on the aforementioned IEC method of human interaction, will design a wooden structure fit for human use using the principals of Grammatical Evolution. This program was run in conjunction with a similar project for students of the School of Architecture, Landscape and Civil Engineering, whereby students had to design and subsequently build a full-scale prototype shelter out of 1x2" and 1x3" wooden beams. It was found that while the GEVA program did provide aesthetically pleasing solutions (and in some cases solutions which were similar to those separately created by architecture students), the solutions invariably needed "post-evolutionary modifications" [O'Neill et al. (2010)] in the most common form of structural enhancements.

2.2.Finite Element

2.2.1 The idea behind the Finite Element method

Rarely in structural engineering is the engineer presented with a trivial problem; nowadays there is increasing demand for complex systems and solutions. The stiffness method of analysis [Ross (1985), Weaver & Johnston (1984), Rockey et al. (1983)] is a useful tool in the engineer's analytical arsenal, but when more complex load patterns or shapes are required this method becomes uneconomical to use. An idea proposed by M.J. Turner in 1956 was to discretize the overall object into a finite number of smaller elements, each with known properties. These elements can then be analysed individually to find local engineering solutions such as stresses and deflections per element, and the overall system can be described by summing the behaviour of the finite elements to find the global solution.

The elements themselves can take on the most appropriate form for the task at hand (they are user defined) [Ross (1985)]. However, there are some commonly used shapes:

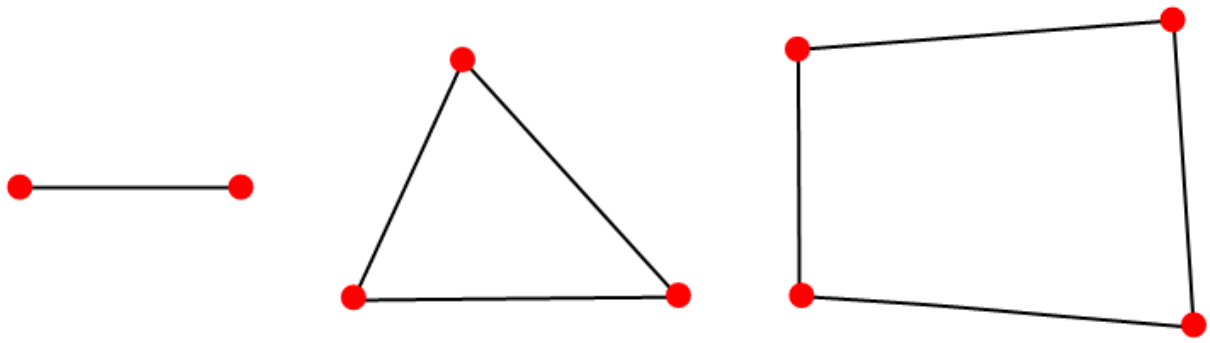


Figure 2: l - r: 2-noded, 3-noded and 4-noded elements

The bar (simple 2-noded), triangular (simple 3-noded) and quadrilateral (simple 4-noded) elements are the most basic of the elements, defined by simple corner nodes (Figure 2). They lack mid-sided nodes and as such cannot readily approximate curved shapes [Ross (1985), Weaver & Johnston (1984)]. The addition of mid-sided nodes allows for more sophisticated measures (Figure 3), allowing for the possibility of modelling curved surfaces more readily.

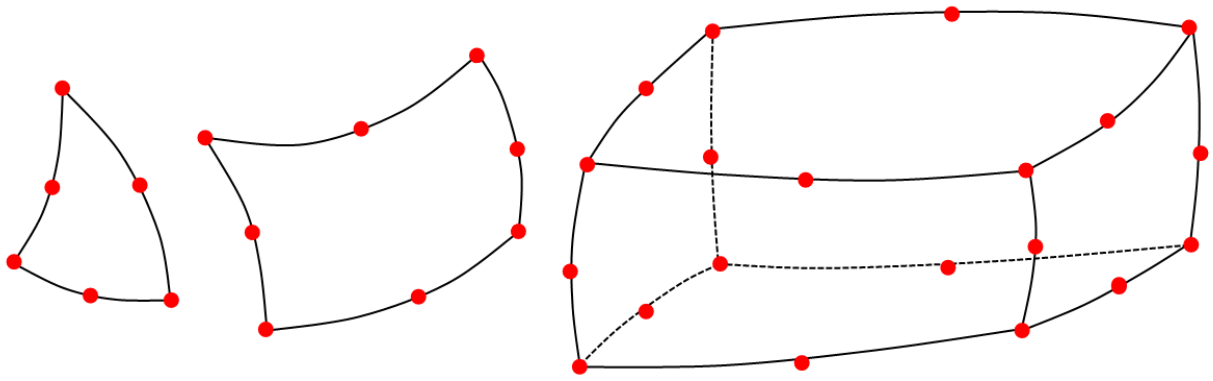


Figure 3: l - r: 6-noded, 8-noded, 20-noded curved elements

Now, it follows that the finer the subdivision of a body into its elemental constituents (termed the "finite element mesh"), the more accurate the model will become. A basic illustration is provided in Figure 4.

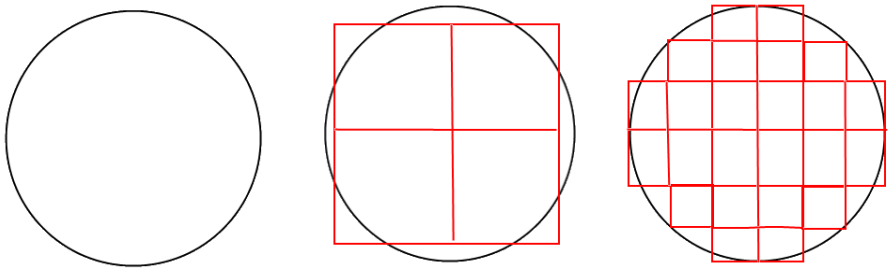


Figure 4: Original object (left) is approximated by increasingly fine meshes to produce a more accurate result

2.2.2 Structural Analysis Methods – Skeletal Structures After Ghali

This topic is covered extensively in greater detail in Ghali's *Structural Analisys, a Unified Classical and Matrix Approach* [Ghali et al. (2009)], and a summarised version will be presented here.

The GEVA-Blender shelter program uses 100m x 200mm timber beams of variable length to create arbitrary structures [O'Neill et al. (2010)]. These structures can be classified as skeletal structures under Ghali's terminology, meaning that their compositional members are long in comparison to their cross-section, making it possible to treat them as a series of 1-Dimensional beams which form 2-D or 3-D structures. In analysis of these structures the objective is to determine both the internal forces (stress, shear, moment) and the external reactions. The first

step is to ascertain the determinacy of the structure: Statically Determinate or Statically Indeterminate.

2.2.2.1 Statically Determinate

These structures can be solved easily using equations of static equilibrium:

$\Sigma F_x = 0$ – Sum of all forces in the x-direction equals zero

$\Sigma F_y = 0$ – Sum of all forces in the y-direction equals zero

$\Sigma F_z = 0$ – Sum of all forces in the z-direction equals zero

$\Sigma M_x = 0$ – Sum of all moments in the x-direction equals zero

$\Sigma M_y = 0$ – Sum of all moments in the y-direction equals zero

$\Sigma M_z = 0$ – Sum of all moments in the z-direction equals zero

2.2.2.2 Statically Indeterminate

These structures cannot be readily solved using the equations of static equilibrium, and require compatibility conditions equal in number to the degree of statical indeterminacy (the degree of statical indeterminacy is equal to the number of unknown forces in excess of the equations of statics [Ghali et al. (2009)]). This will invariably be the case with all outputs of the GEVA-Blender program. These outputs can also be assumed to be "linear elastic" in their response (Figure 5) due to the structures being small in overall size, and the relatively small forces involved (self-weight of the timber only applies [Sunley & Bedding (1985)]). Linear elastic statically indeterminate structures can be analysed via two main matrix analysis / simultaneous equation methods: the flexibility (force) method and the stiffness (displacement) method (other methods exist, e.g. moment distribution, which will not be covered here).

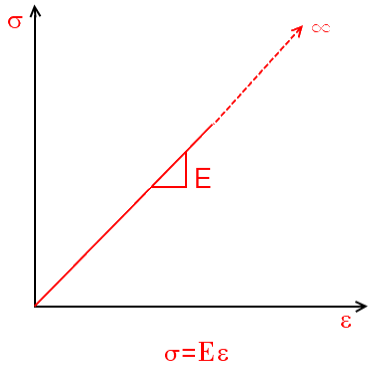


Figure 5: Material Linearity: Stress-Strain relationship [adapted from Ghali et al. (2009)]

2.2.2.3 The Force \ Flexibility Method

The force \ flexibility method of analysis is most applicable for hand calculations in that a degree of insight is required to apply it successfully [Rockey et al. (1983)], and as such will not be covered in great detail in this section. The method involves removing a certain number of reaction forces such that the structure is reduced to a statically determinate one, allowing it to be analyzed in a mathematically similar way to the stiffness method.

2.2.2.4 The Stiffness \ Displacement Method

The stiffness \ displacement method is the preferred analysis method for computer applications, as no structural insight is required. Essentially the computer fixes all members of the structure fully, which reduces the structure to a series of simple beams and columns. To this extent, the method is useful for structures with a high degree of statical indeterminacy. There are 5 basic steps to completing the stiffness method:

1. The structure is fixed completely such to isolate all members from each other.
2. Reaction forces and moments are then calculated for each individual beam and column, and a force matrix {F} is calculated consisting of the sum of the fixed-end forces or individual members. Internal moments and forces are also calculated at the reaction points of the restrained structure, and a matrix of stresses $\{\sigma^R\}$ is calculated.
3. The fixed structure is now progressively released with unit displacements at each successive node (with all other nodal displacements other than the one in question equaling zero). A stiffness matrix [S] is formulated composed of the forces resultant in the displaced structure (S_{ij} is the force or moment required at location *i* to maintain a unit displacement at *j*). The stiffness matrix is always square and symmetrical.
4. The displacement matrix {D} is calculated, i.e. the displacements required to eliminate the restraining forces introduced in step 2. The general relationship is

$$[S]\{D\} + \{F\} = 0$$

5. Once {D} is calculated, the final step is to calculate the matrix of total stresses in the structure, which can be obtained from the addition of the stresses in the restrained

structure and the product of the displacement matrix and the stresses due to unit displacement.

$$\{\sigma\} = \{\sigma^{R}\} + [\sigma^{UD}]\{D\}$$

2.2.3 Computer Structural Analysis

With the advent of the modern computer, analysis of more complex structures has become easier. The methods described by Ghali and Ross are the most popular in that they allow a full stress analysis of all types of structures. For the case in hand, however, Ghali proposes that the stiffness method is the most appropriate, as full finite elements are more suited to 3d elements such as plates, shells and solids. The GEVA-Blender program essentially creates a series of interconnected 1-D bars which form a 3-D space frame. Computer analysis of such a structure follows the same steps as proposed in the previous section:

1. Define global & local co-ordinates, and a nodal numbering system. Each element will have a co-ordinate system specific to its orientation, with a node at either end (Figure 6).

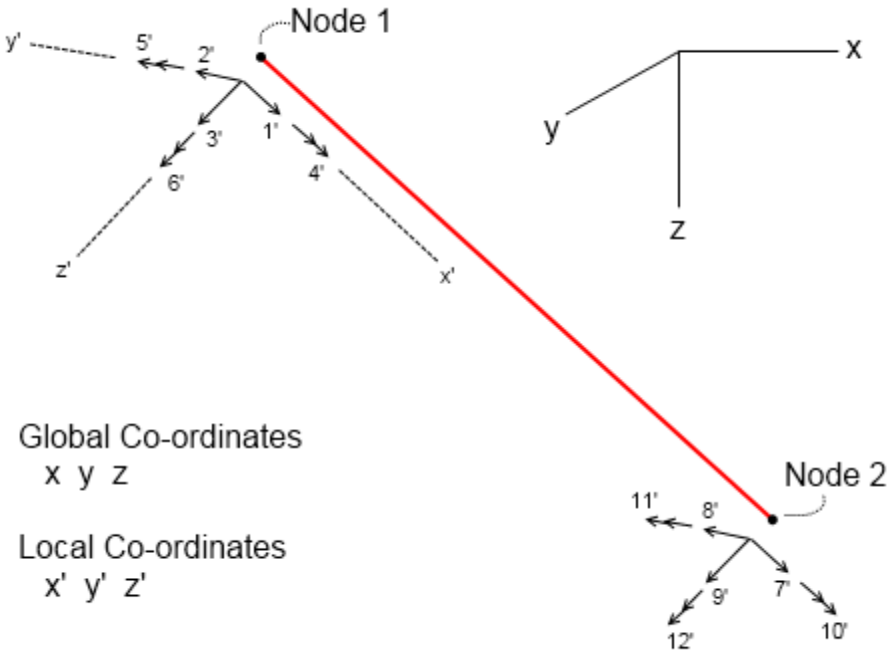


Figure 6: Local and Global Co-ordinates for element with 6 degrees of freedom (space frame) [adapted from Ghali et al. (2009)]

This ensures that forces are resolved relative to the member in question rather than to the global system.

2. Build suitable input file so as to describe the structure sufficiently and which contains all pertinent information in one location
3. Calculate member end forces and assemble Force matrix {F}
4. Release fixed structure and assemble Stiffness matrix [S]
5. Calculate displacement matrix D
6. Calculate stresses

For ease of analysis, Ghali recommends storing all structural information as an input file in the following manner:

Material Properties

[Material no.] [Properties, e.g. Young's modulus, area, density, I-values, etc]

Nodal Co-ordinates

[Node no.] [x value] [y value] [z value]

Element Connectivity

[Element no.] [first node number] [second node number] [material no.]

Support Conditions / Restraints

[Node no.] [x predescribed displacement] [y pre. displ.] [z pre. displ.]

Loading Conditions

[Node/Element no.] [F_x] [F_y] [F_z]

For describing restraints, Ghali suggests prescribing zero displacement for specific nodes, ensuring they are fixed in position.

This “input file” method of describing the structure works particularly well for space frame structures where it is usual to have many (if not all) of the elements with the same properties. Multiple element properties can easily be defined and assigned to appropriate elements if needed.

2.2.4 Automating the Finite Element Method

Zhang and Van der Werff (1998) led a project team to create an automated program that allowed packages of finite element analysis to communicate with Computer Aided Design Of Mechanism (CADOM) packages. The project, titled CIMOME (Computer Integrated Manufacturing of Mechanism Environments) essentially created a software environment for Computer Aided Design and Computer Aided Manufacture (CADCAM), facilitating user interaction and interface. It was noted that having an original mechanism model design which was not neutral enough (i.e. the raw data for the model was not of broad enough language to be accessible by different programs) led to narrowing of the field of possible compatible programs which could work with this model. To this end, CIMOME aimed to create a new universal data language entitled GMM (Generic Mechanism data Model), which would enable communication with a wide base of programs. Utilizing GMM facilitated the interaction process, both between the user and the program itself, and within the program, between analysis packages and CADCAM packages. A two-way system, GMM first took the output from the CADCAM programs, converted it to a usable language, analyzed it using Finite Element-based methods, and finally presented the user with the end results. The paper itself [Zhang & Van der Werff (1998)] focuses on the first component of GMM utilization, GMMFEM (from GMM to Finite Element Modeling), and demonstrates the method with the finite element program RUNMEC. This method can be compared with Ghali’s method as described in the previous section, and a preference of methods would depend on the accessibility of raw model data. If node and member positions are known and provided, then Ghali’s method is the appropriate one to use. However, if the overall structure is defined by a different method, as in processed .dxf files or .dwg files which are much more complex than raw input data, then Zhang & Van der Werff’s methods are more applicable.

2.3.Mixing Genetic Programming and Structural Analysis together

An excellent comprehensive review of evolutionary computation and structural design was compiled by Kicinger in 2005. It states that in evolutionary computation in structural engineering design, the most difficult aspects of the design are:

i) appropriate representation of the engineering system itself and

ii) finding a suitable evaluation function.

It is the aim of this paper to investigate the possibility of utilizing the Finite Element method of structural analysis as such a fitness function (in conjunction with the Interactive Evolutionary Computation method), enabling the program to assess and evaluate viable solutions using state of the art structural analysis software. However, Kicinger also recognises that using structural analysis programs for evaluation and optimisation of proposed solutions has an inherent flaw in that "an exact location of the boundaries between feasible and infeasible regions (design solutions) cannot be specified", i.e. there is no way of discerning more optimal or less optimal solutions, the balance of the feasible and creative. Outputs of analysis programs are not of algebraic format, which can be manipulated to obtain more optimal solutions (as is the case with grammars), but are a direct singular solution from the structural analysis program itself, with no scope for modification.

If it is proposed to use structural analysis results as the fitness function, then there are methods of minimising the number of infeasible proposed solutions according to Kicinger, such as special genetic operators which continually restrict such solutions during the initial grammar reproduction phases. Kicinger also argues, however, that in general the less restriction on the evolutionary procedure the better, in order to create the most varied populations.

3. INITIAL WORKS

3.1. The Grammars

Since commencement of this project, there have been three main iterations of the GEVA grammar. The first grammar was that of the original shelter design challenge [O'Neill et al. (2010), the second was termed the Higher Order Function (HOF) grammar [Yu (2001)], which introduced a new method of generating functions, and the most recent was termed the "bridge grammar", which combined the methods of the HOF grammar with new geometric representations to create bridge-style structures. Each grammar represented a new set of challenges to be overcome, mostly in the analysis of the Blender outputs themselves.

3.2. Analysis Methods – Trials done using the Shelter Grammar

As discussed in chapter 2, the original GEVA-Blender program was used to create a shelter of appropriate scale for human use. This grammar was used as a test-bed for analysis methods, so that future grammars would prove easier to analyze due to familiarity with the methods, etc. The intention at the beginning of the project was to save potential individuals in a .dxf file format, and then open this file using the analysis program. A number of analysis programs were intended to be tested for suitability of analysis of the GEVA-Blender structures, including:

i) STRAP

ii) Robot Structural Analysis (Robot)

The initial choice of analysis program for use in this project was STRAP; being available on the UCD network, it was both the most appropriate and the most readily available. Logistical difficulties came into effect, however, due to the lack of UCD Network access in the Complex and Adaptive Systems Laboratory (CASL) building, meaning that analysis using STRAP would need to be carried out in a separate building on a separate computer. For both availability and ease of use, Robot was deemed the most appropriate program available, being freely downloadable under a student licence from the Autodesk website [www.autodesk.com]. Initial trials proved promising; it was possible to directly import .dxf format files created in GEVA-Blender, which allowed for easy rendering (Figure 7).

The Blender outputs, while in a .dxf file format, were not immediately compatible with either the Robot or the STRAP programs. With STRAP, the files were invariably too corrupted

or inconsistent to allow the application to run, while with Robot, the files were allowed to open, but the structures themselves were found to be corrupted (Figure 7). In Robot, each beam was represented by 8 nodes (as expected of a 3D finite element representation of a beam), but these nodes were connected by diagonal bars to form a series of N-shapes (Figure 8), rather than connected by horizontal and vertical bars in a "brick" layout.

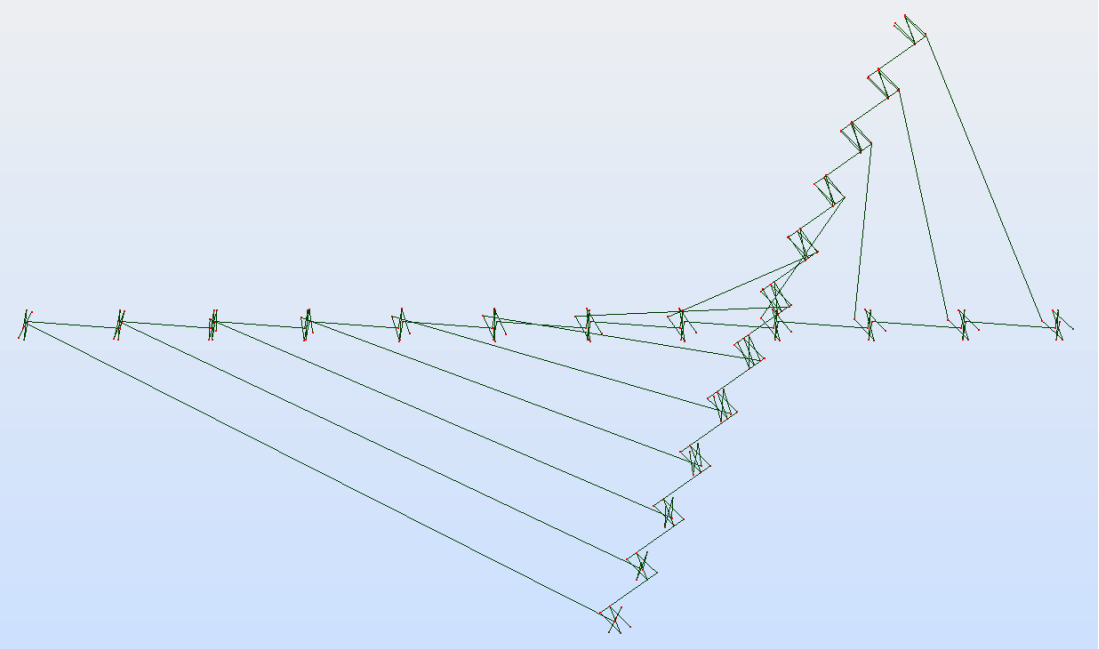


Figure 7: Test output using Shelter grammar, initial corrupted Robot file

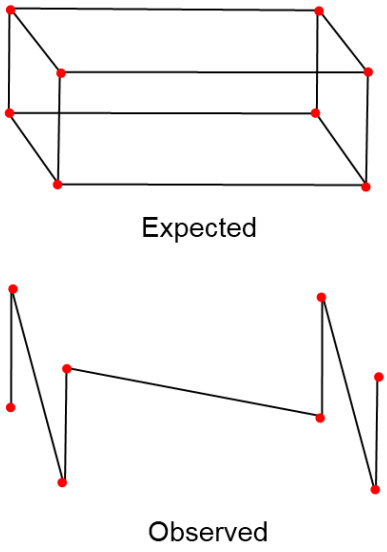


Figure 8: Expected "brick" beam representation vs. observed "diagonal" beam representation

While this was an initial setback, it was realized that the very nature of these diagonal bars would allow for manipulation of the file which would result in a fully analyzable structure.

The way GEVA operates is that it intersects adjacent beams on their centre lines. Essentially, there is a frame of one-dimensional elements that runs through the centre of each beam in the structure itself which makes up its skeleton. With the "diagonal" beam representation (Figure 8), the diagonals at the ends of each member intersect at their mid-points. It is then possible to create a string of connected nodes at these intersection points, completing this central 1D skeleton (Figure 9).

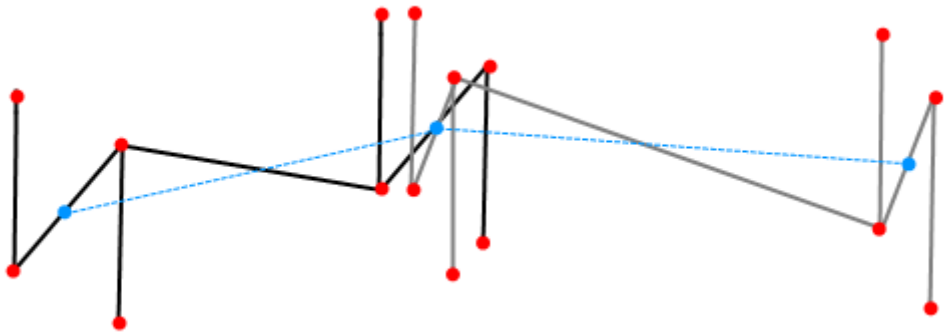


Figure 9: Two connected beams (black and grey), and the central 1D line that represents both (blue)

Finally, all original nodes and bars can be deleted, leaving a clean wireframe model of the structure (Figure 10).

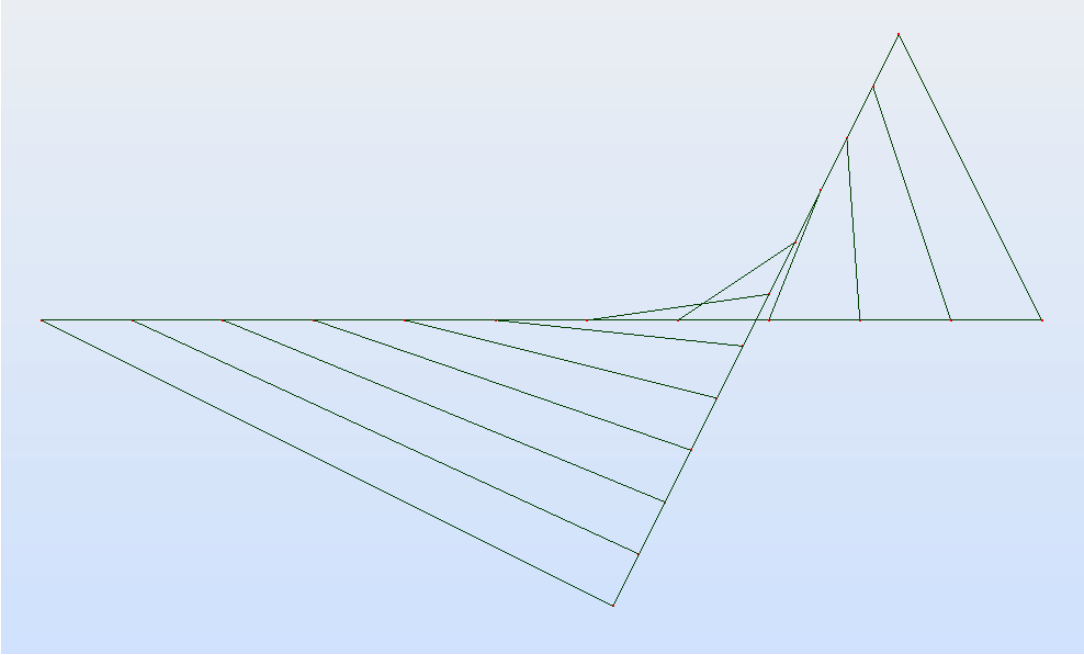


Figure 10: Test output using Shelter grammar, amended Robot file

Analysis of these amended structures is then easy. It is possible to assign section properties to each bar such that they represent timber beams (Figure 11). The structure can then be analyzed for the stresses and displacements resulting from the action of its self-weight (see Appendix 3 for Robot analysis of the stresses in the vertical bar from the structure in Figure 11).

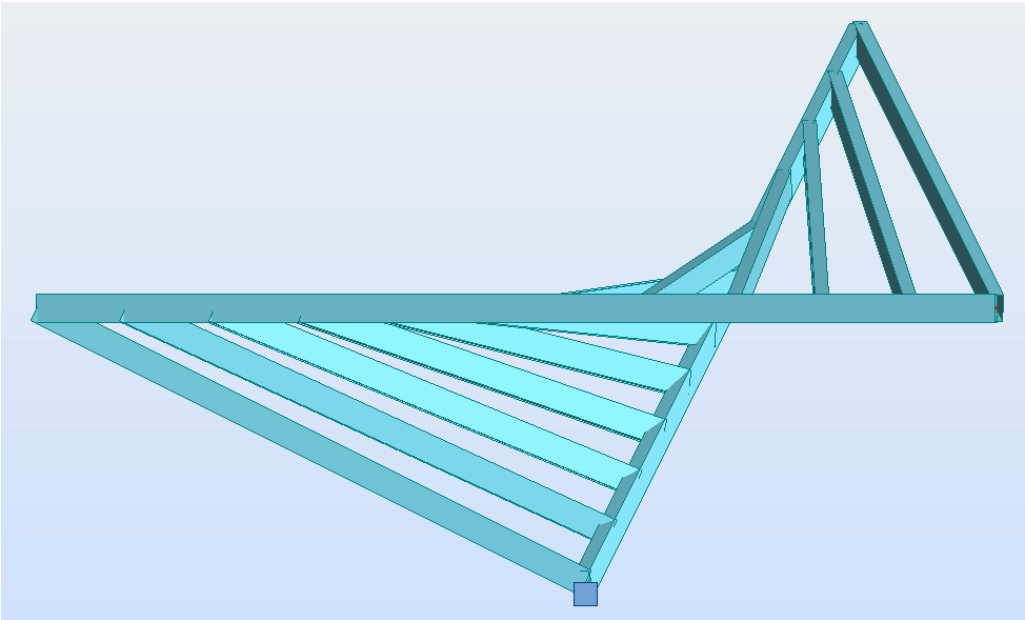


Figure 11: Test output, rendered

3.3.Frustrations encountered while working with Blender

The intention to analyse larger structures using a separate program resulted in an unpleasant discovery: there are numerous problems associated with using Blender 3D as a design package. Blender itself is intended for use as an animation tool, which has inherent implications for anyone wishing to use it as a design program. With animation, the focus is purely on aesthetics. As such, there is no requirement for an animation package to create solid objects; the user only requires exterior faces and 2-D planes. With blender, while it may seem that solid objects (e.g. beams) are being created, when these elements are analyzed in a more thorough program such as Robot or AutoCAD, it is found that they are not in fact solid objects, but a series of interconnected facades which give the appearance of a solid body (Figure 12). With Robot, it is not possible to further manipulate these elements to create the desired 3-D effect without having to re-create the entire structure, which for large bridge structures with upwards of three hundred nodes and beams would not be feasible.

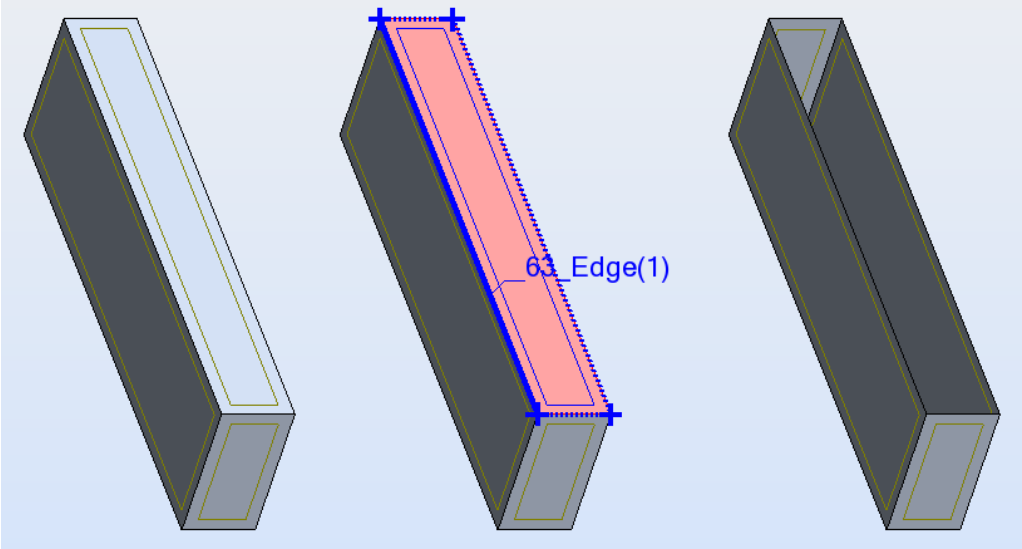


Figure 12: What seems like a solid brick on the left (a) is actually just a composition of faces and edges (b) which can be deleted to display a hollow shell of faces

One potential solution for this problem would be to have GEVA-Blender create single one-dimensional lines instead of beams. In many respects, this would be ideal, as it would allow the user to specify the working materials within the analysis package itself. This would also provide greater power to the engineer to “upgrade” any particular section that the analysis

package deems to be unfit or that fails. An addendum to this is that GEVA only creates a single type of object – the standard timber beam. With element selection within the analysis package, the user has the option of specifying as many separate elements as they deem appropriate for the task at hand.

There is, however, a fundamental flaw in this solution: Blender cannot create simple one-dimensional lines. This may seem terribly counter-intuitive, that it skips over the most basic forms of design (single points and 1-D lines) and provides the user with much more complex functions, but the nature of animation does not lend itself well to a single dimension. One possible solution might be to create extremely thin 2-D planes that, for all intents and purposes, look like lines, but that are compatible with Blender's architecture. Unfortunately, however, this would make the outputs very difficult to see during the iteration process, and once the file was exported to Robot for analysis, there would still be four bars for every single beam (in a rectangular shape), instead of one bar per beam. This could be fixed with the intersect function in Robot, but the user would then essentially have to "join the dots" to complete the structure. For a design with a lot of nodes and members, this would be tedious and time consuming.

3.4.Export File Formats

After much experimentation with different file export combinations, it was discovered that many of Robot's difficulties with Blender outputs are to do with file types. The Robot analysis package is very sensitive to subtle differences in the .dxf file formats coming from Blender. Blender itself is also tricky to use, as there are multiple methods of achieving the same apparent result with various functions of the program, some with better results than others. A case in point is exporting .dxf files from Blender.

There are two options for exporting .dxf files. One can either use the F2 key which will call up one type of exporter (file > export > DXF also performs the same function); the other is accessed through file > export > Autodesk DXF (scripts > export > Autodesk DXF also performs the same function). These two options provide very different results. While both give .dxf files, the former invariably yields partial structures in the "diagonal" arrangement (with all vertical members missing and random additions) (Figure 13 - a), while the latter provides a host of options which allow the user to tailor the output file specifically to their needs. The key is to set the "Mesh" option to "3DFACEs" (demonstrated in Figure 15), rather than the standard

"POLYFACE". This leads to a full model export in the dxf file (Figure 13 - b & c). These figures show the two examples of input rendering in Robot – it is possible to explode each object into finite elements, an option which creates nodes and bars for each beam.

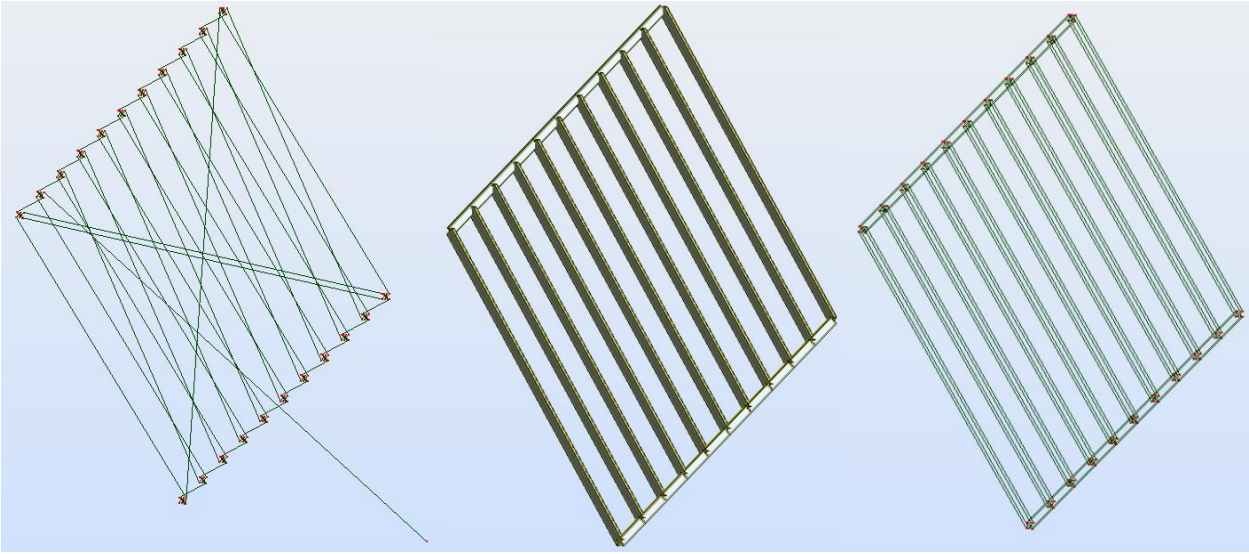


Figure 13: .dfx output variances when viewed in Robot. a) – diagonal arrangement (with additional bars not present elsewhere); b) full output (beams rendered); c) full output (beams exploded into finite elements)

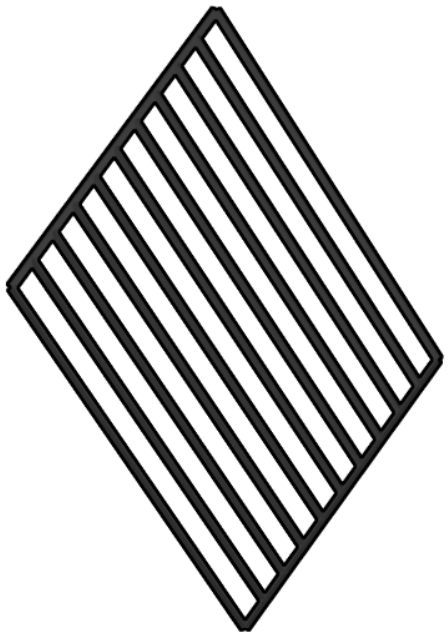


Figure 14: Original .dxf test output, as rendered in Blender

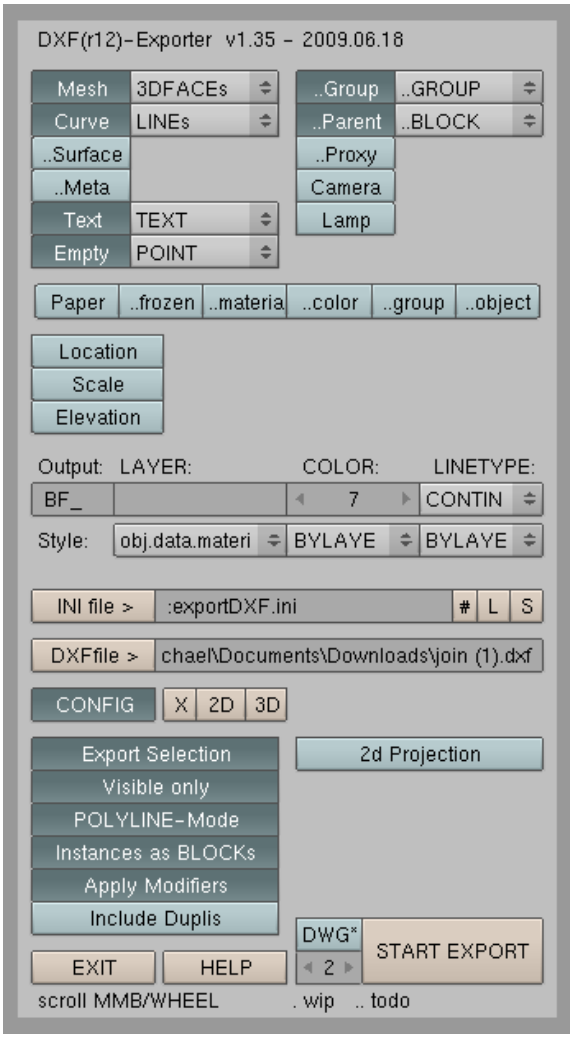


Figure 15: Correct settings of .dxf exporter

An interesting point can be raised in relation to AutoCAD and Robot Structural Analysis. Both programs are produced and run by Autodesk, and one would expect that their internal architecture is broadly similar. However, Robot was not created by Autodesk, but was a separate program which was bought by the company. To that extent, it differs in some respects to AutoCAD.

A key selling point of all the Autodesk software is compatibility. The programs are supposed to communicate with each other and support similar file types. Despite that, unfortunately, it is not possible to import any objects from AutoCAD into Robot that do not consist solely of “LINEs”. This encompasses regions, blocks, 3d objects (including extruded regions) and solids. On a confusingly related note, Blender ties in here.

Now, it has been noted that the correct export of Blender .dxf file types requires the “Mesh” option to be set at “3DFACEs”, rather than “POLYLINE”. However, when set at “POLYLINE” and the file opened in CAD, the beams are single coherent objects rather than the assortment of six faces (creating a single hollow block), which is the desired (but ultimately elusive) outcome for Robot. It would be expected that two compatible programs released by the same company would have the same results when opening a file, but it is not the case. With the “POLYLINE” option selected and the file opened in Robot, the structure is displayed as a collection of faces in the same fashion as if the file were opened in Robot with 3DFACES.

4. THE DESIGN CHALLENGE

4.1. Synopsis

In 2008/09 there was a design challenge that formed part of a module for fourth year Architects and Structural Engineers. The challenge was to design and build a timber shelter fit for two persons which would provide protection from the elements. The NCRA group at UCD caught wind of this challenge and decided to formulate a program based on the GEVA platform that would generate virtual models of such structures [O'Neill et al. (2010)]. This year, a new challenge was put forth: current fourth year Architects and Structural Engineers were split into three groups, each charged with designing a bridge. These bridges were to be designed in accordance with Co. Carlow's zoning and planning laws for installation in St. Mullin's archaeological site, to facilitate access between the church grounds and the Holy Well. Students were to undertake the design, building and installation of these bridges in cooperation with Carlow County Council (who were to approve the designs and subsequently fund the material costs). It was decided within the NCRA that another design challenge was needed which would focus the minds of all those involved and give the GEVA project some drive (as the original shelter challenge had done). The new bridge design challenge was deemed perfect for a number of reasons:

a) It would allow experimentation with a new form of shape grammar, termed HOF grammar (Higher Order Function grammar).
b) A bridge is by its nature a purely engineering structure, while the first application of GEVA (the shelter challenge) was developed as an architect's aid and as such had no engineering considerations. It was therefore considered a logical progression of the grammar and a perfect tool for this project.

While the bridges designed by the teams of student architects and engineers would be constructed and eventually installed in a public location for use in pilgrimages, the NCRA's bridges would not be intended for construction.

The aim of the new design challenge was to create a viable design subject to most of the same constraints as those of the students' teams:

a) Bridges must be of small scale air dried oak sections, moisture content 20% or more, available lengths 4.9m or less, grade D30
    - Strength properties of grade D30 (from BS EN 338 – 2003 – Structural Timber Strength Classes):

        Permissable stress in bending = 30 N/mm^2

        Permissable stress in tension (parallel to grain) = 18 N/mm2

        Permissable stress in compression (parallel to grain) = 23 N/mm^2

        Mean Modulus of Elasticity = 10,000 N/mm^2

        Density = 530 kg/m^3
b) Bridges must be prefabricated and disassembled for shipping to site, i.e. no glue. Stainless steel bolted joints and steel cable are permitted for connections.
c) Bridges must comply with Eurocode Part 1.2, also British Standard BS EN 1995-2:2004 Eurocode 5 Timber Bridges
d) Bridges must attend to self weight (dead load), anticipated crowding of pedestrians, and all lateral loads specified in codes/regulations
    - Pedestrian loading as defined by BS 5400-2: 2006 is $5kN/m^2$
    - Lateral loading and wind load will not be considered in this project
e) Bridges must attend to guidelines regarding detailing and durability to ensure a 20 year life span.
f) Bridges must clear the maximum flood level.
g) Bridges must attend to issues of disabled access where appropriate, meaning minimum 2m width and appropriate slope for wheelchair access
h) All bridges should have appropriate deck surface to ensure non-slip surface and draining of water away from structure
i) Attend to local soil conditions for foundation design

These constraints only serve to focus the design process by further eliminating variables and unknowns from the initial selection process.

4.2. Assumptions

A number of assumptions were made in the creation of the new bridge grammar with relation to the design challenge, most of which were made with the intention of freeing GEVA from the large number of constraints which the planning regulations would have imposed. These included:

Site specific details – the river banks of the intended site in St. Mullins are uneven; one side is higher than the other. This was ignored on the premise that the lower side can be built up to the same level as the higher.

Foundations – It was assumed that foundations of the designs would be a post-evolutionary consideration and thus would not fall under the jurisdiction of the GEVA-Blender project.

Slenderness of members – Member length was not limited in the grammars, in order to allow GEVA to create more interesting and varied results. This has obvious structural implications, which are dealt with in Chapter 7.

4.3. The Higher Order Function (HOF) Grammar

The original shelter grammar used first-order functions, i.e. a function could call a method which relied on a number or argument. With the Higher Order Function grammar however, the argument to a function can be another function (similar to a function of a function in second-order analysis) [Yu (2001)]. This allows for patterns of re-use (Figure 17), which generates more coherent structures.

The original shelter grammar operated on a very hit-and-miss principle. Individuals consisted of randomly placed groups of beams, each with a specific pattern. Patterns included curves, circles, ladder-type structures (with horizontal beams spanning across an outer frame, Figure 16 a), ordered simple beams (Figure 16 b), and randomly ordered beams (Figure 16 c), among many others. What the grammar lacked was any coherence in the relation of these patterns to each other; individuals could consist of any combination of any or all of the grammars, often resulting in a jumbled-up mess (Figure 16 d).

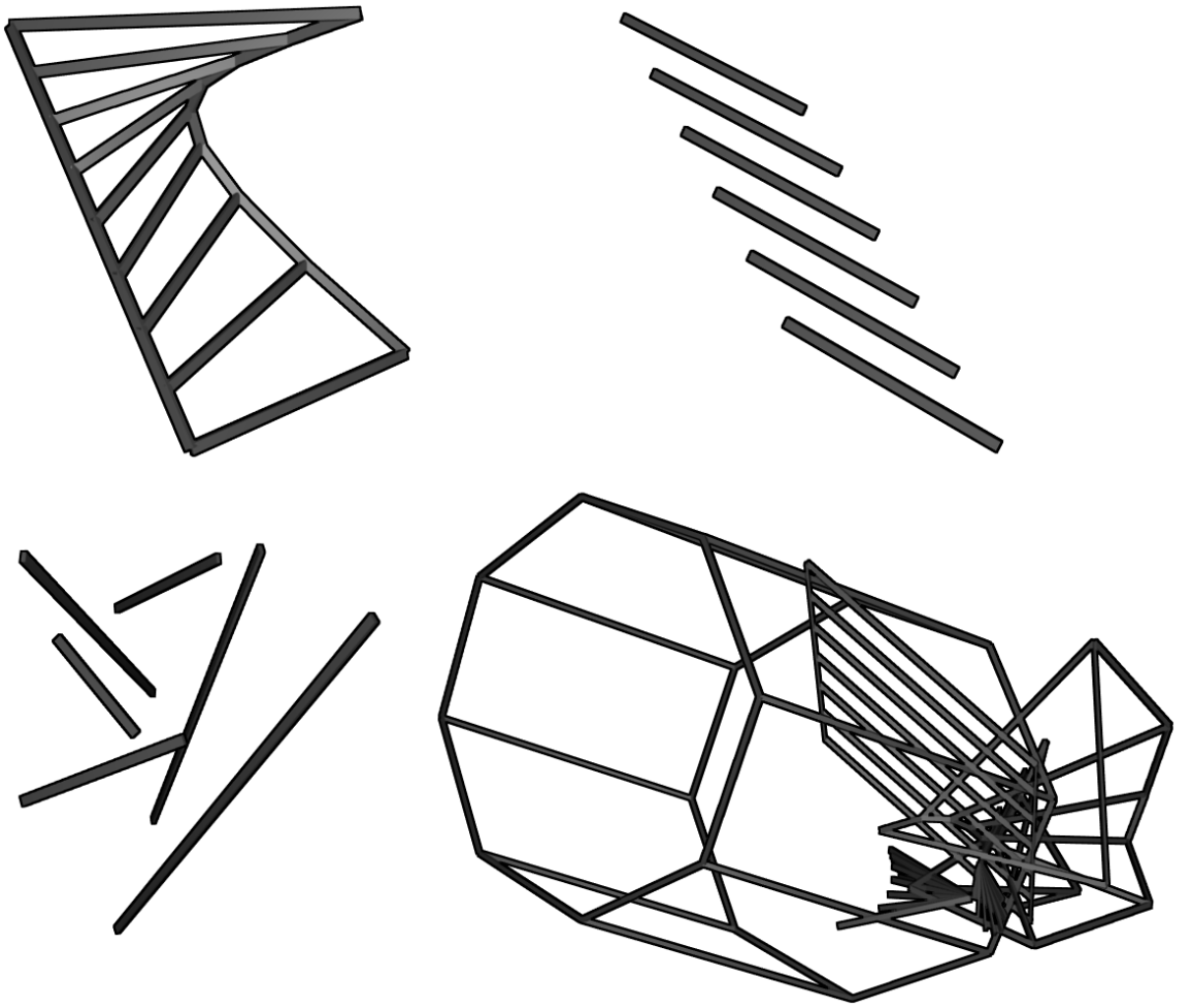


Figure 16: Original shelter grammar individuals: a) ladder structure, b) ordered beams, c) disordered beams, d) random collection of many types

The notion of connections between the groups of structures was not addressed until the implementation of the HOF grammar, whose introduction changed the focus of the GEVA-Blender program. This new grammar was to be considered an improvement on the previous grammar type as it introduced the notion of connections between groups of members for the first time, which produced more coherent designs (Figure 17).

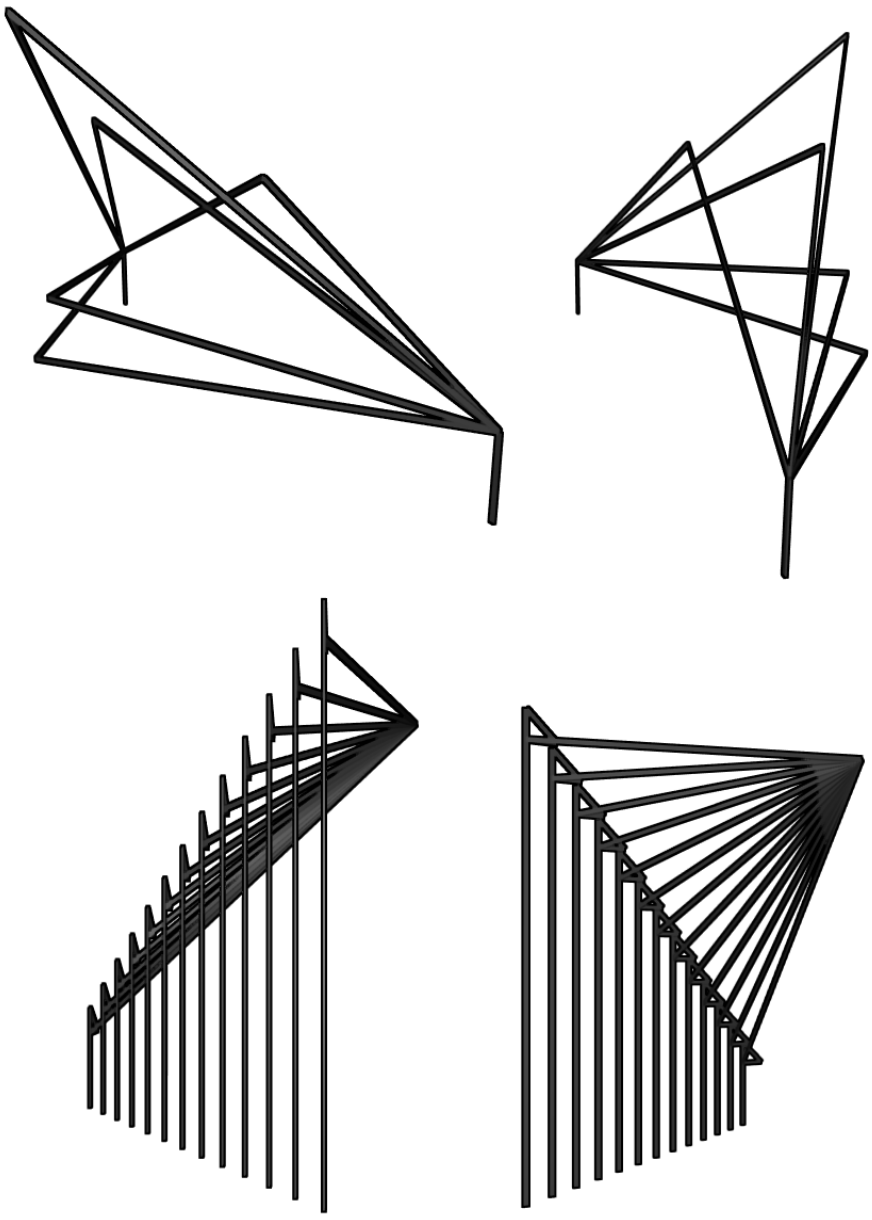


Figure 17: Coherent and connected designs a) and b) created using HOF grammar, showing patterns of re-use

Randomly placed groups of designs (Figure 16 d) no longer appeared in the individuals, the addition of higher-order functions which could recursively call themselves resulted in single consistent designs rather than multiple disorganised designs.

4.4. The Bridge Grammar

The GEVA Grammar's third iteration was created with bridge design specifically in mind. The grammars had set parameters – a basic outline was hard-coded (set) in every design:

Span of 10m

Bridge width of 2m

Horizontal walkway connecting the two ends of the bridge via a straight path

Variable parameters in the grammar included:

Number and positioning of horizontal beams in walkway

Number, length, angle and positioning of beams in vertical handrail sections

Rise of arch from 0% (flat bridge) up to 20% of its length.

Number and angle of branches in handrails

Structural considerations of the grammars were ignored at the time of their implementation in recognition of the fact that work done by this project would accommodate that need. The main focus of the grammars was that of aesthetics, with GEVA spawning ideas for intricate handrails.

Initial bridge grammar outputs had a variety of methods for creating structures, with varying forms:

i. randomly creating points, reflecting them about a central axis and then connecting them up to create a symmetrical truss-style bridge (Figure 18 - b) or an aesthetically pleasing but structurally infeasible design (Figure 18 - d),

ii. generating a sine wave to act as a rudimentary arch (Figure 18 - c). This grammar was used as a test-bed for the bridge forms; later iterations of the bridge grammar mirrored both handrails about the central longitudinal axis of the bridge to create a symmetrical structure with handrails on both sides.

iii. randomly creating points, but with every point connecting back to a single point at one side of the bridge, creating a messy structure (Figure 19 b)

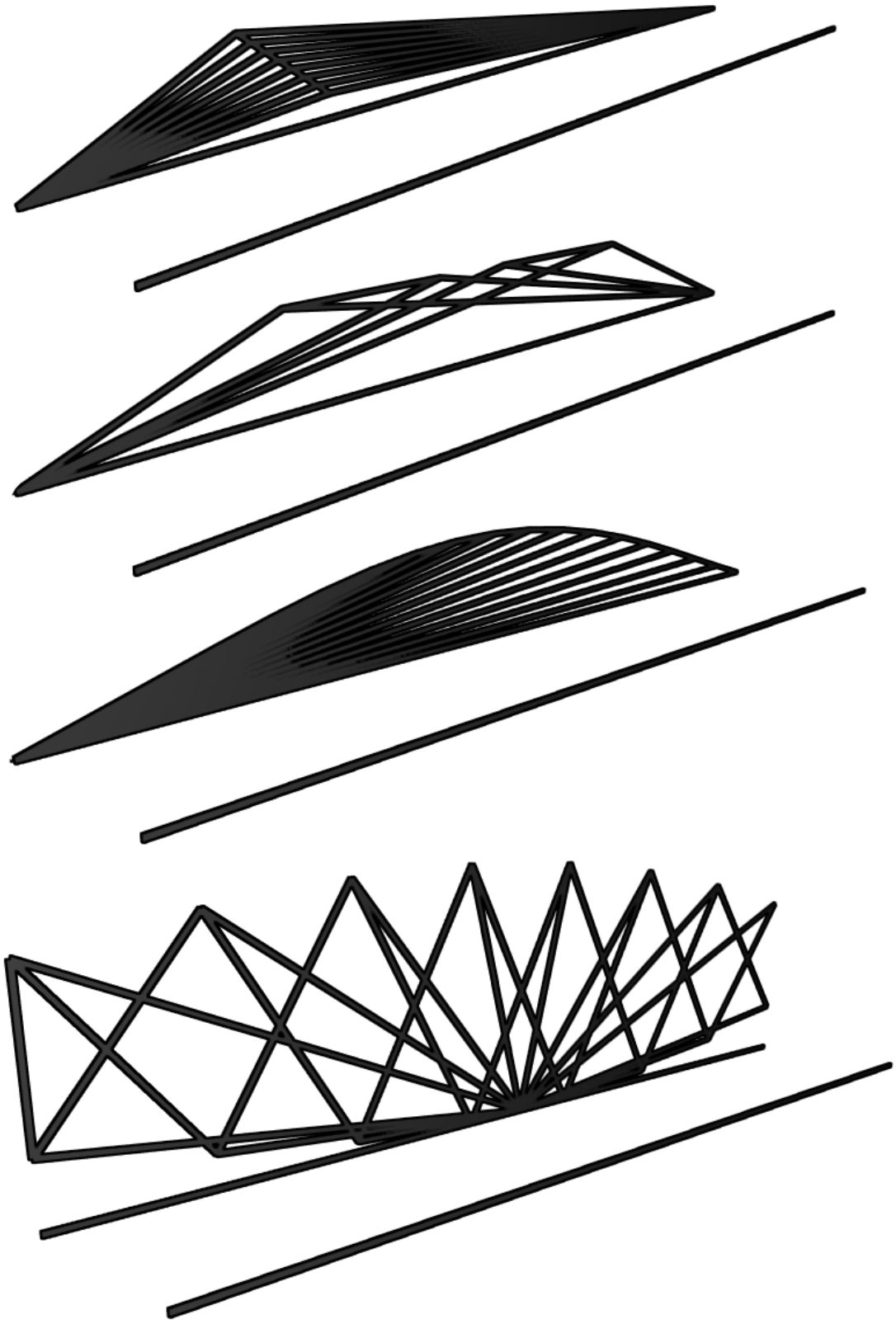


Figure 18: Initial 2D bridge grammar outputs a), b), c) and d)

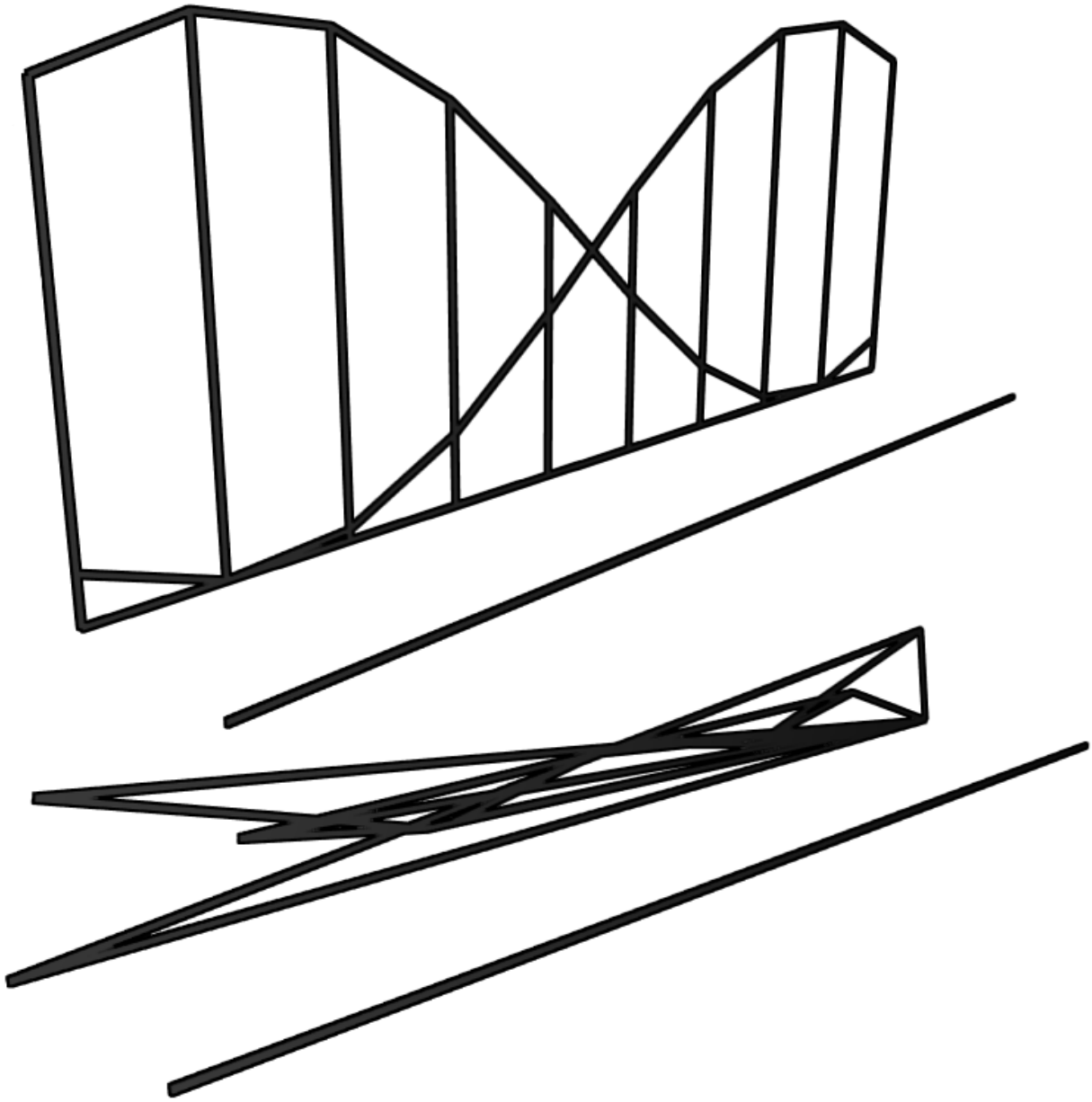


Figure 19: Initial bridge grammars with more interesting (a - top) or infeasible (b – bottom) results

The newer iteration of the bridge grammar takes these ideas a step further. An arch parameter is defined with a variable height, along with second-order curves for the handrails which can vary in three dimensions (see Figure 20 & Figure 21). Branches for the vertical handrails are also introduced, with anything from one to five arms. The two sides of the arch are essentially identical, with one being a mirror and offset version of the other. The corresponding points along the bottom of the bridges are then linked up to form the walkway.

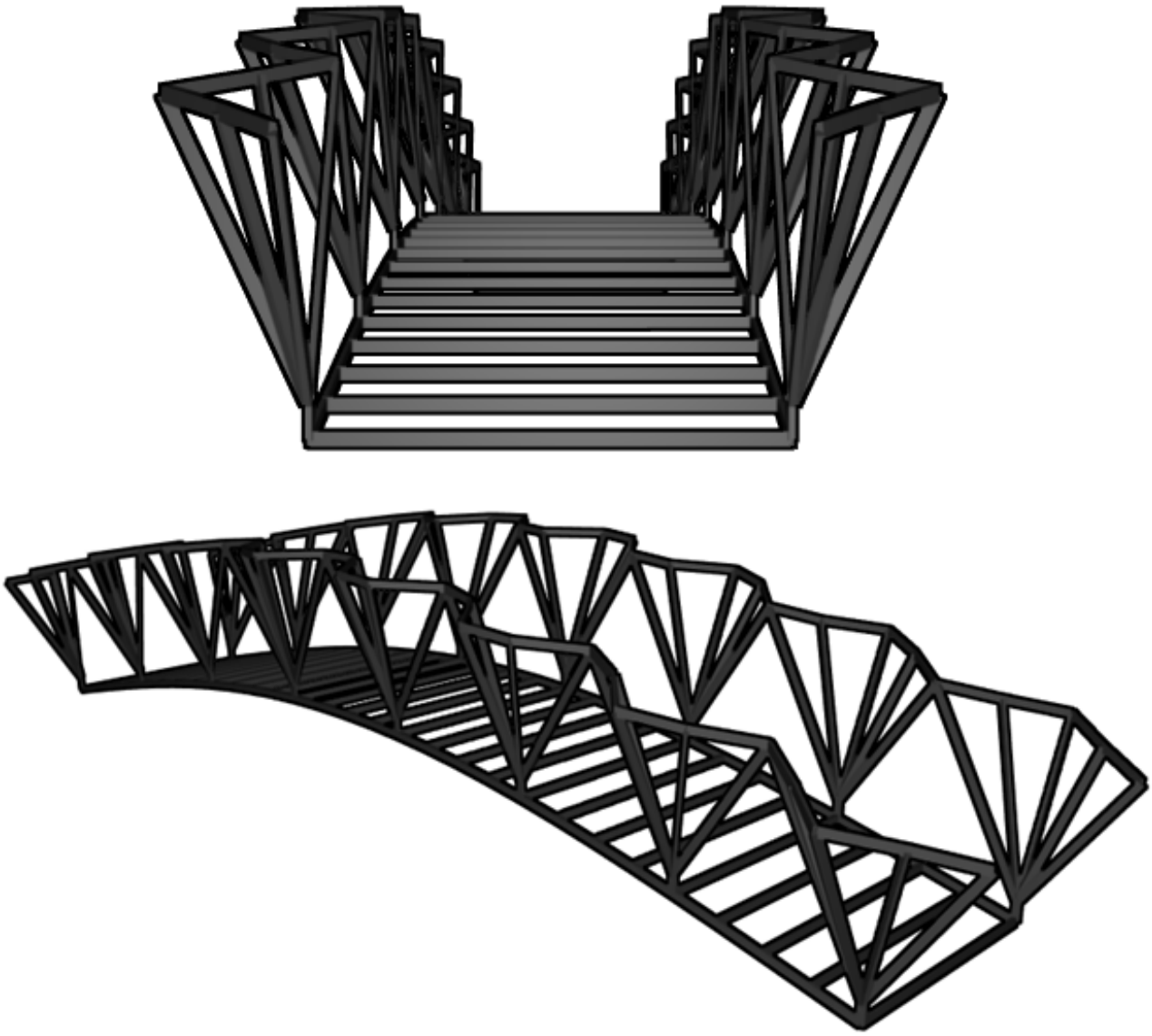


Figure 20: Sample a - arched bridge created using the 3D bridge grammar

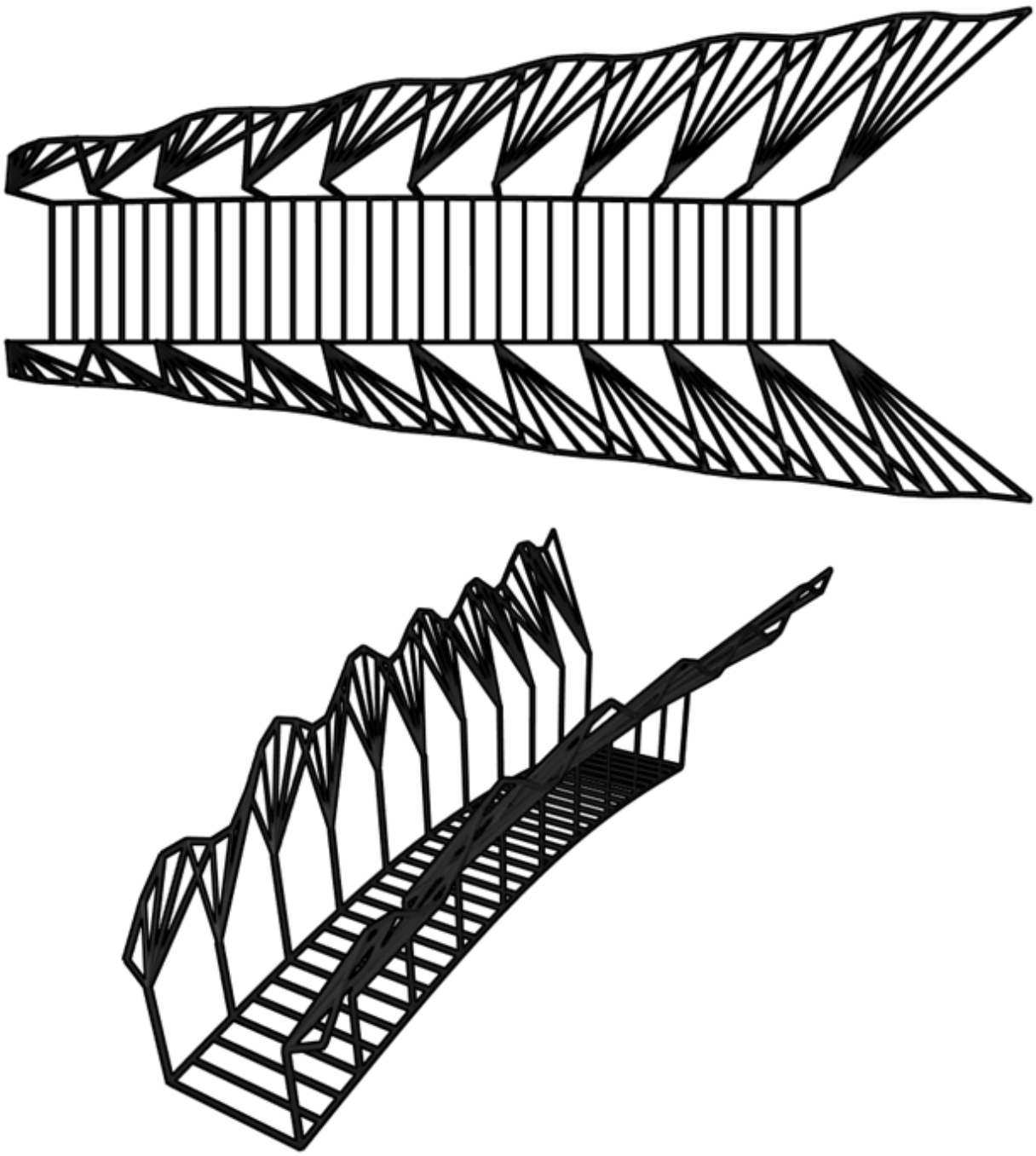


Figure 21: Sample b – winged arched bridge created using the 3D bridge grammar

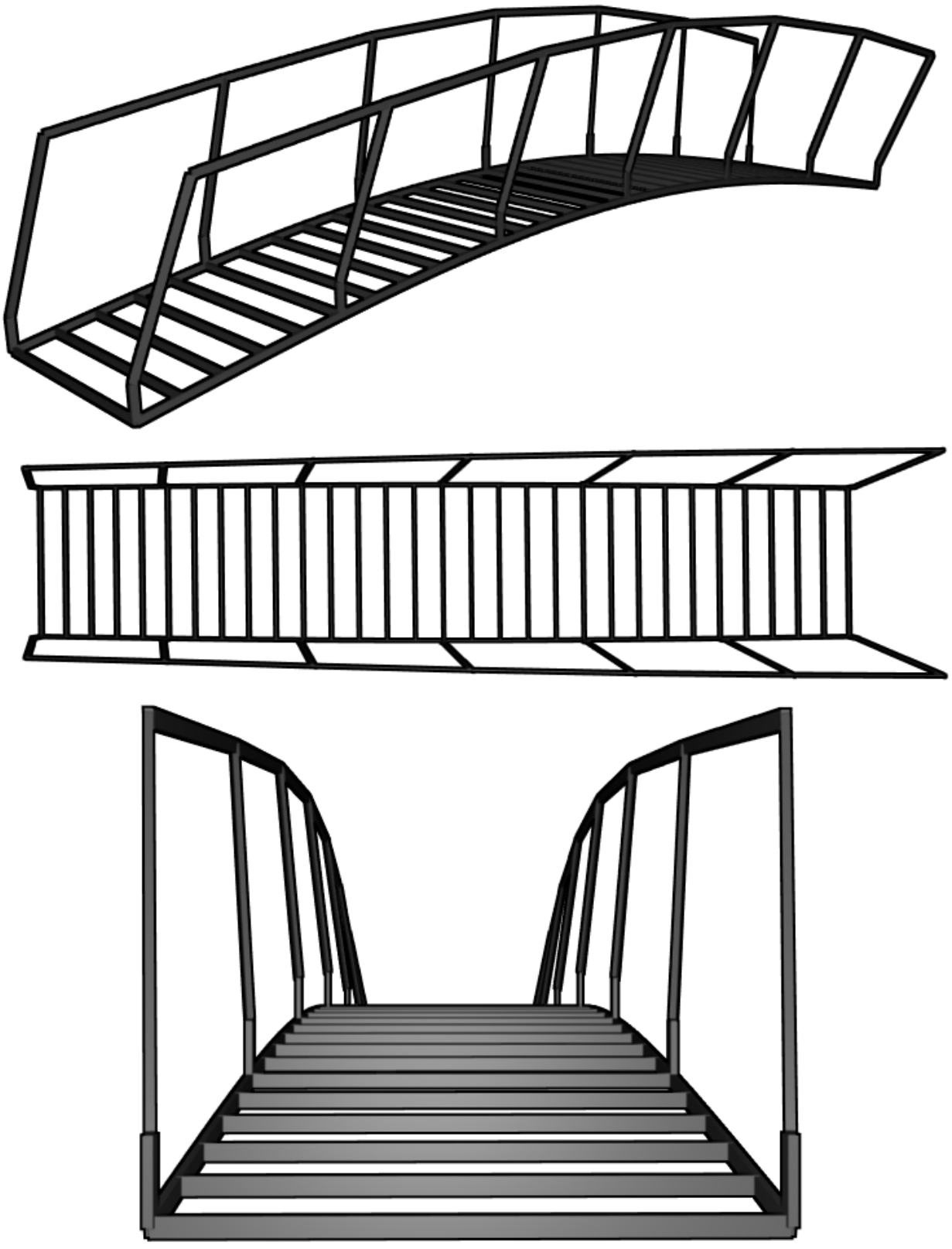


Figure 22: Simple arch bridge with vertical Vierendeel-style handrail design

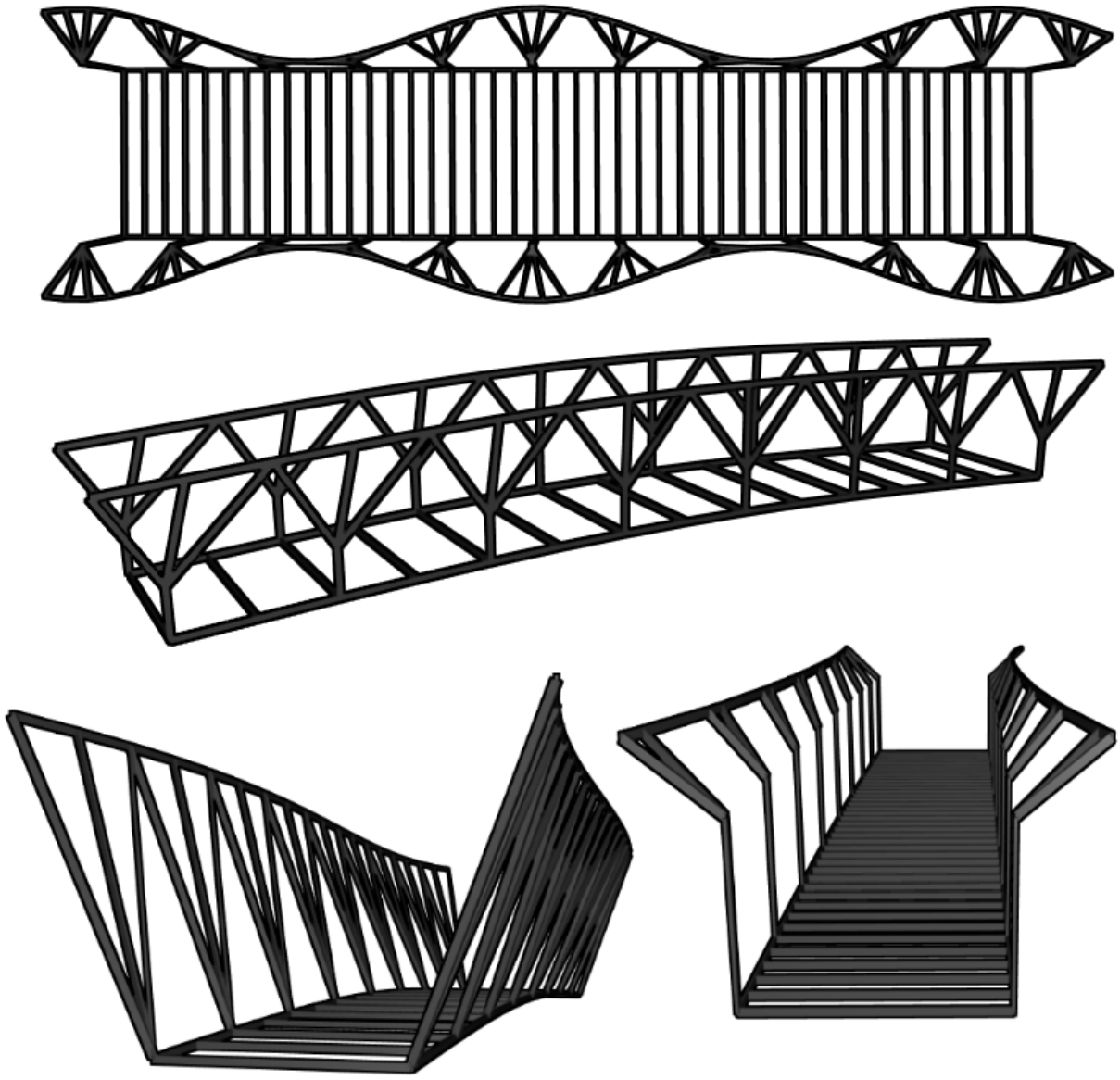


Figure 23: Variety of bridge designs

5. THE PROGRAM

5.1. The Analysis Program

The reason that major commercial analysis packages such as ANSYS and Robot were discounted was that although they provided a high degree of accuracy in their results, they have too many drawbacks:

a) They are expensive – GEVA-Blender is a free tool created by the NCRA. Its primary function is for experimentation and creation of new ideas. The structural analysis section of the program is designed to act as an add-on or plugin, i.e. a small section relative to the host program. It would be unrealistic to expect the user to pay hundreds of euro for a section of the program they may never use.

b) They require too much computing power – As an extension to point a), the file size of the GEVA-Blender program is less than 50Mb. Commercial analysis packages, however, can consume up to one hundred times this amount of storage space, and require much faster processing speeds which many users may not have access to. A smaller, more freely available program would be more appropriate.

c) They are closed-source, i.e. their coding cannot easily be adjusted or changed to suit the needs of the GEVA project. The idea is to have a seamless integration between the two sections – GEVA-Blender and the analysis package. With large commercial packages merely starting the program can take quite some time, which would impede the usability and flow of the GEVA program. A smaller program can be trimmed down to the bare essentials allowing for much faster accessing of required files.

d) They have very specific file formats – GEVA-Blender can only create a finite list of varied file types, the most appropriate of which are the CAD-oriented .dxf and .dwg files. While most packages have .dxf capability, slight variances in the scripting of the actual files themselves mean that what is observed in the Blender program is not necessarily what is observed in the analysis program. A smaller program will have a more specific file input format, but that file format is invariably easier to create and adjust than high-level .dxf's.

To this extent, a number of smaller open-source analysis programs were identified for testing; however only one such program was deemed to be appropriate for the task at hand.

5.2.SLFFEA

San Le's Free Finite Element Analysis (SLFFEA) [http://slffea.sourceforge.net/index.html] program is a small finite element analysis software package written in C and freely distributed under the terms of the GNU General Public Licence. Under these terms the source code of the original program may be altered as much as needed, and the program itself can be distributed by any means. This allows the SLFFEA program to be fully incorporated into the GEVA package and gives the GEVA developers full licence to distribute it.

Once the analysis program had been selected, it was necessary to verify the reliability and accuracy of this program. A simple test was derived to check that the results matched those of the larger packages: a similar structure was created in both a commercial package and in SLFFEA. Identical materials, loads, and fixities were used, and when compared the results fell within acceptable margins of error.

5.2.1 The test structures

Two test input files were created for SLFFEA which would create a simple 2-D and 3-D triangular truss bridges constructed from 100mm x 200mm timber members (Figure 24 and Figure 25), with a UDL of 5kN/m imposed on its base in the 2D case (Figure 24), and on the uppermost beam in the 3D case (Figure 25).

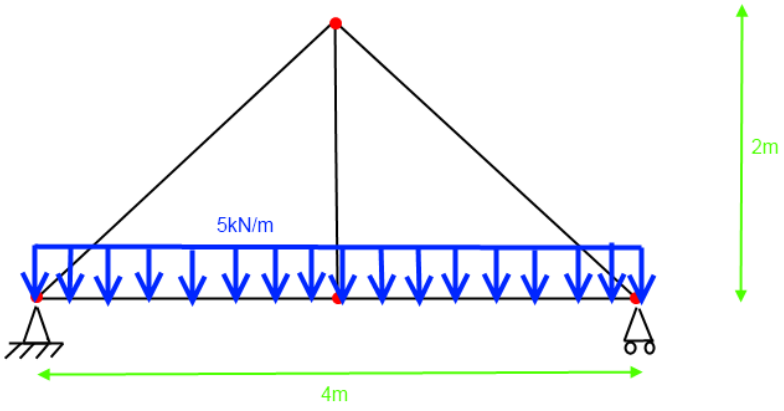


Figure 24: 2-D SLFFEA Test Structure

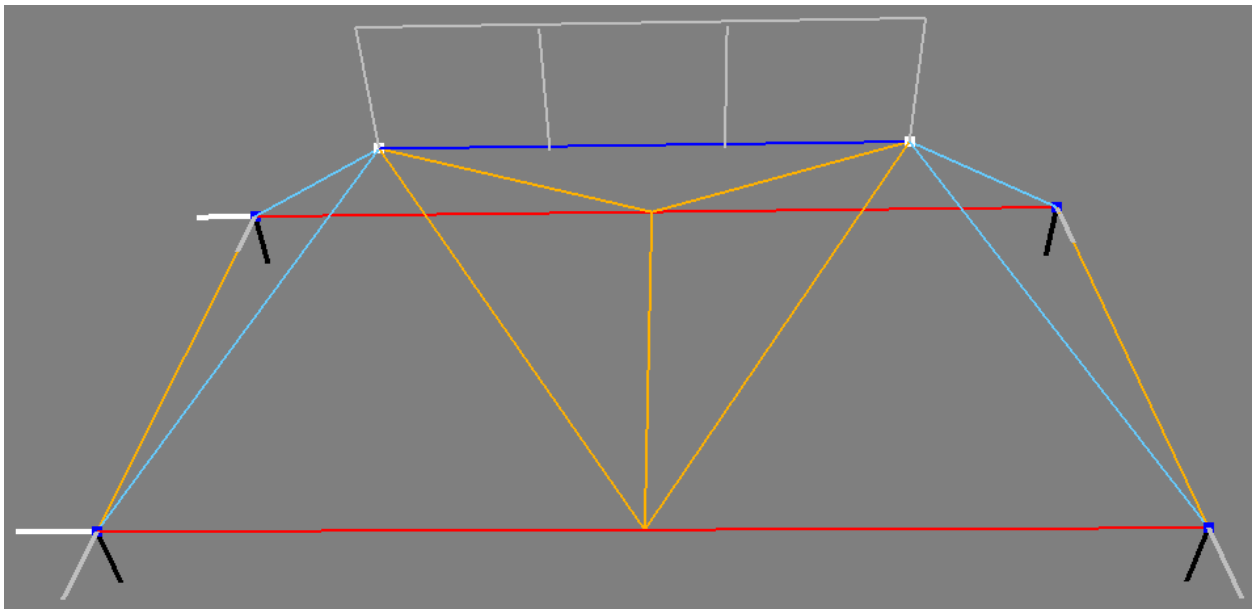


Figure 25: 3-D SLFFEA Test structure, SLFFEA XX stress results. The loading can be seen on the uppermost bar, while the restraining directions can be seen at the four extreme corners of the structure: pinned at one end and rollers at the other.

These structures were then copied in the commercial package STRAP, and both were analysed. Table 1 & Table 2 show results from corresponding bars in both structures, with maximum tension and compression values for both programs listed.

	SLFFEA	STRAP
Max Tension	0.477 N/mm^2	0.425 N/mm^2
Max Compression	0.335 N/mm^2	0.295 N/mm^2

Table 1: Comparison of SLFFEA and STRAP results, 2-D case

	SLFFEA	STRAP
Max Tension	0.142 N/mm^2	0.139 N/mm^2
Max Compression	0.332 N/mm^2	0.31 N/mm^2

Table 2: Comparison of SLFFEA and Strap results, 3-D case

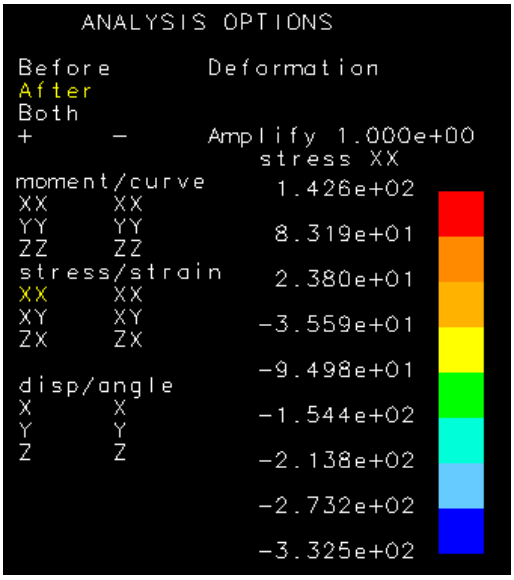


Figure 26: XX Stress values for structure in Figure 25, from SLFFEA

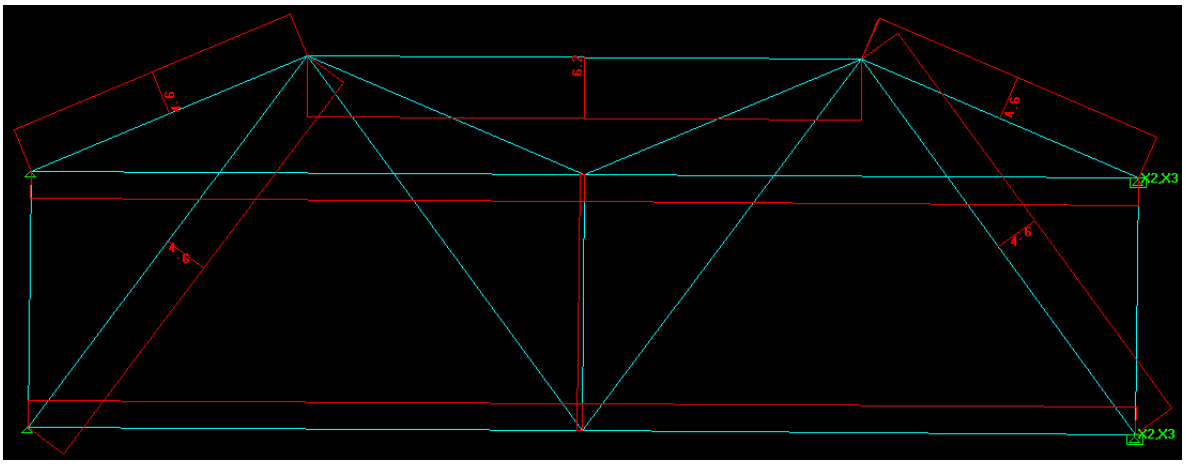


Figure 27: Axial force results for 3-D truss structure as calculated in STRAP. Results in kN.

These results prove conclusively that SLFFEA is a reliable program for the task at hand, with comparable analysis results to commercially available and widely used software.

5.3.Storing Information from GEVA

The way the GEVA program operates is that at the start of each generation a number of individuals are created (in the case of the bridge grammar these individuals are bridges). From these individuals, GEVA then builds the structure one beam at a time. In order to store the data for these beams so that it is accessible to the analysis program, a list of all of these beams needs to be created and then stored in a separate file for each individual. A highlighted problem with GEVA, however, is that for each successive individual, it retains all knowledge of all aspects of previous individuals, i.e. the first individual will contain nodes numbered 1 to 300, the second individual will contain nodes numbered 1 to 300, *and* nodes 301 to 600, the third will likewise contain all nodes from 1 to 900, and so on. This also holds true across generations, which quickly leads to a situation where any later individuals will retain tens of thousands of nodes. As the inventory builds with each new individual, this gives rise to an ever increasing latency effect, with GEVA taking more and more time to create new individuals. In order to minimise this latency, a method has been devised which will count the number of total nodes created so far, and will subtract any which have previously occurred (in previous individuals or generations) from the current list of nodes. This ensures that for each new structure, only the relevant nodes are stored. The files that had been written containing the full list of nodes are then over-written with only the essential information.

5.4. Building the Input File (See Appendix 1: analysis.py)

The input file types of the SLFFEA program lend themselves very well to the GEVA method of creating structures, and enable Python programs to be written which will extract the relevant data from the GEVA and Blender code and use it to build a file. In essence, what is required is:

a) A numbered list of nodes, each one consisting of a co-ordinate in xyz space

e.g. 3 0.0 10.5 4.5

This represents node number 3, which has co-ordinates (0, 10.5, 4.5).

b) A numbered list of beams, each one consisting of the indexes of the nodes the beam connects,

e.g. 7 2 4

This represents beam number 7, which connects nodes numbers 2 and 4.

c) The node numbers which will act as fixing points and the directions in which they will provide fixities (i.e. (x, y, z) translation, (x, y, z) rotation).

d) The nodes or elements upon which the load is to be applied, and the nature of that load (i.e. (x, y, z) components and moments).

In the Python programming language, the method GEVA uses to create its elements is a function called “MakeBoard(x0, y0, z0, x1, y1, z1)”. When this function is called (every time a structure is created the function is called for each individual member in the structure), a beam is created that joins up points (x0, y0, z0) and (x1, y1, z1) (Figure 28).

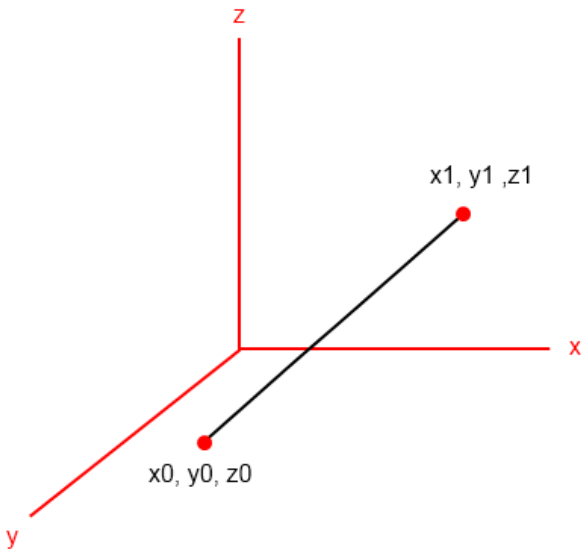


Figure 28: Beam in x-y-z space

What needs to be done to store these points is to create two lists: the list of beams (beamslist) and the list of nodes (nodeslist) (see Appendix 1: Analysis_2.py). Lists are indexed within Python, so there is only need to obtain the actual nodes themselves; these can be referenced after the fact by citing the index of the list (for example: nodeslist[3] will return the fourth item on the list of nodes, namely node number 3). Both Python and SLFFEA start indexes from 0 rather than 1 (they are written in similar languages which share a common architecture) meaning that the fifth index will be numbered 4, the sixth will be numbered 5, and so on.

A more demanding task is that of automating the selection of both the horizontal walkway elements and the fixing points in each structure. An original test program was written which would perform all the same functions as the "MakeBoard" function that GEVA would call (i.e. it would create a series of beams by specifying two points, and would then compile a list of those points in both a list of nodes and a list of beams). However, in this test program it would then require some user input to ascertain the desired fixing points of the structure, along with the load to be added and the elements upon which that load was to act. With the integrated GEVA-Blender program, the entire process is automatic, requiring minimal input by the user. To that extent, a more complicated code was required to 'search' for the extreme four corners of each structure GEVA creates (these points are not fixed on every structure, they vary from individual to individual, necessitating the construction of an algorithm that would search for the specific points).

These four points would be when:

x is minimum, y is maximum, z is minimum
x is minimum, y is minimum, z is minimum
x is maximum, y is maximum, z is minimum
x is maximum, y is minimum, z is minimum

Yet more complex still is the determination of which beams comprise the walkway of the bridge. For pedestrian loading simulations a UDL of $5kN/m^2$ is applied across the deck (in accordance with BS EN 1995 – 2: 2004, Eurocode 5: Design of timber structures - Part 2: Bridges), with the loaded elements being listed in a similar style to the beams list:

Element no.	Beam load in y dir	beam load in z dir
4	0	-5.0

This example represents element 4 having a uniformly distributed load of 5 (input files are devoid of units, as long as consistent units are used throughout then answers will be correct and as expected) acting in the negative z direction (downwards), with no load in the y direction. For determination of the beams in the deck GEVA was told to search for any element where the x and z components were identical (meaning that the beam only varied along the y-axis, which means it is horizontal) and to add that beam to the list of loaded elements. Once this list had been compiled, the two extreme members (those with the minimum x value and the maximum x value) would be the members whose nodes would be affixed to the ground. One end of the bridge is set to pinned, with restraints in the x, y, and z directions (but no rotational restraints), while the other end of the bridge is pinned, with no restraint in the x-direction, allowing for expansion of the bridge.

5.5. Analysing an Individual

The Blender GUI has been redesigned to accommodate the analysis tools. With usability in mind, two simple buttons have been added: An "Analyse" button, and a material selector in a drop-down menu (Figure 29).

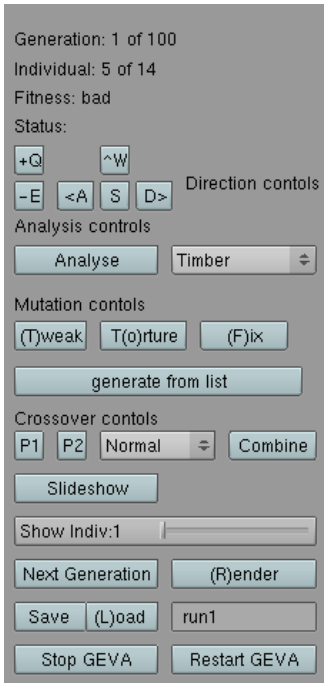


Figure 29: Redesigned Blender GUI

In order to analyse an individual, the user simply selects a material (timber or steel are currently the only two options, with timber being the default choice) and clicks the "Analyse" button. Once clicked, this initialises the analysis.py program (see Appendix 2 for a comprehensive breakdown of the program), which in turn assembles the correct lists of nodes and beams, finds the elements which compose the horizontal walkway, finds the four corners of the bridge (and sets them as fixing points – one side being pinned and the other side resting on a roller), builds the input file for the SLFFEA program, runs the analysis on the file itself, opens up the analysis report if the analysis was run successfully, and reports back on the analysis to the user, stating whether or not any of the elements in the structure fail in tension or compression.

5.6.Reading the Output file (See Appendix 2: analysis.py)

When SLFFEA successfully runs an analysis it saves a copy of the analysis results as a separate file. This file is in the same format as the original input file, except with displacements and stresses already present in the structure (e.g. the structure now carries a state of pre-stress; it is possible to re-use this file as an input file again, enabling non-linear analysis to be performed). It is then possible to search through this file with Python and retrieve the pertinent data for the structure – the internal stresses in the members. The program can then scan these results and ascertain whether or not pre-defined stress limits have been reached or not, and the user is then alerted if any section of the bridge fails.

6. FINDINGS

When a beam is loaded, a stress distribution is set up, with tension (presented in red in this project) in the bottom fibres and compression (presented in blue in this project) in the top fibres (Figure 30).

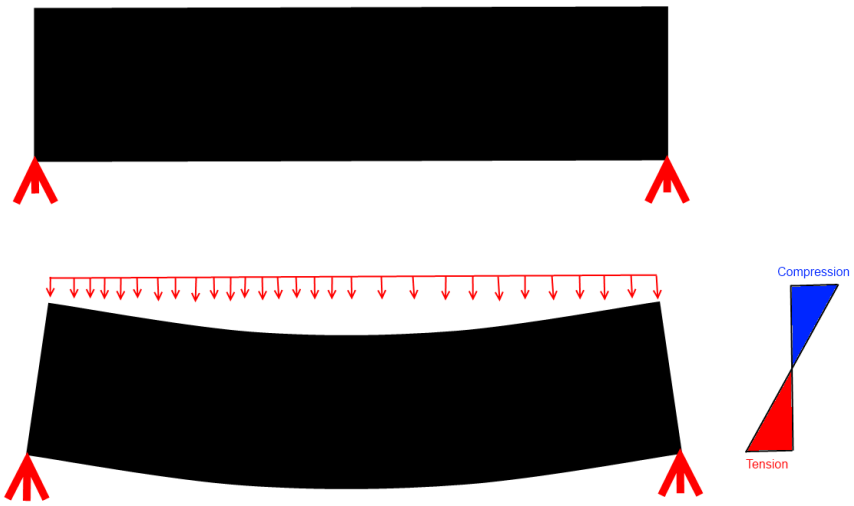


Figure 30: Stress distribution across a beam

With a beam bridge, the same is true, as can be seen in Figure 42, Figure 51 and Figure 52. With the GEVA bridge designs, the handrails are an integral part of the structure of the bridges, and variations in the designs can have interesting results. The bridges can be broadly classified into five main domains:

1. "Linear" designs can be termed as GEVA designs in which the uppermost member of the handrail is of a linear type, e.g. Figure 22, Figure 23 (middle, bottom left, bottom right), Figure 31.

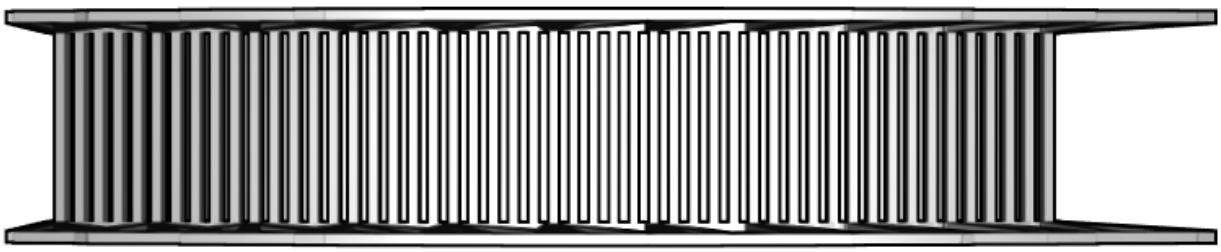


Figure 31: Linear bridge design

2. “Wave” designs can be termed as GEVA designs in which the uppermost member of the handrail follows a variation of a sine wave, as in Figure 20, Figure 23 (top), and Figure 32.

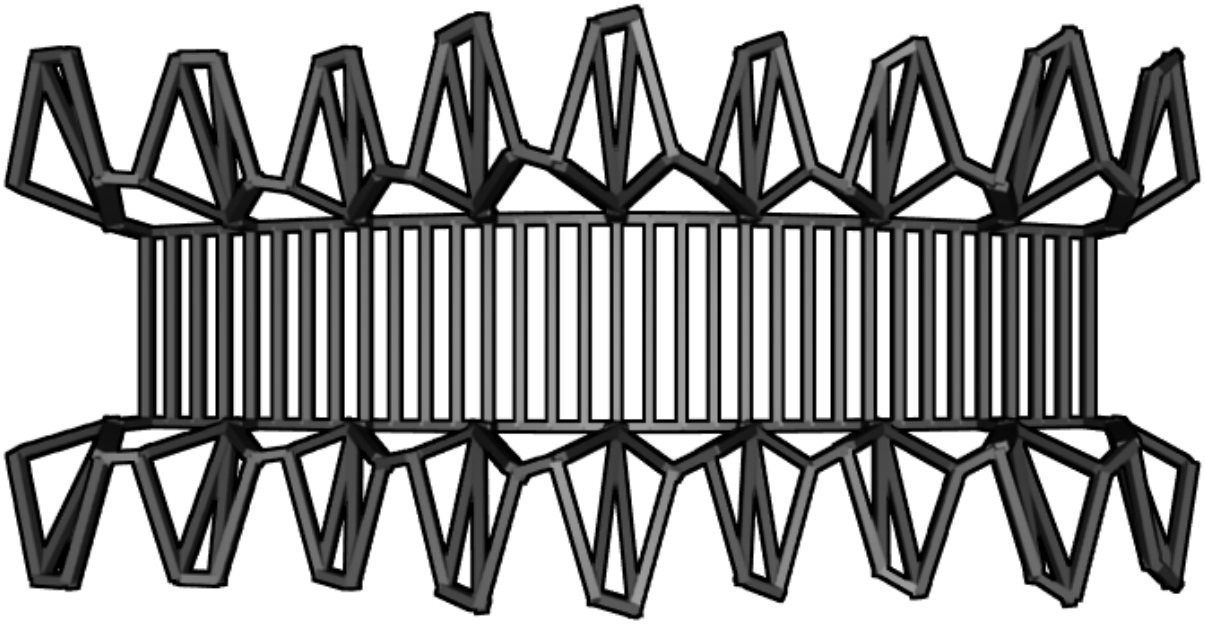


Figure 32: Wave bridge design roughly based on Sine wave

3. “Truss” designs do not necessarily have an uppermost member in the handrail, but rather an interconnecting mesh of triangulated members which form a rudimentary truss-type structure in the handrail, e.g. Figure 33.

Figure 33: Truss bridge design

4. “Ribcage” structures are a variation on the “Wave” designs, and appear to mimic the ribcage of a whale, e.g. Figure 34, Figure 35.

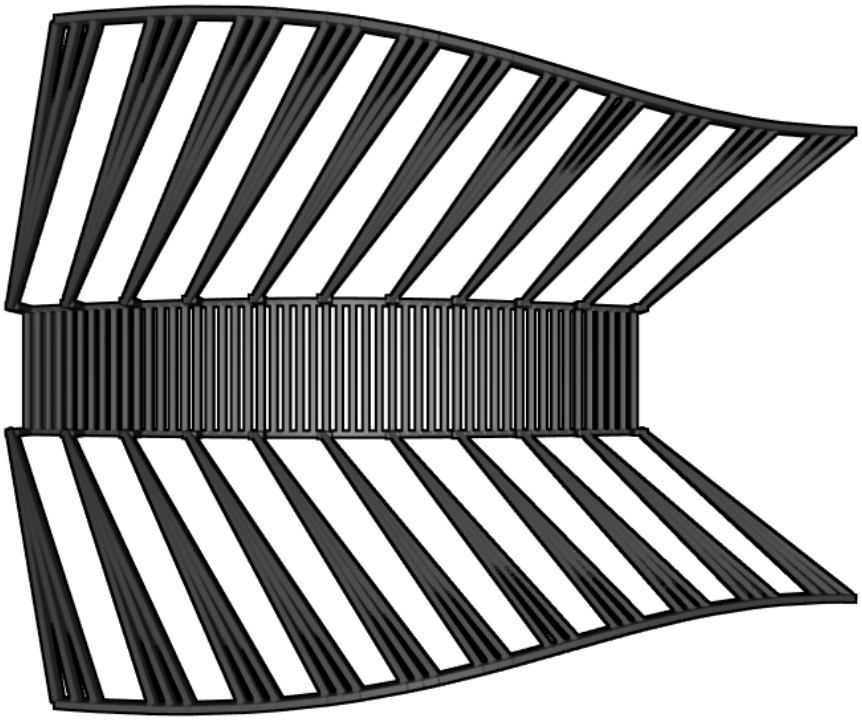


Figure 34: Wide ribcage design

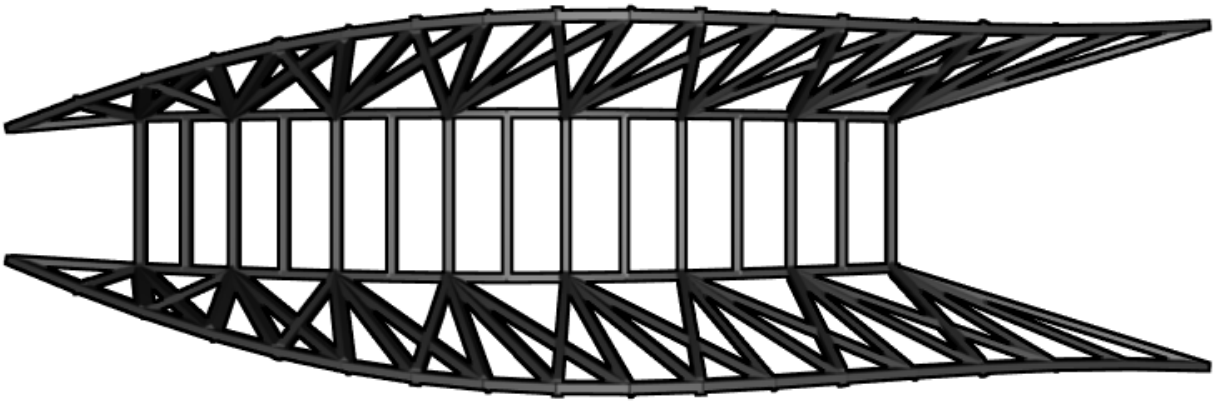


Figure 35: Narrow ribcage design, incorporating truss-style uprights

5. “Random” designs do not appear to have a defined structure to the handrail member orientation, and follow many different patterns, varying between the ordered random pattern (e.g. Figure 36, and the wildly disorganised (Figure 37).

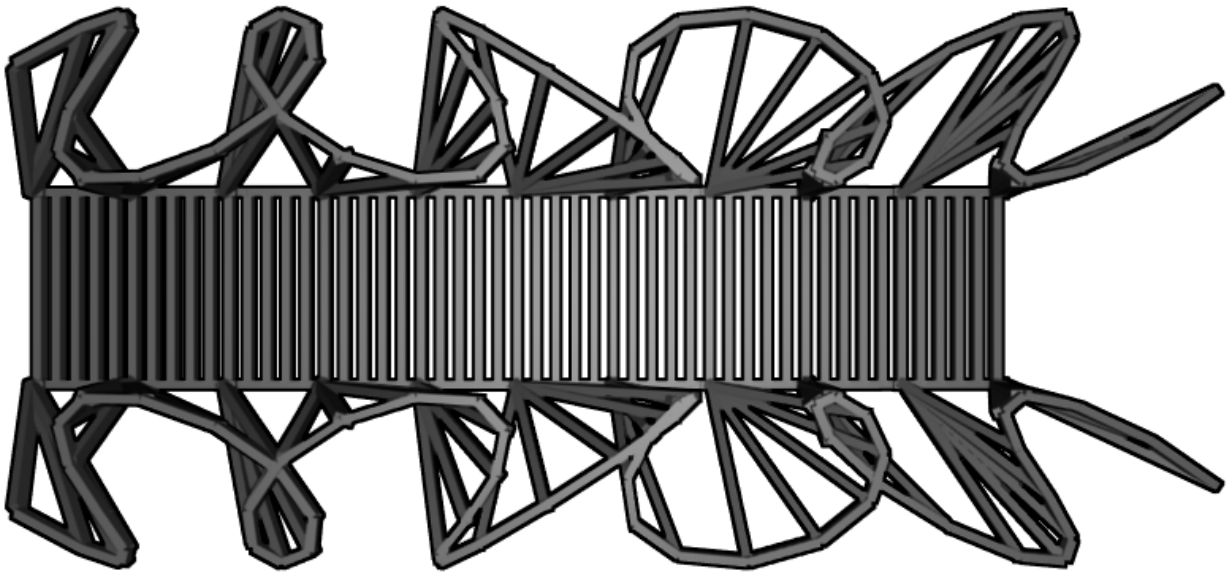


Figure 36: Random organised bridge design with epic handrails

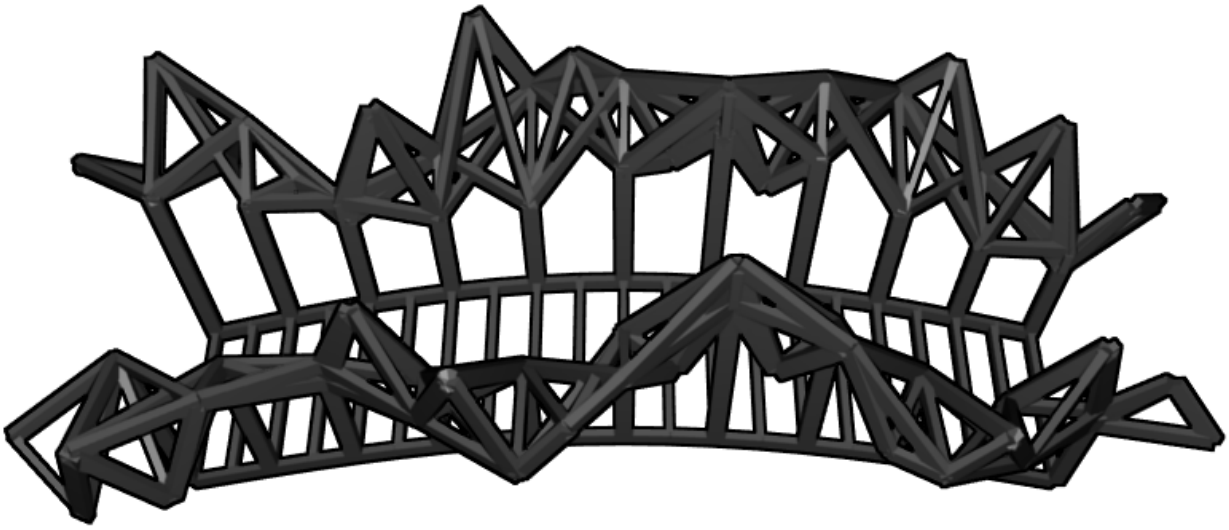


Figure 37: Random disorganised bridge design

Aside from handrail types, GEVA has many other variable parameters, all of which have a measurable impact on the structural rigidity of the bridges. The handrails are made up of three main parts (which GEVA can choose to use or not):

1. The vertical stay
2. The branches
3. The handrail / upper member

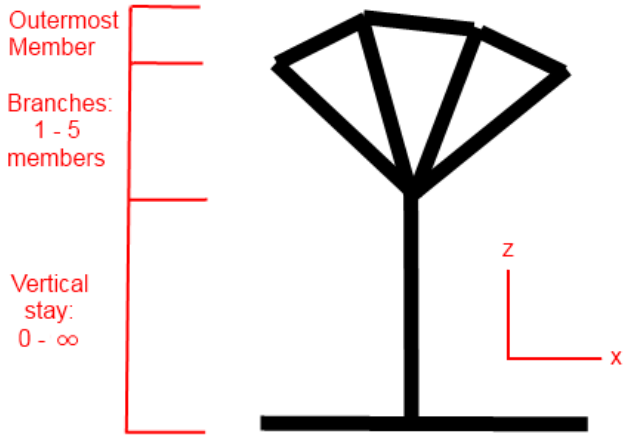


Figure 38: Bridge handrail components, indicating variable parts

The vertical stay can vary in height from bridge to bridge (Figure 38), from 0m (not in the bridge design, e.g. Figure 35), to an enormous height (Figure 42), and is always orthogonal to the x-y plane. The branches (Figure 38) can vary in number from 1 (Figure 44) to 5 (Figure 33), and can vary in angle with respect to the vertical stay from both the x-axis (angled out from the bridge at a 90^{o} angle) and from the y-axis (angled towards one end of the bridge, Figure 39), or a combination of both (Figure 34). Finally, the outermost member connects the ends of all the branches together.

The optimal angle of the branches is considered to be symmetrical about the vertical stay (as described in Figure 39 and can be seen in Figure 33), which allows for equal stress paths along the length of the bridge in both directions, lowering the stresses.

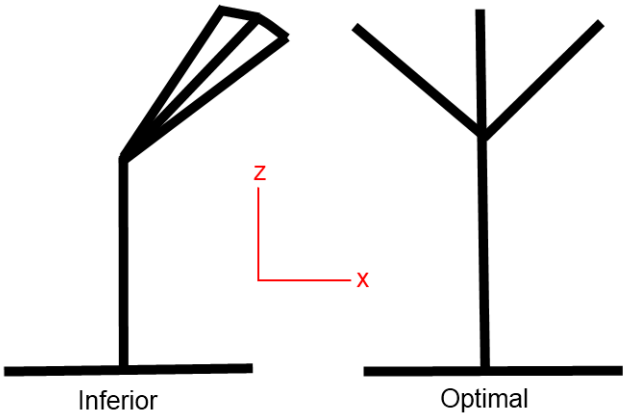


Figure 39: Optimal branch angles (on right) are equal on all sides, rather than offset (on left)

What is also readily visible is that most of the bridge designs (e.g. Figure 20, Figure 21, Figure 23, Figure 34 and Figure 35) have handrail branches that are angled out from the edge of the walkway itself, i.e. greater than 90 degrees (Figure 40).

Figure 40: Taking a transverse cross-section of a bridge, the section on the right has a handrail offset at a greater angle than the one on the left

While having angled handrails aids in lateral stability of the bridges by increasing the second moment of area of the bridge in the yz-direction, it lowers the compressive capacity of the top of the handrail by lowering the second moment of area of the bridge in the xz-direction, thus increasing the stresses in the fibres. In order to compensate for the loss in relative depth of the bridge cross section, the height of the handrails needs to be increased in proportion with the width of the bridge.

6.1. Bridge Restraint Types: Pinned - Roller

Two fixing situations were analyzed in this project, pinned-roller and pinned-pinned bridges. Due to the variable arch value in the bridges, some bridges operate more efficiently with a roller support at one end, while highly arched bridges maximise efficiency with full translational restraint. The pinned-roller bridge results are of more analytical interest, as the design of the entire bridge (including the handrail designs) has structural implications, with the deck operating in tension and the upper handrail sections operating in compression. For future work with the GEVA-Blender program, a highlighted topic for further study is the inclusion of structural analysis results in the fitness function of the evolutionary program. To this extent, the preferred restraint method would be pinned-roller, as the design of the overall bridge would have more of a bearing on the stress state of the bridge.

For pinned-roller bridges, analyses were run on over 50 varying bridge types, with interesting conclusions. The bridge analysis results can be classified into five sections, relating to the stress state of the bridges:

1. Ultra Low-Stress — Less than 6 N/mm^2
2. Low-Stress — 6 N/mm^2 – 10 N/mm^2
3. Medium-Stress — 10 N/mm^2 – 14 N/mm^2
4. High-Stress — 14 N/mm^2 – 18 N/mm^2
5. Failure — Above 18 N/mm^2

6.1.1 Ultra Low-Stress Bridges

The height of the vertical stays of the bridges has the most tangible effect on the stress state of the bridges. In all four presented cases (Figs. 41 – 44), the vertical stays are enormous, to the point where in Figure 42 GEVA has evolved a form of suspension bridge, with the two end members acting in compression and all other members acting in tension to help carry the bridge's load. This case can also be seen in Figure 41, Figure 43 and Figure 44, but with less pronounced compressive and tensile forces.

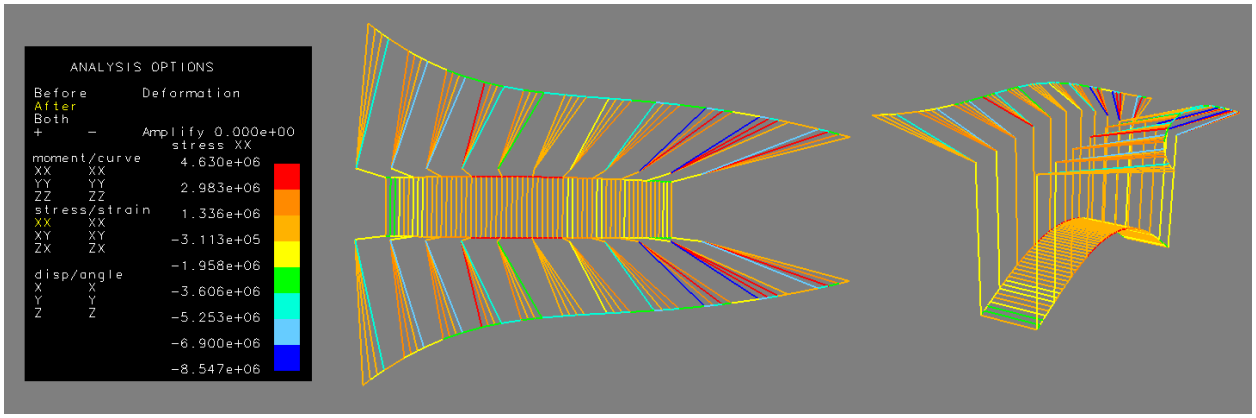


Figure 41: Ultra low-stress bridge with maximum tensile stress of 4.63 N/mm^2

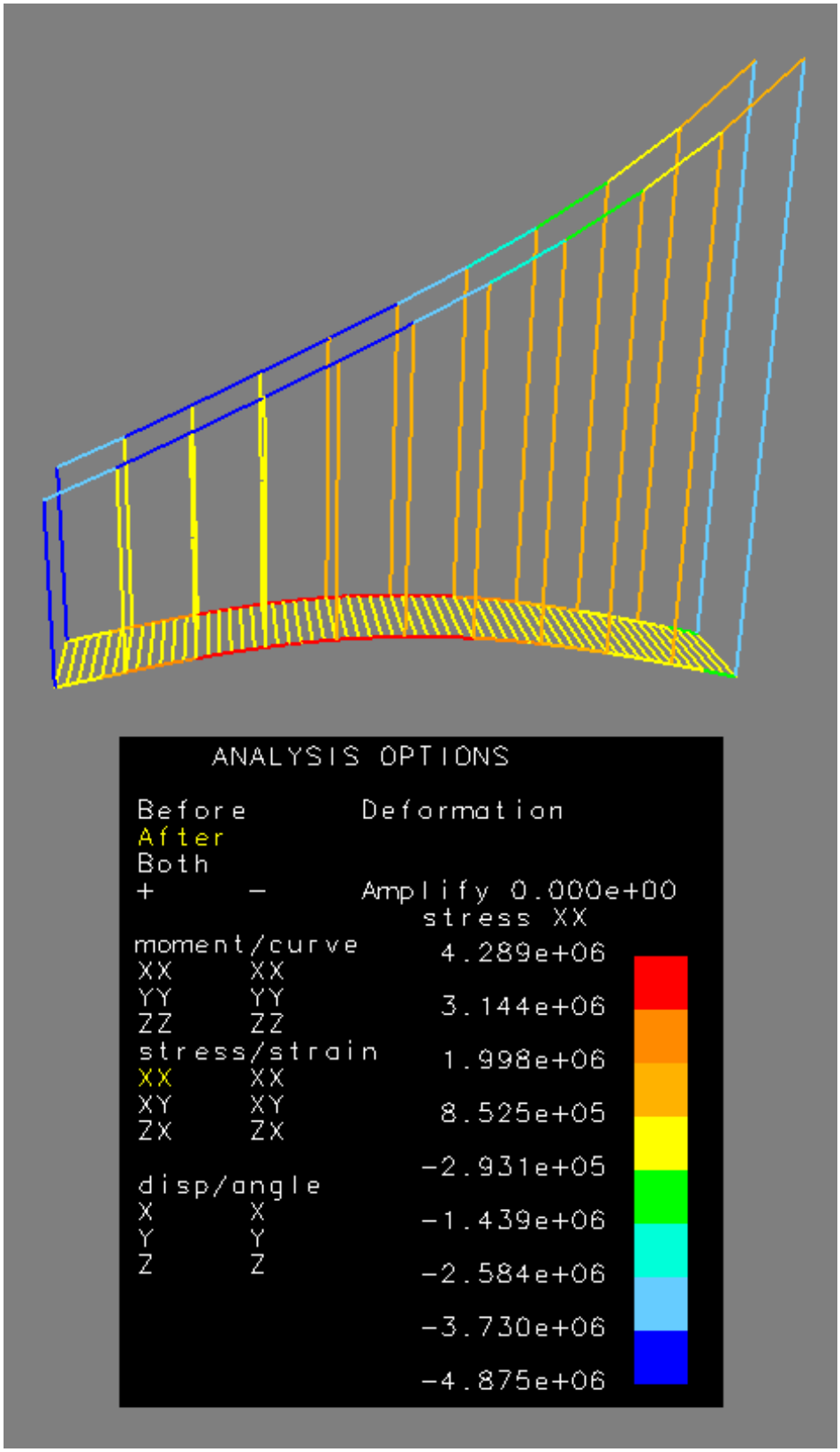


Figure 42: Ultra low-stress bridge with maximum tensile stress of 4.289 N/mm^2

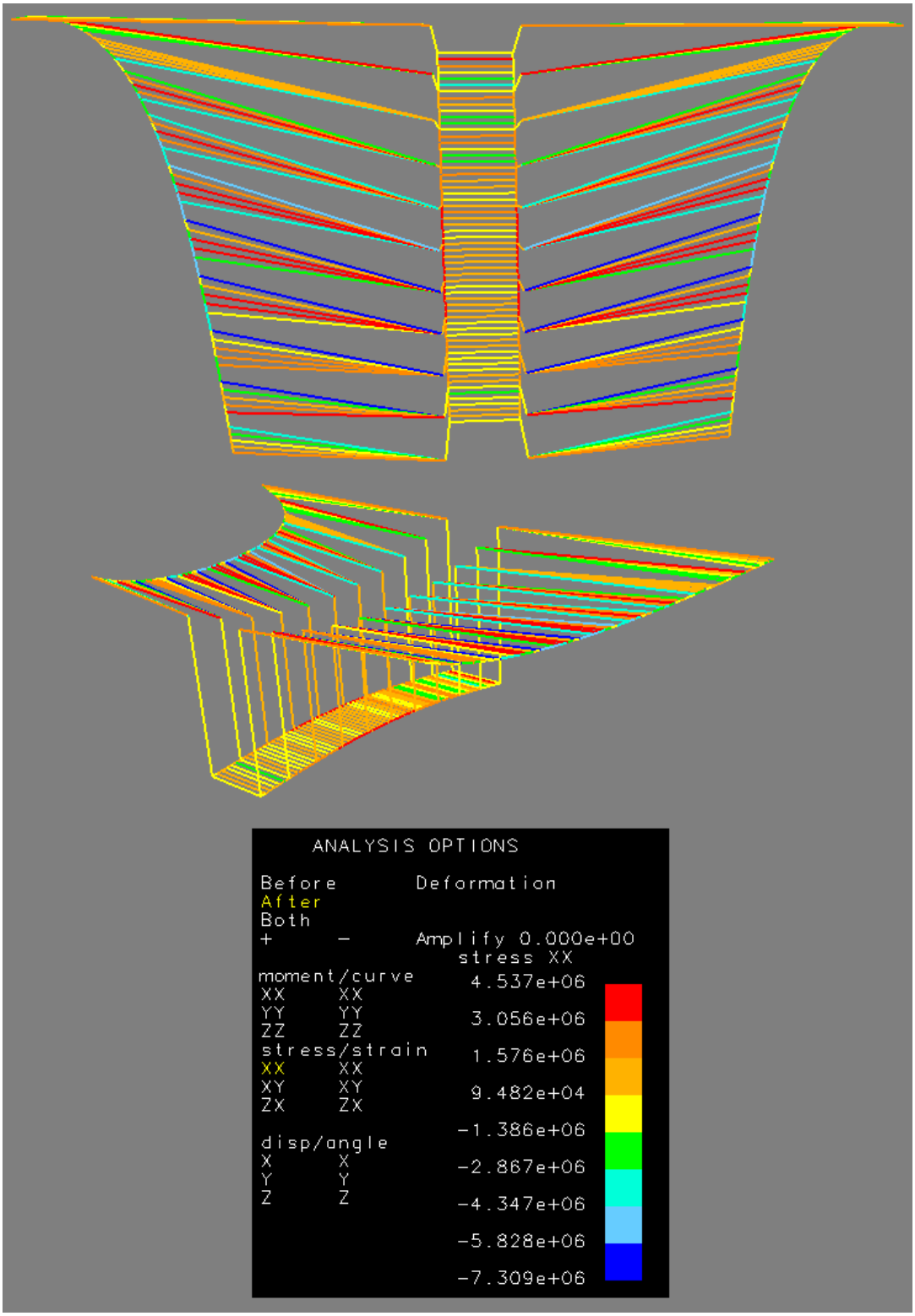


Figure 43: Beautiful and efficient ultra low-stress bridge with maximum tensile stress of 4.537 N/mm^2

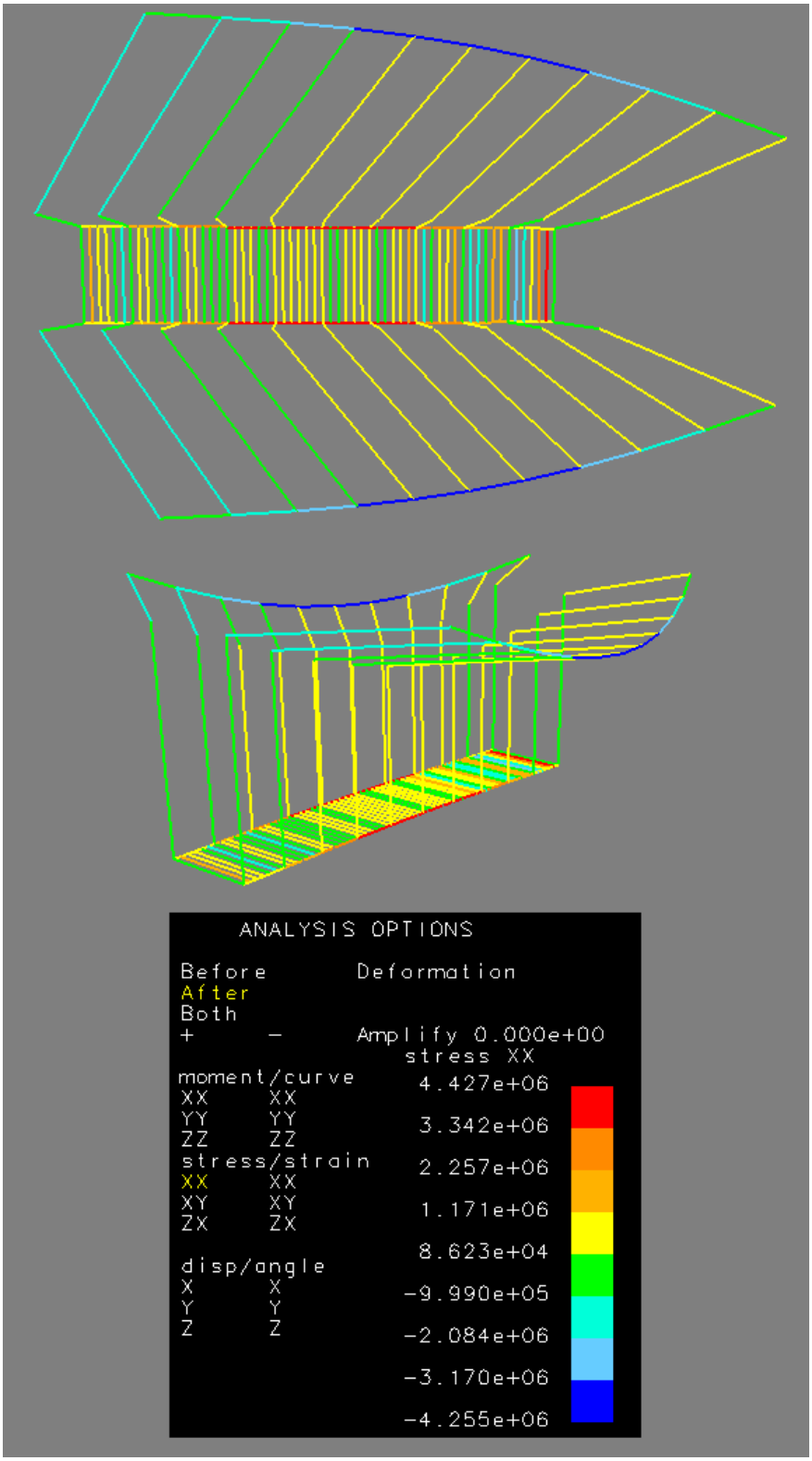


Figure 44: Ultra low-stress bridge with maximum tensile stress of 4.427 N/mm^2

6.1.2 Low-Stress Bridges

Continuing on from the Ultra Low-Stress Bridges, the height of the vertical stays can still be seen to be playing a vital role in the stress state of the bridges. What is noticeable with these bridges over the ultra low-stress ones is that the vertical stays (or height of the handrails / branches in Figure 46 and Figure 49) are smaller, thus placing more stress on both the upper handrail section and the walkway itself.

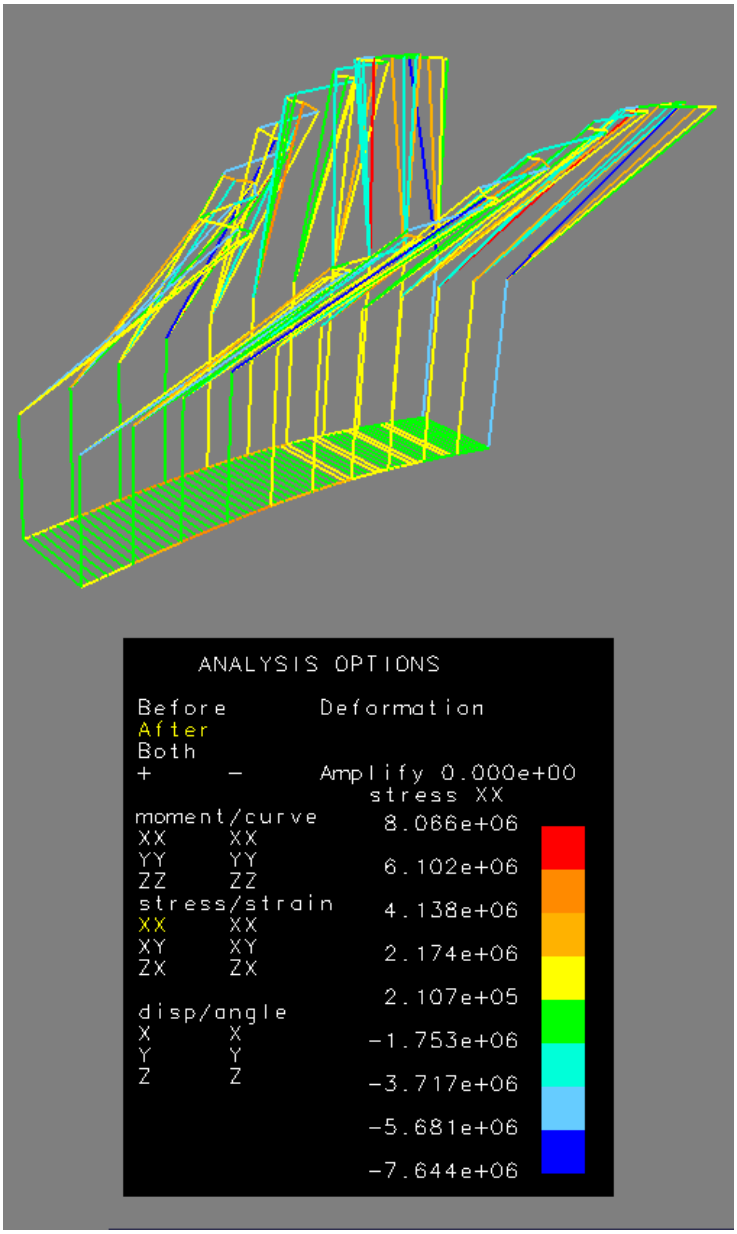


Figure 45: Low-stress bridge with maximum tensile stress of 8.066 N/mm^2

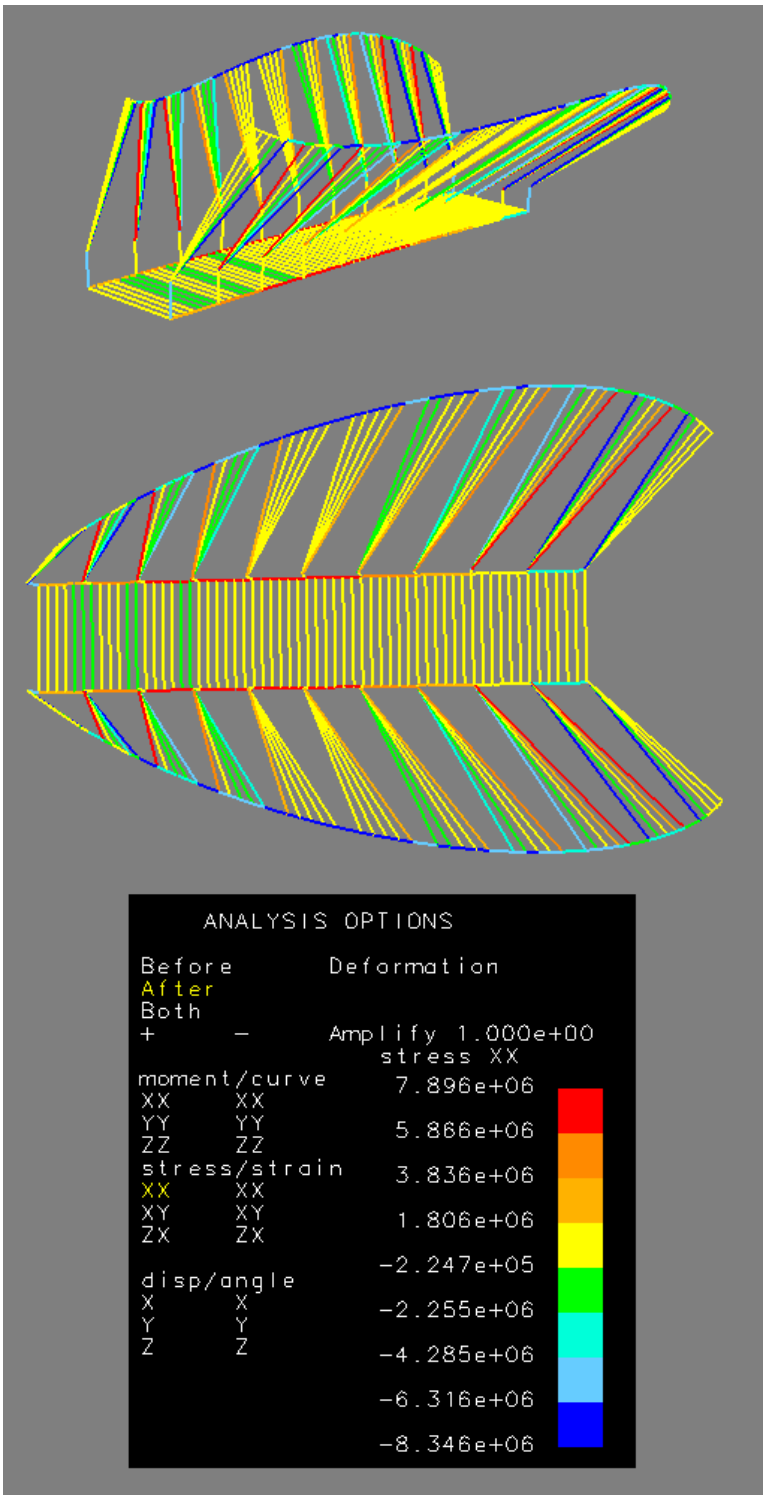


Figure 46: Low-stress bridge with maximum tensile stress of 7.896 N/mm^2

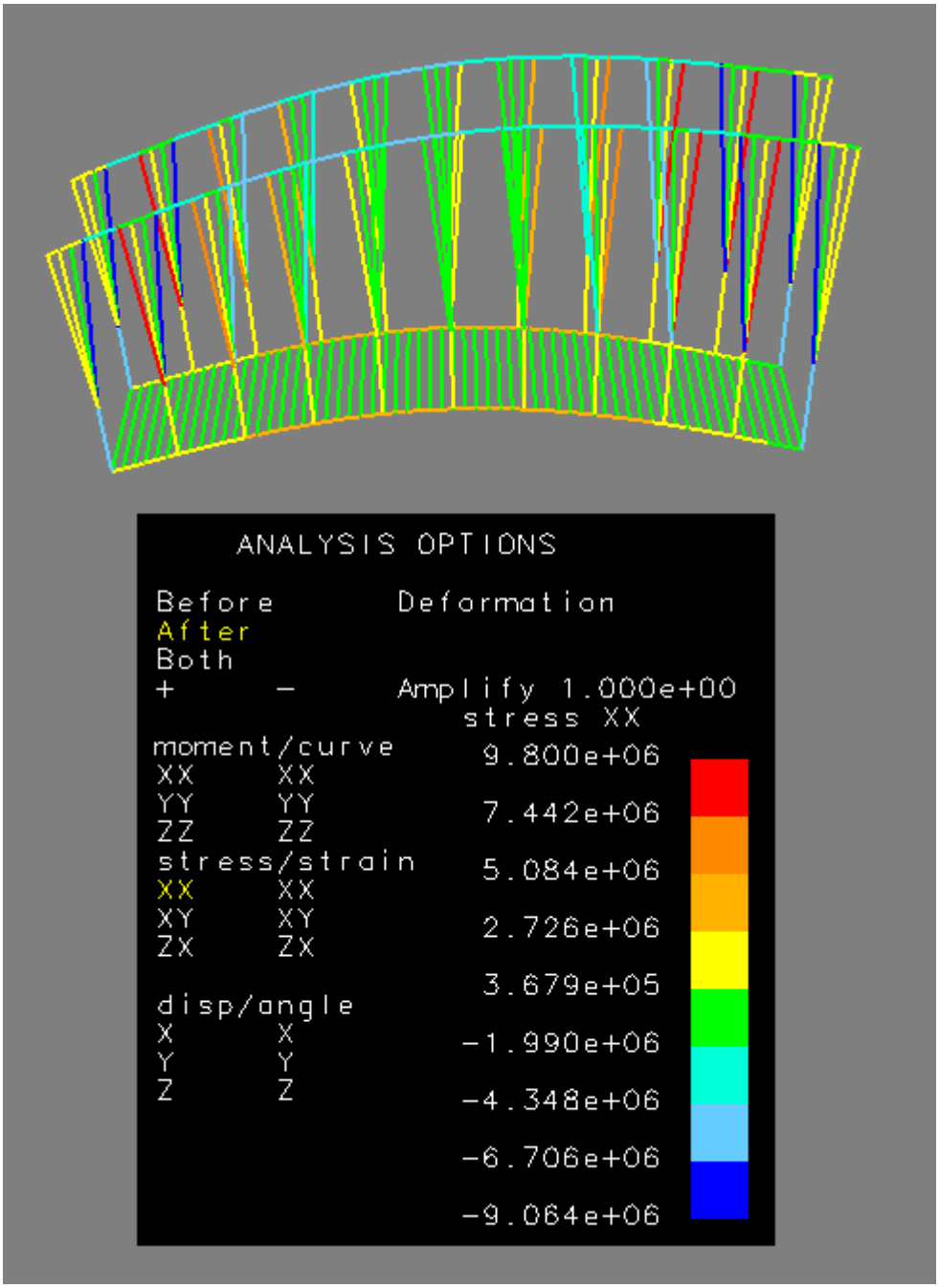


Figure 47: Low-stress bridge with maximum tensile stress of 9.8 N/mm^2

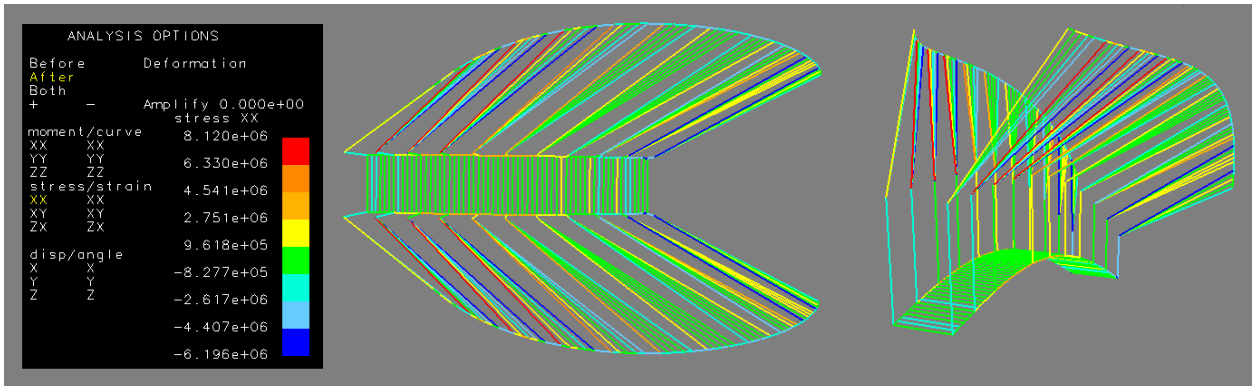


Figure 48: Low-stress bridge with maximum tensile stress of 8.12 N/mm^2

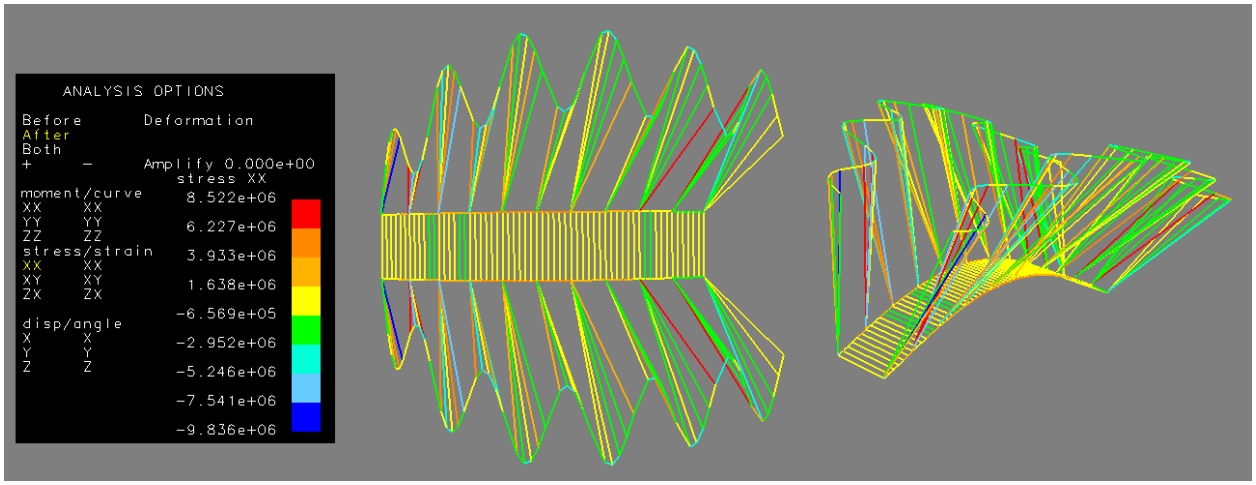


Figure 49: Low-stress bridge with maximum tensile stress of 8.522 N/mm^2

6.1.3 Medium-Stress Bridges

In the continuing vein, the medium-stress bridges have yet smaller vertical stays, and are beginning to look more like regular bridges. The majority of bridge designs in GEVA fall within this category, with a varied selection presented here. What is interesting to note, however, is that the less "radical" the bridge design becomes, the greater the stress within the bridge...

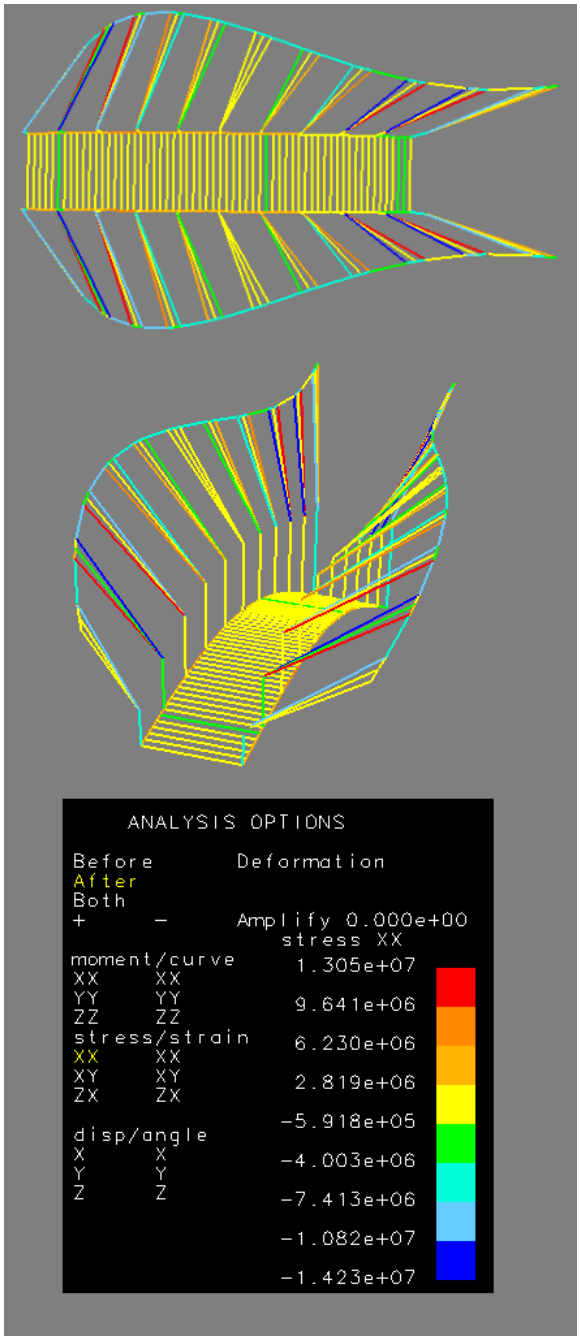


Figure 50: Medium-stress bridge with maximum tensile stress of 13.05 N/mm^2

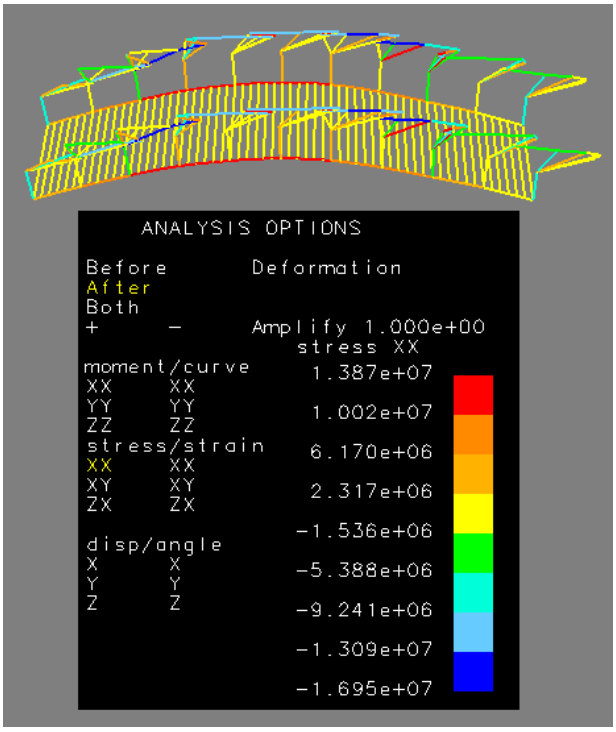


Figure 51: Medium-stress bridge with maximum tensile stress of 13.87 N/mm^2

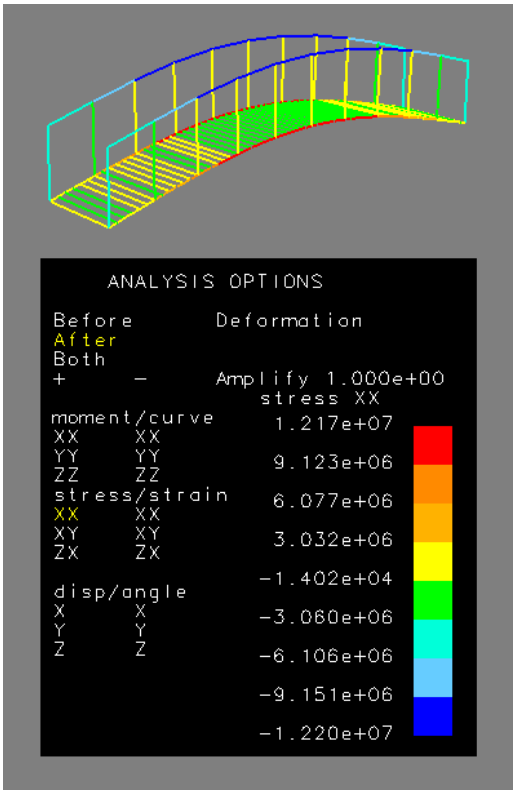


Figure 52: Medium-stress bridge with maximum tensile stress of 12.17 N/mm^2

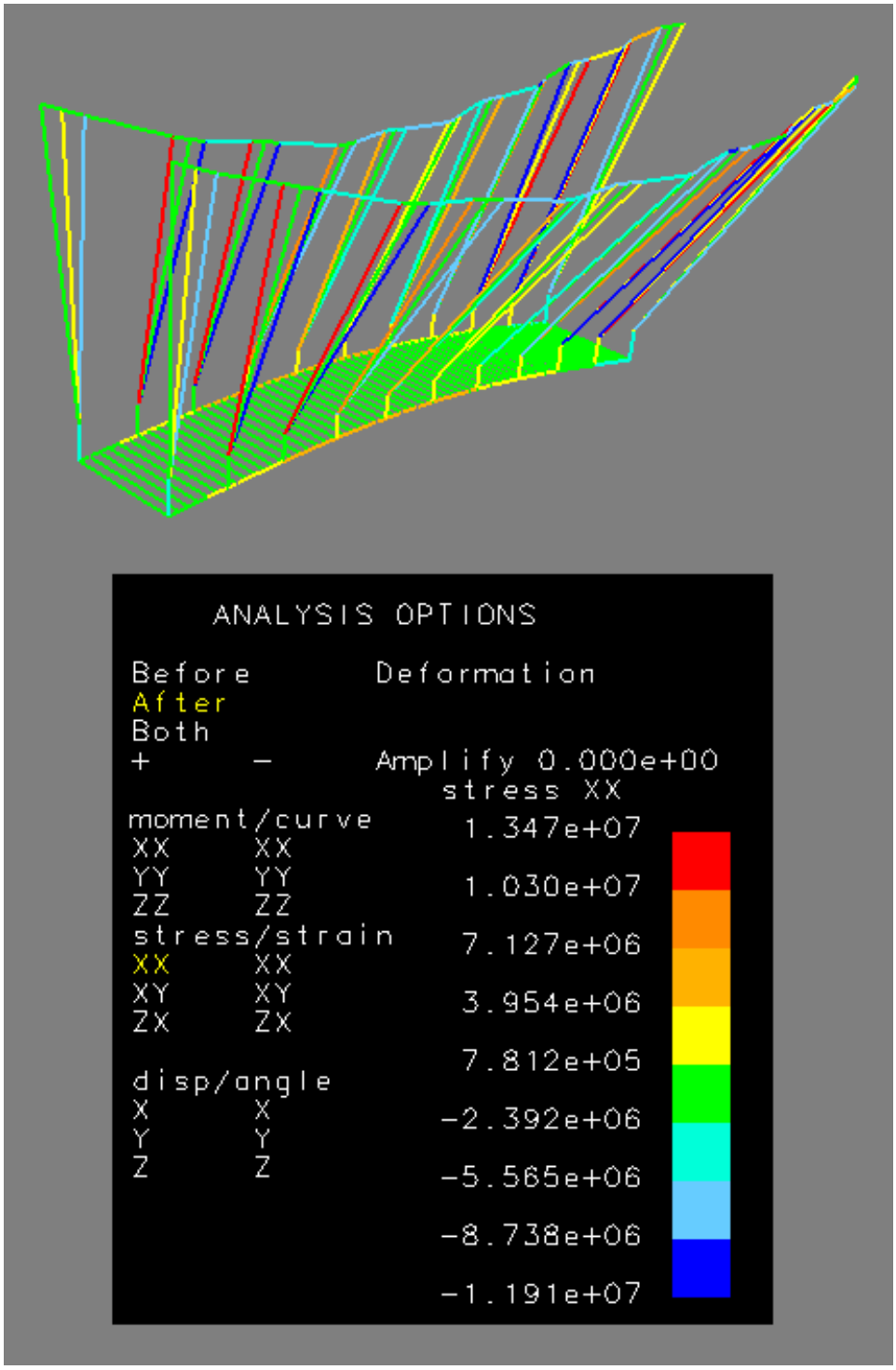


Figure 53: Medium-stress bridge with maximum tensile stress of 13.47 N/mm^2

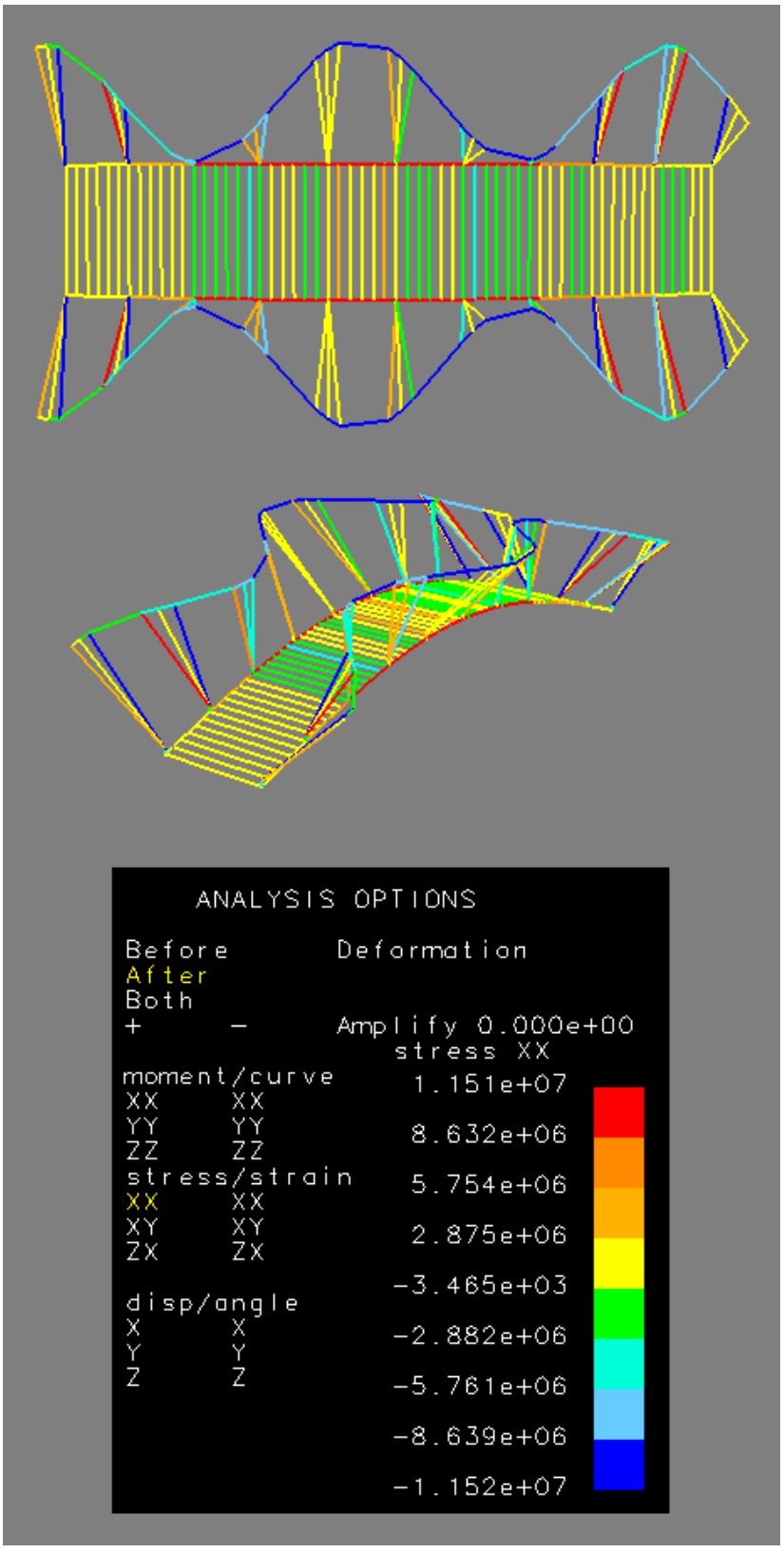


Figure 54: Medium-stress bridge with maximum tensile stress of 11.51 N/mm^2

6.1.4 High-Stress Bridges

These bridges are close to the material limits, but in most cases still feasible. Vertical stays are lower than those in 6.1.3.

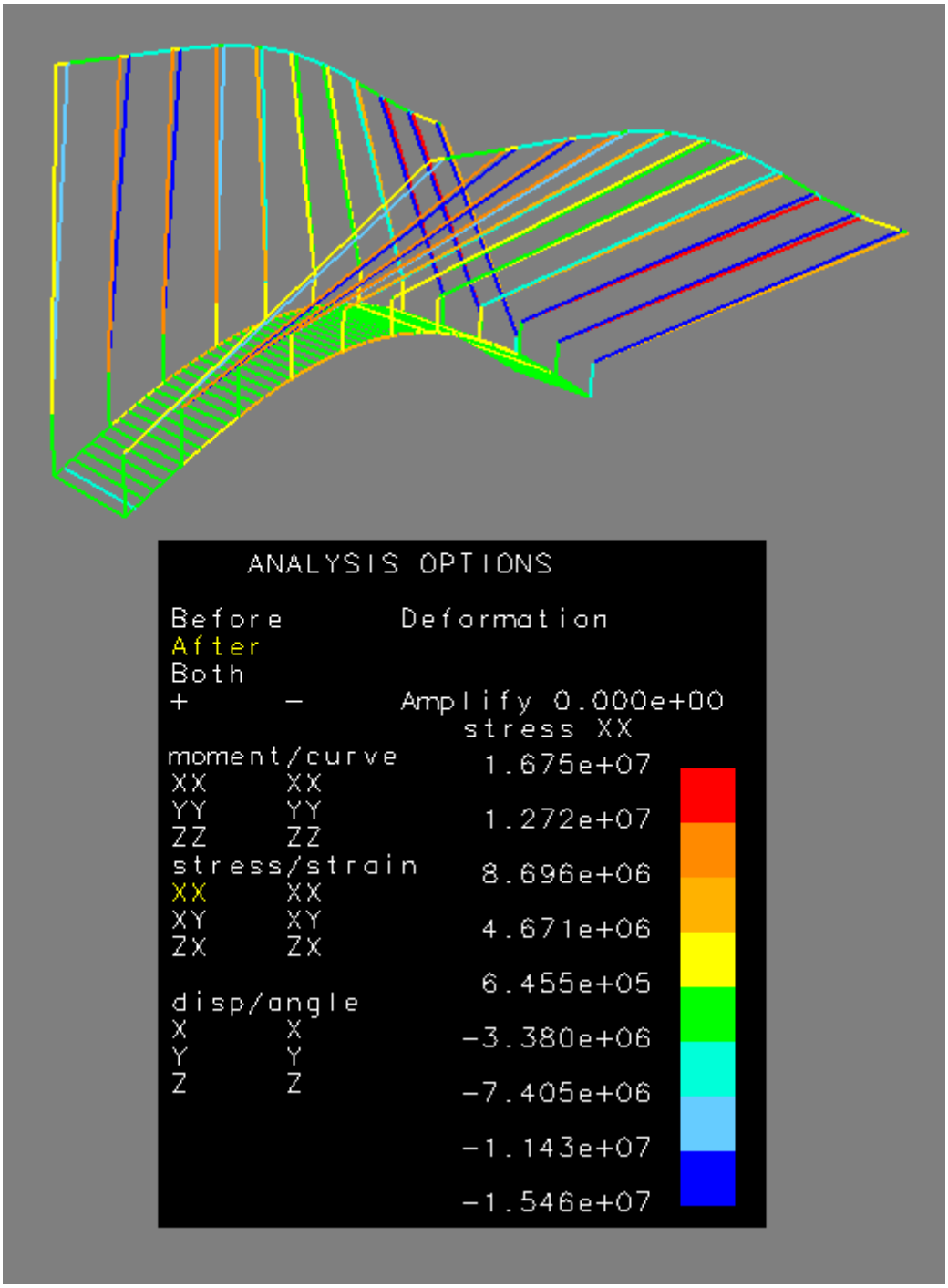


Figure 55: High-Stress bridge with maximum tensile stress of 16.75 N/mm^2

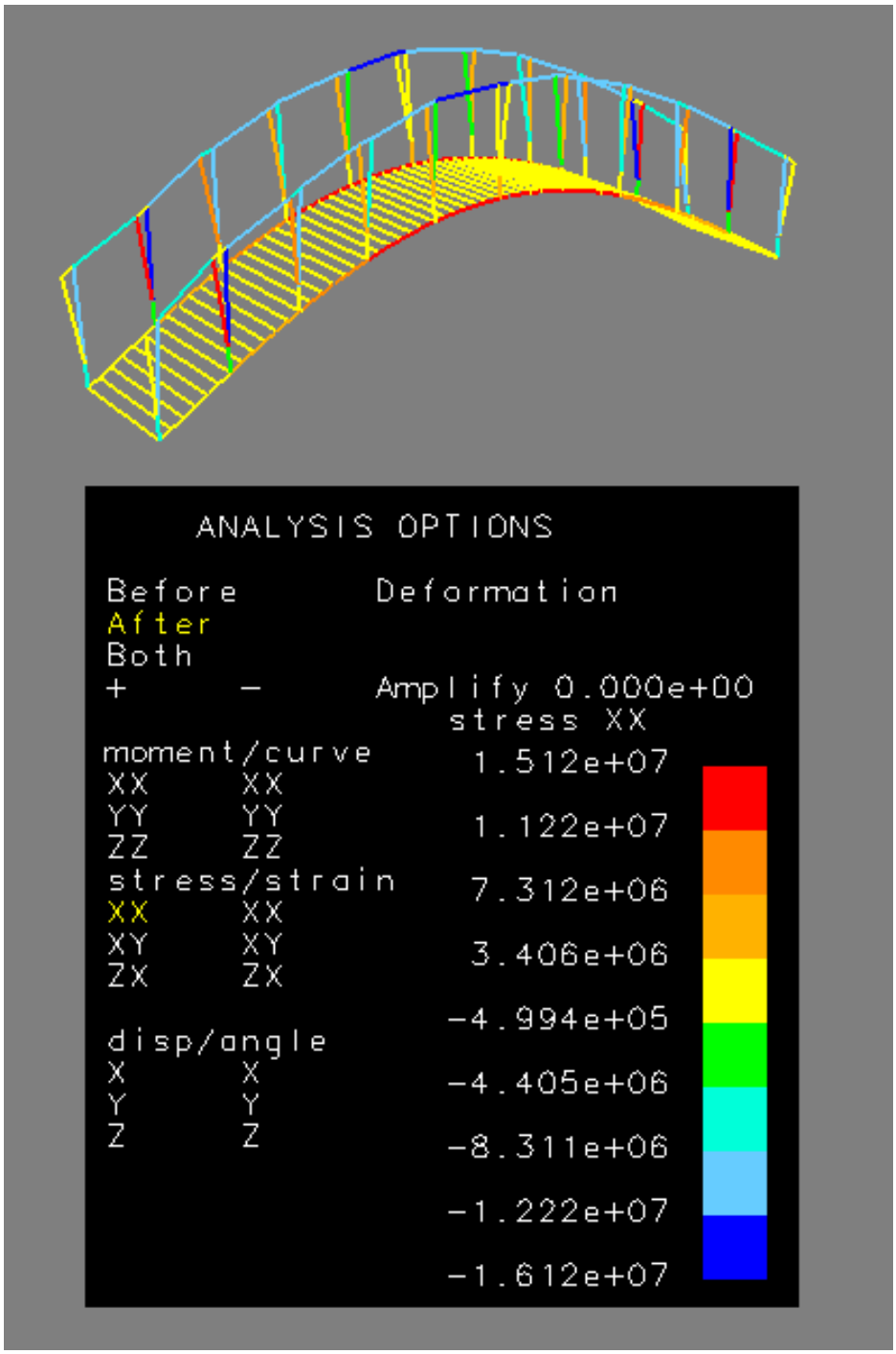


Figure 56: High-Stress bridge with maximum tensile stress of 15.12 N/mm^2

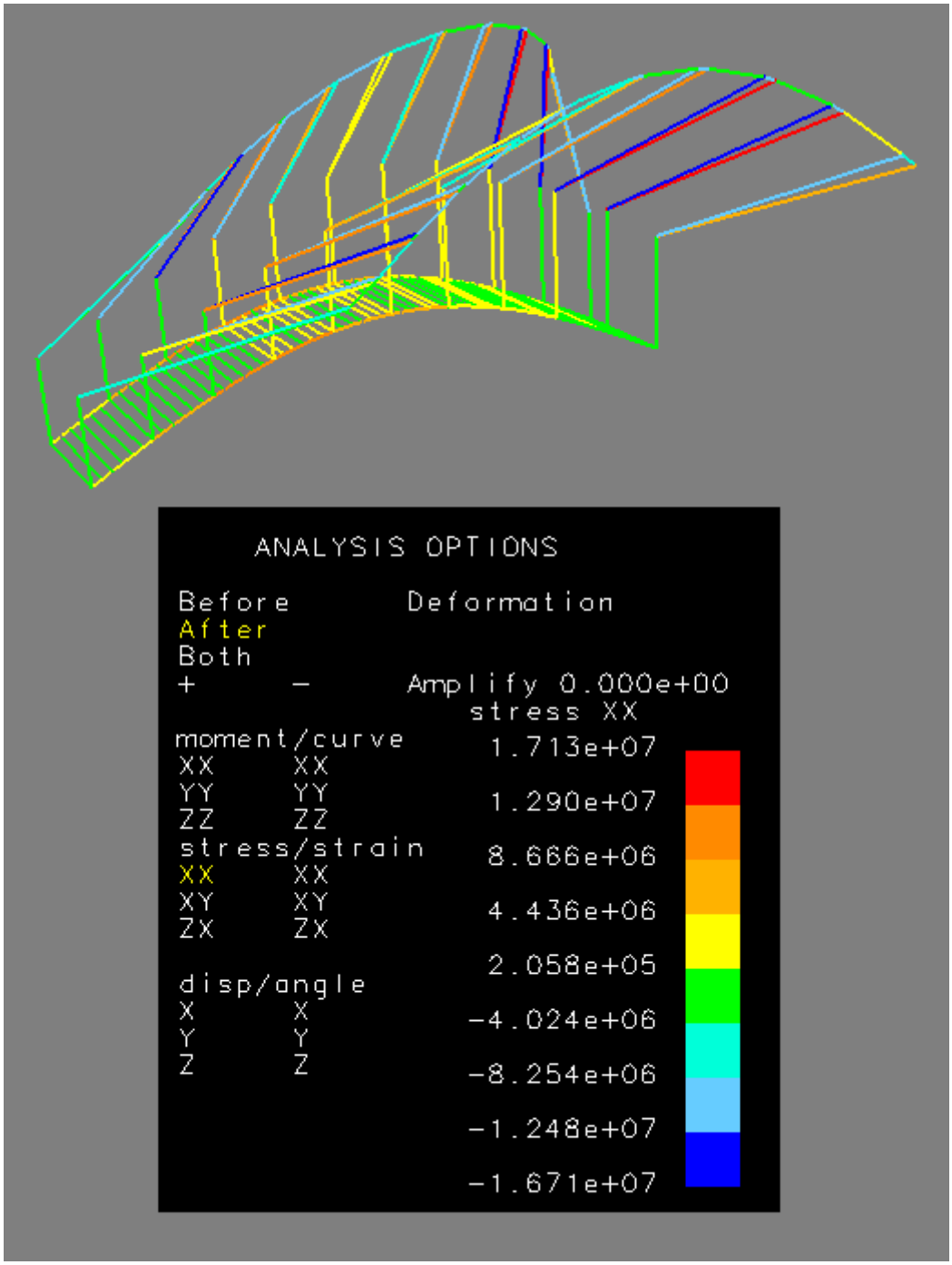


Figure 57: High-Stress bridge with maximum tensile stress of 17.13 N/mm^2

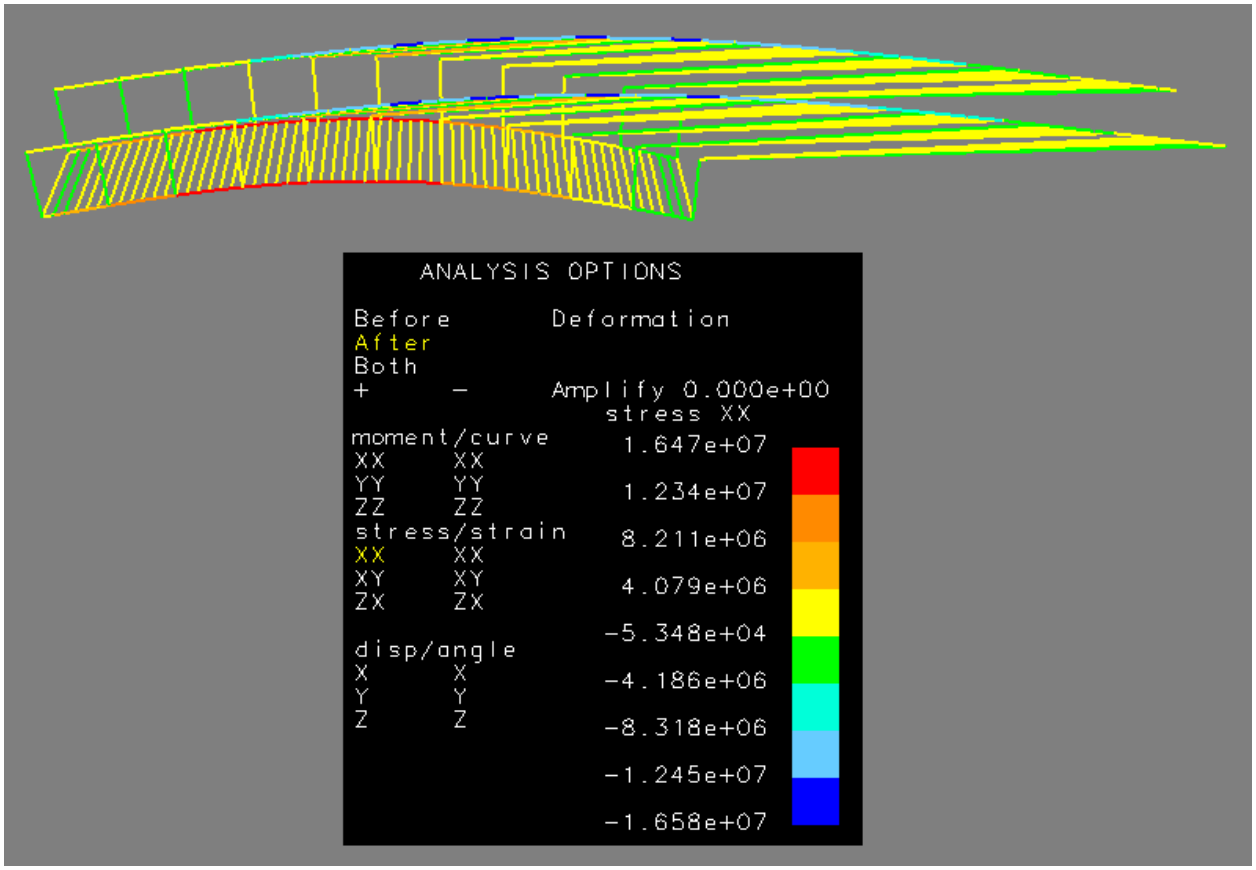


Figure 58: High-Stress bridge with maximum tensile stress of 16.47 N/mm^2

6.1.5 Failed Bridges

Based on permissible grade stresses from BS EN 338-2003, any part of a bridge which exceeds the maximum tensile stress (parallel to the grain) of 18 N/mm^2 for class D30 hardwoods fails the entire bridge. The following is a selection of bridges for which this tensile capacity (and in the case of Figure 59 and Figure 60, the compressive capacity parallel to the grain – 23 N/mm^2) has been exceeded, thus failing the bridge.

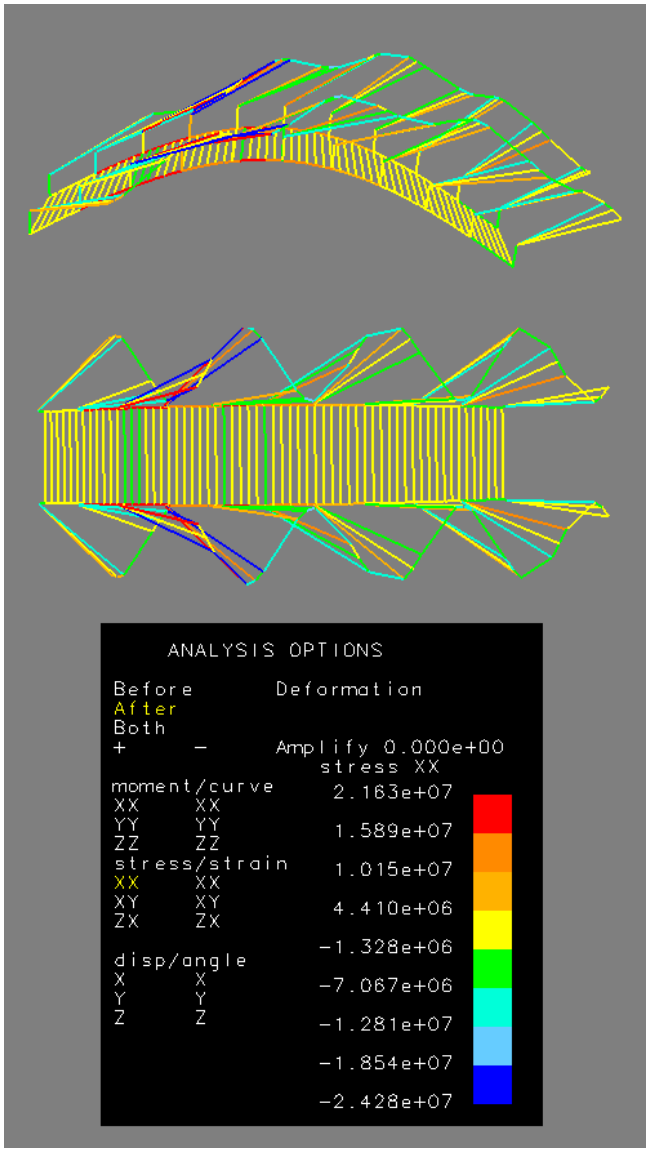


Figure 59: Both the maximum tensile stress of 18 N/mm^2 and the maximum compressive stress of 23 N/mm^2 have been exceeded

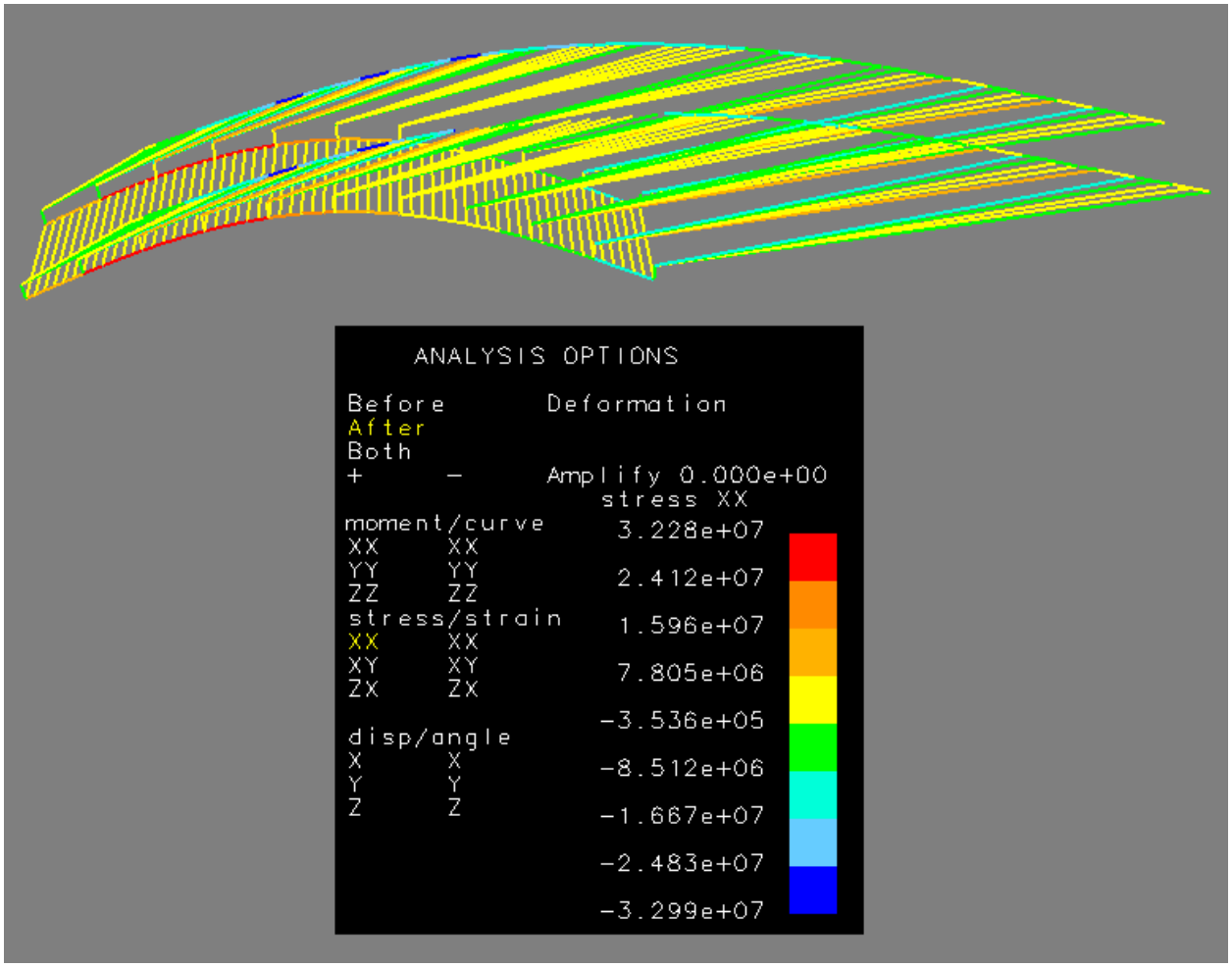


Figure 60: Both the maximum tensile stress of 18 N/mm^2 and the maximum compressive stress of 23 N/mm^2 have been exceeded

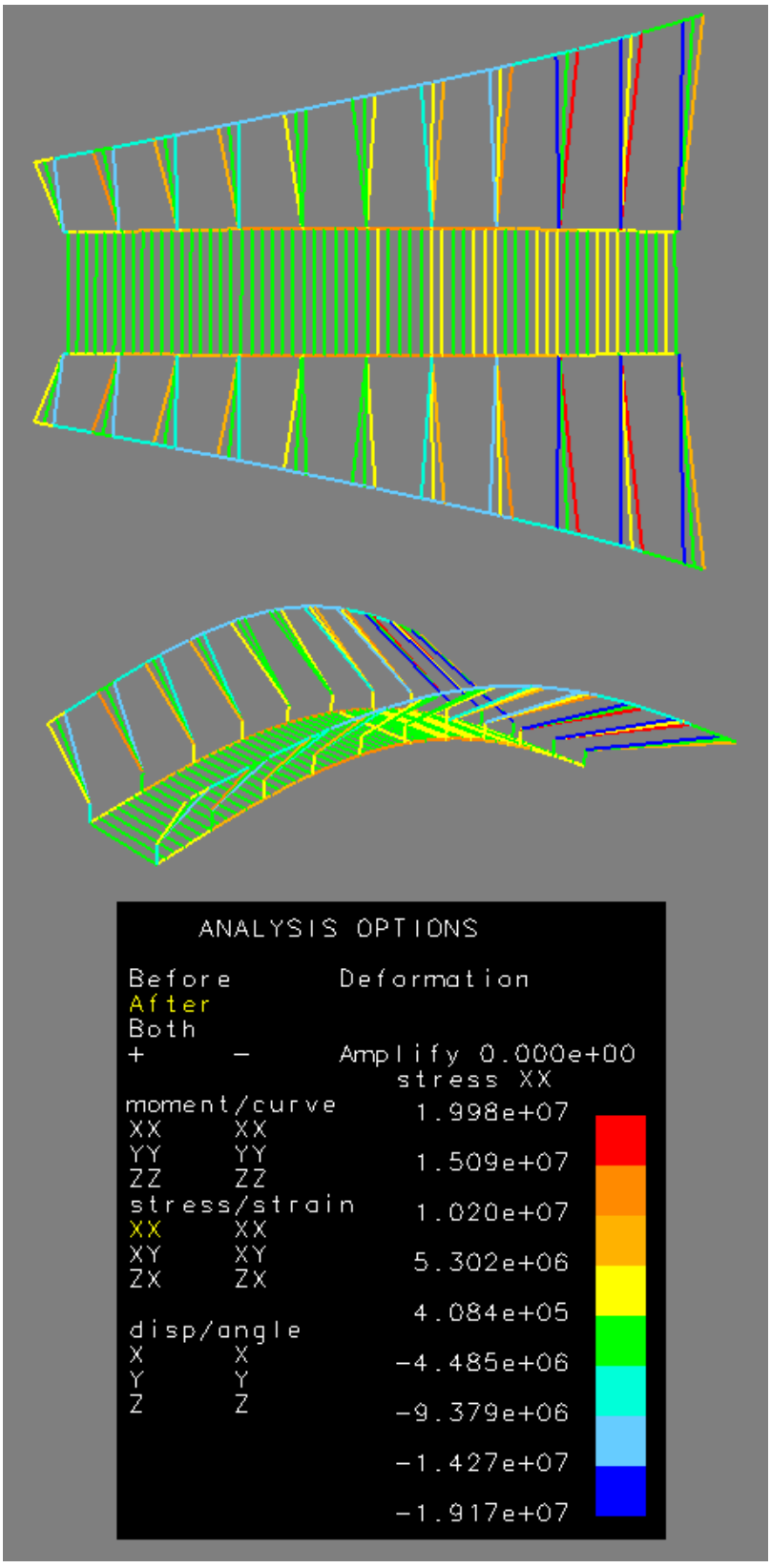


Figure 61: The maximum permissable tensile stress has been exceeded in numerous members, failing this bridge

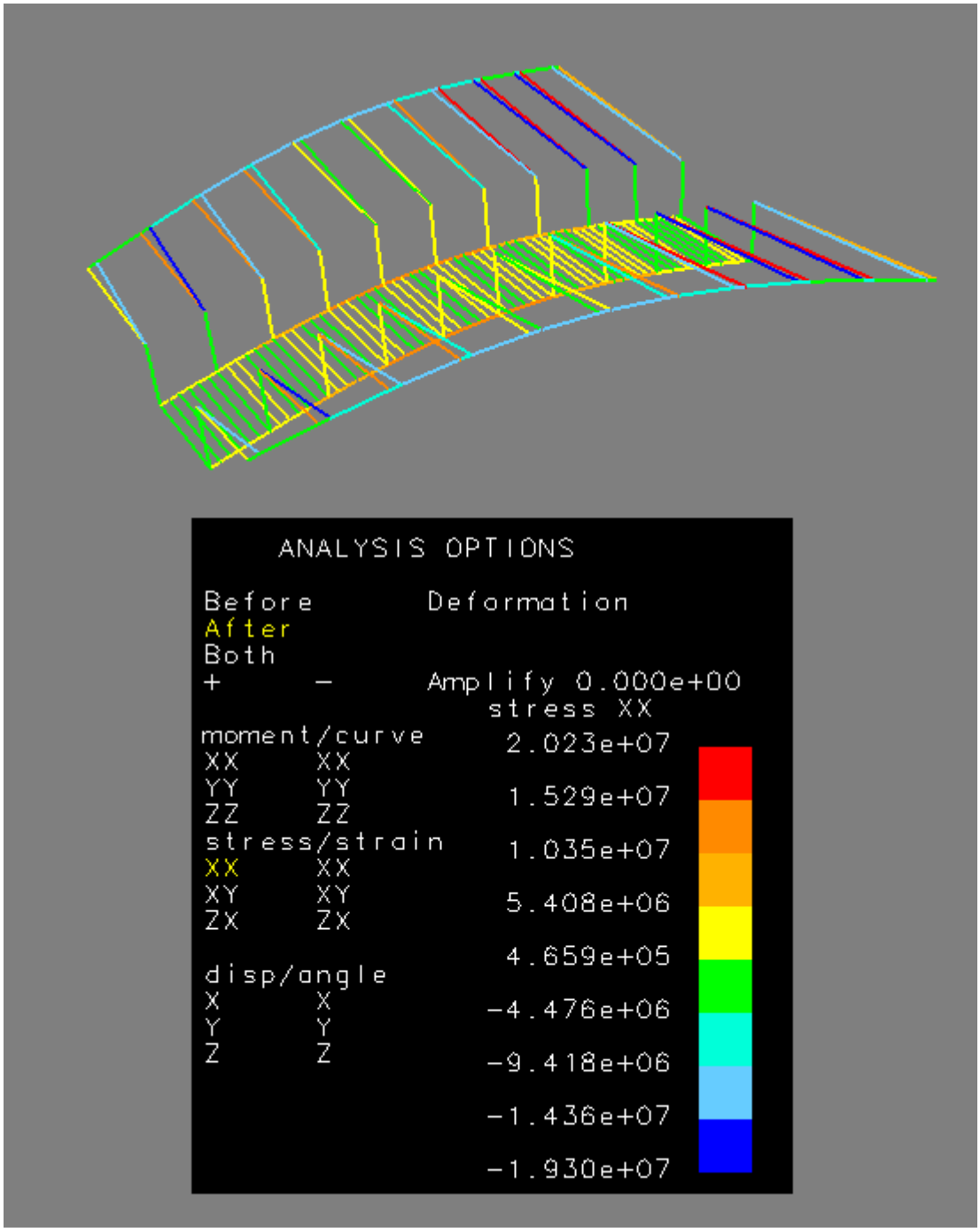


Figure 62: A maximum tensile stress of 20.23 N/mm^2 fails this bridge

SLFFEA has the capacity to illustrate bridge failure by amplifying the deflection of the bridges. This is illustrated in Figure 63, where the centremost members of the walkway can be seen to deflect excessively with increased load, until eventual failure.

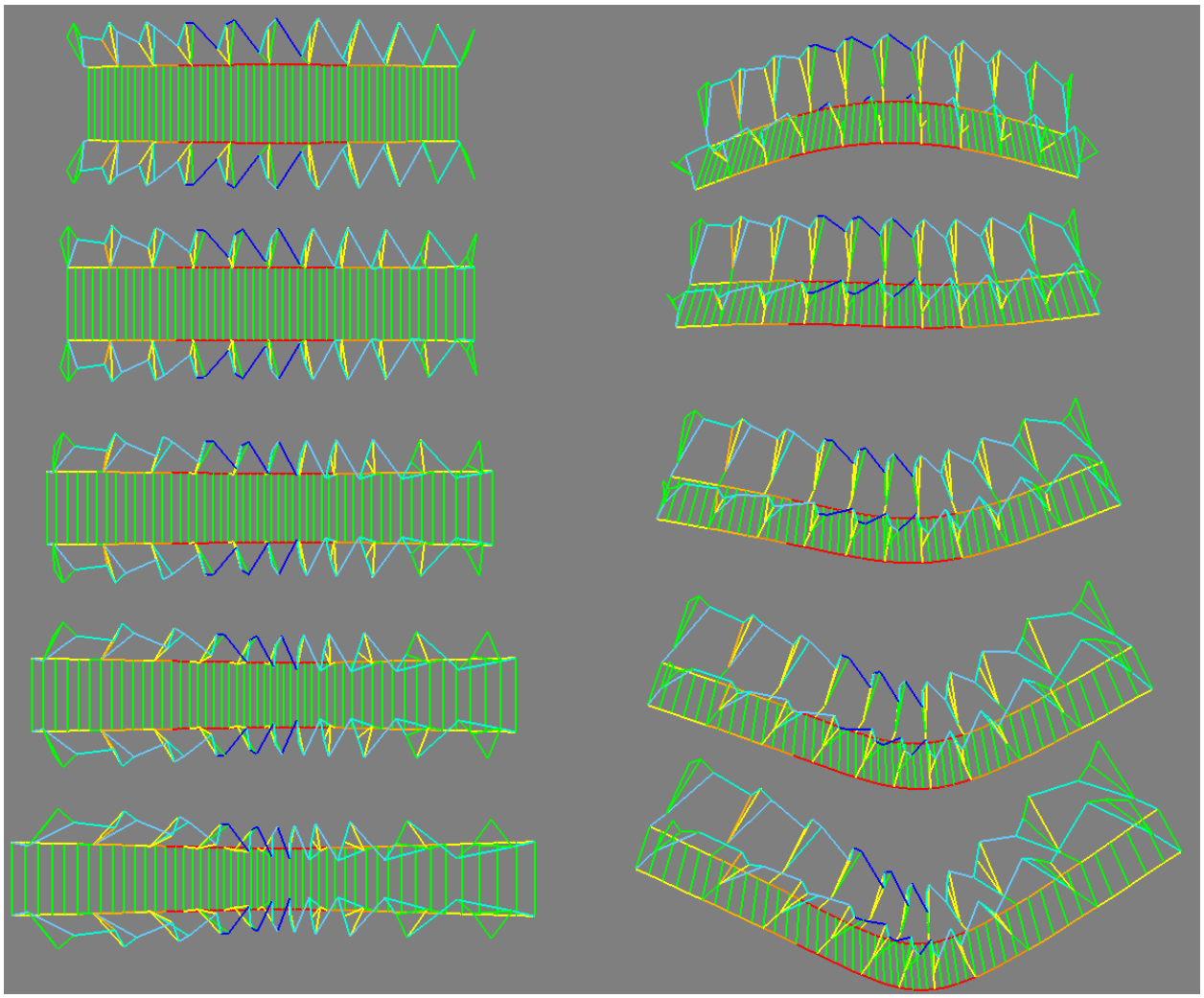


Figure 63: Progressive failure of Wave style bridge, showing gradual compression of upper handrail members in "accordion" style

6.2.Pinned-Pinned Bridges

The pinned-pinned bridges are of less analytical interest, as the stress state of the bridge has a direct correlation with the height of the arch of the bridge; the higher the arch, the more efficient the bridge. The handrail design has little or no impact on the stress state of the bridges, as the deck section transmits all the load via compression to the supports - it can be seen from the analysis results that the only part of the bridges that carries any notable stress is the arches of the deck.

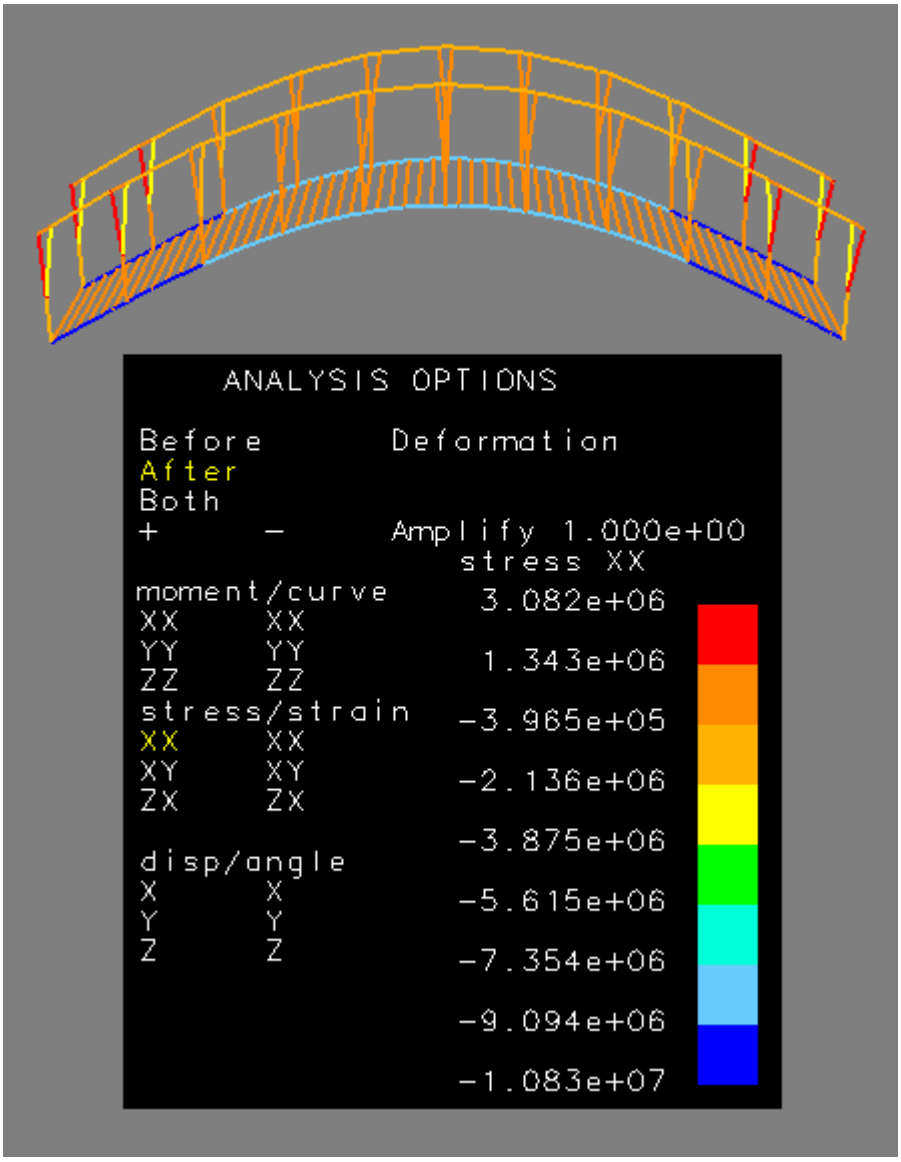


Figure 64: Low-Stress bridge with high arch, maximum compression 10.8 N/mm^2

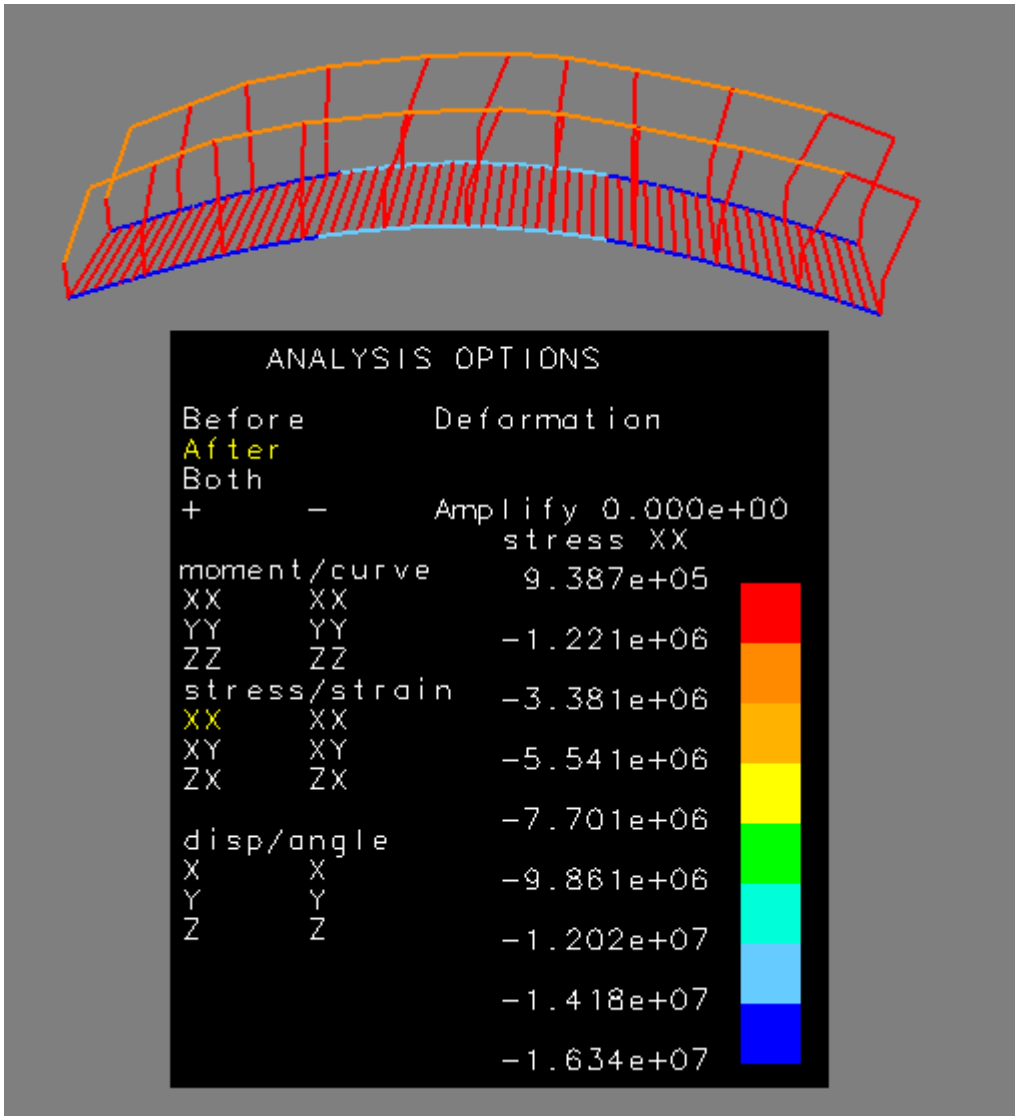


Figure 65: Medium-Stress bridge with medium-rise arch, maximum compressive stress of 16 N/mm^2

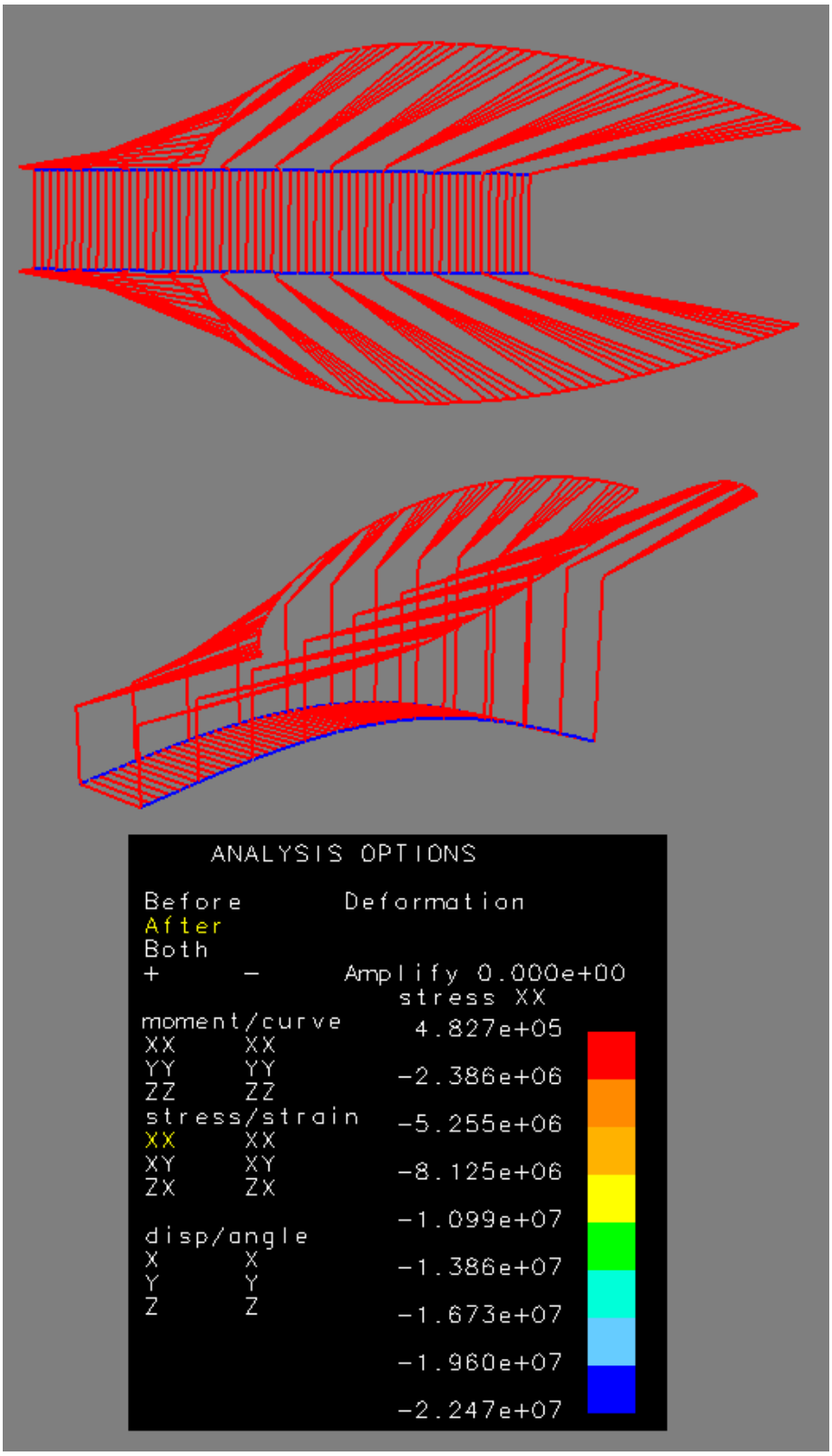


Figure 66: High-Stress bridge with low-rise arch, maximum compressive stress of 22.4 N/mm^2

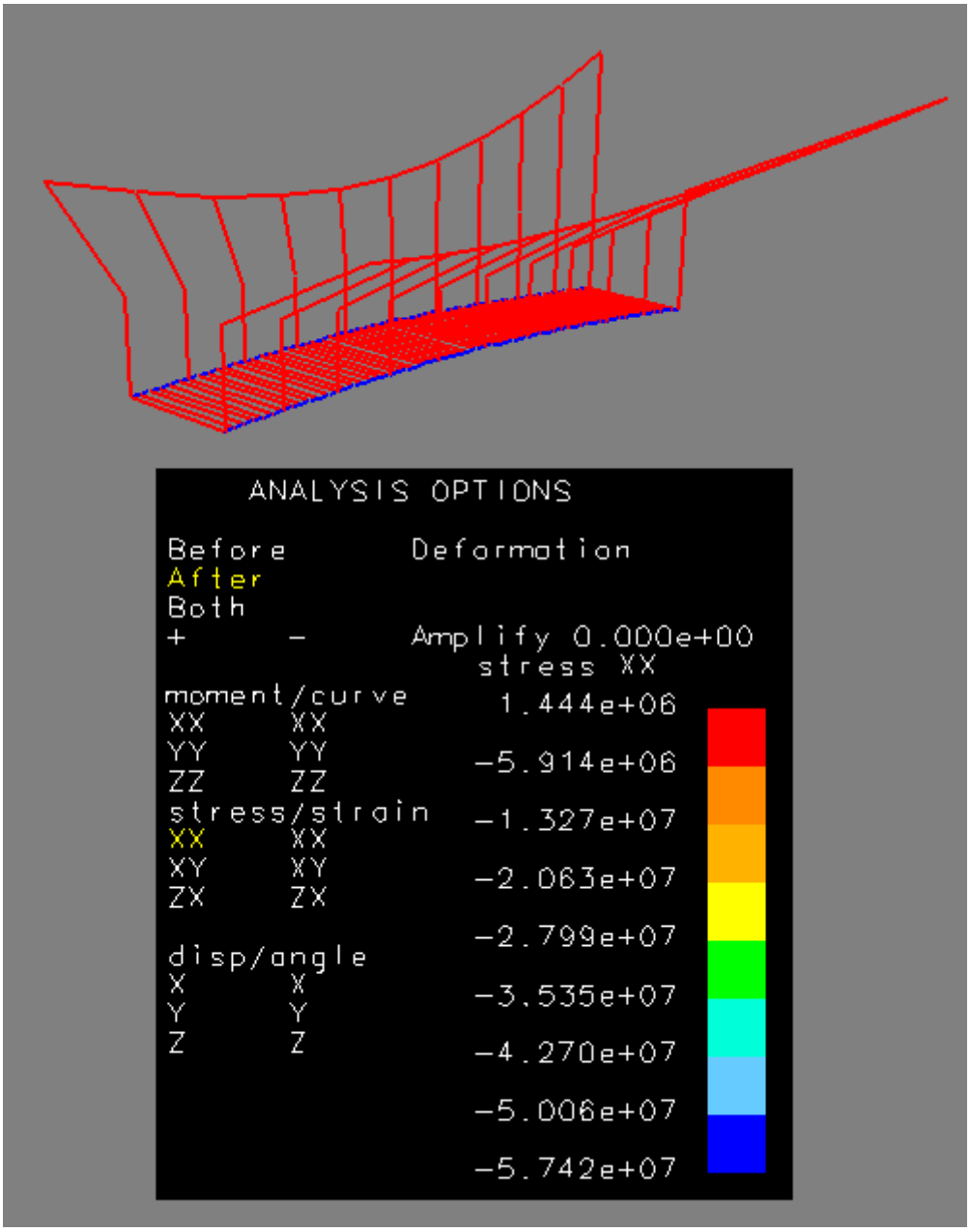


Figure 67: Failed flat bridge with maximum compressive stress of 57 N/mm^2, far exceeding the allowable compressive stress of 23 N/mm^2

7. CONCLUSIONS & RECOMMENDATIONS

7.1. Slenderness, Buckling & Moments, and the Use of Structural Analysis as a Fitness Function

The conclusions to be drawn from this study are interesting. With a rolling restraint at one end of the bridge, an arched design is not the most ideal structure for a 10 meter span. However, the situation is reversed with the introduction of full translational (pinned-pinned) restraint, with the arch becoming the most dominant structural feature of the bridge (Figure 64), with a direct correlation between the stress state of the bridge and the height of its arch. With a pinned-roller case, variations in the maximum stresses in the many designs of bridges, some with arches (e.g. Figure 59) and some without (e.g. Figure 44), have proved that the presence of an arch is not the most structurally pertinent feature, but rather it is the height of the vertical stays which dictates the internal stresses.

A flaw in this logic is glaringly obvious, however, in the size of the members themselves. With all of the ultra low-stress and low-stress bridges, the lengths of the members are unrealistically long. If these designs were to be built, the members would buckle long before any load could be successfully applied. Compressive members at the extreme ends of the bridges (see Figure 42 and Figure 47) would need to be substantially thickened to prevent slenderness of the members from causing local buckling.

Another issue which GEVA has no appreciation of is moments within the structure. It is clear, from Figure 41 Figure 43 and Figure 44, that the lower stress structures would generate huge moments within the sections due to the presence of enormous cantilevers.

At present, GEVA operates on the interactive evolutionary computation method described in Chapter 2. As explained, this method involves the user specifying whether or not they like the current design by simply pressing the up button to assign a “good” fitness value, and the down button to assign a “bad” fitness value (the default setting is for bad fitness for all individuals). The current fitness values are 1000 for a “fit” design, or 1000000000 for an “unfit” design – the greater the number, the worse the fitness of the individual.

One identified solution to the slenderness and moments problem is to include structural parameters in the fitness function of the GEVA program itself, thereby ensuring that infeasible or unfit solutions are penalised. If the stress levels of the structure were set as the fitness function,

structures with greater stress present (e.g. the "Failed Bridges" section in the previous chapter, incorporating Figure 59 and Figure 62) would be assigned a "low" fitness level, while the structures with a relatively low stress level (e.g. the "Ultra Low-Stress Bridges" section in the previous chapter) would be assigned a greater fitness level, ensuring they would be selected for re-population for the following generation. However, if moments were to be taken into account the lower stress bridges wouldn't fare so well due to relatively high moments, as a result of their cantilevered sections. A balance between both stress and moments would therefore need to be reached in the fitness function, which would evolve more structurally sound individuals.

7.2. Limitations of the code & Recommendations

One of the more difficult aspects of creating the Python code was where to set the limits of what the code could do. With such a powerful language, the possibilities are unlimited, with the only boundary being the time in which to complete the project. To this extent, it was decided to keep the program simple in its objective, but to have it as neat as possible. A number of areas for further work with this program would be:

a) Materials

There are currently only two selectable materials in the new GEVA-Blender interface – Timber and Steel. With Timber, the default section size is 100mm x 200mm, however the user can specify the section size of the timber they wish to use in the structure in the code itself. With steel the default section size is a 254x146x43 UB, although as with Timber the user may specify which I-section they wish to use (the program recognises the imputed steel section value by reading it off the British Standards steel tables from BS 4 Part 1: 2005). These options could be taken further with the addition of more steel sections, including universal columns (U.C's), angles, T-sections and circular and rectangular hollow sections (C.H.S and R.H.S). Additional materials such as Aluminium, or combinations of multiple materials would be useful for the next phase. It would be possible to analyse the stresses in the structure and then to write a program which would replace certain members in the overall structure with more appropriate ones, such as steel cables for high tension members or concrete columns for high compression struts.

b) Load Cases and further analysis

The current version of the analysis code only allows for one specific load case as defined by BS 5400-2: 2006 – pedestrian footbridges under 30m should have live load of 5kN/m^2 imposed on them. While this is taken into account, there is no accommodation for wind loading or lateral loads, an issue which needs to be addressed. SLFFEA allows for multiple load scenarios and even 2^{nd} and 3^{rd} stage non-linear analysis, which could be added at a later date.

c) Vibrations

SLFFEA is not designed with bridge analytics in mind; rather this project has adapted it to model a bridge as a beam. As such, there is no vibration analysis present in the data. Future work on this program would need to write a specific code for vibration analysis of the bridges if the scale of the designs were to increase.

d) Temperature

Thermal gradients and thermal expansion of the bridges are not taken account of in this analysis. However, as with vibrations above future work on this program would need temperature effects to be taken into consideration if the scale of the designs were to increase.

e) Foundations

This is perhaps one of the bigger missing sections of the GEVA-Blender program, in that it still has no appreciation of how its structures interact with the ground. Foundations are obviously one of the most important parts of any structure, and future improvements of this program would require serious attention in this area.

f) Connections between members

Another serious omission from the GEVA-Blender arsenal is the notion of connections between members. At present, the program composes its structures by joining up the nodes of individual bars and intersecting the beams with each other. This is acceptable for proof-of-concept in the creation of new grammars for GEVA, but proper connections have been identified as an area for future improvement in the GEVA program as a whole.

8. REFERENCES

British Standards Institution (2003), BS EN 338 – 2003: Structural Timber Strength Classes, BSI, London

British Standards Institution (2004), BS EN 1995 – 2: 2004, Eurocode 5: Design of timber structures - Part 2: Bridges, BSI, London

British Standards Institution (2006), BS EN 5400-2 2006: Steel, Concrete and Composite Bridges - Part 2: Specification for loads, BSI, London

Darwin, C., (1859), On the Origin of Species, John Murray, London

DeJong, K.A., (2006), *Evolutionary Computation*, MIT Press, Cambridge

Garshol, L.M. (2008), *BNF and EBNF: What are they and how do they work?* [online], available at http://www.garshol.priv.no/download/text/bnf.html, accessed 19-01-2010

GEVA - Grammatical Evolution in Java, (2008), http://ncra.ucd.ie/geva/

Ghali, A., Neville, A.M., & Brown, T.G. (2009), *Structural Analysis, a Unified Classical & Matrix Approach*, 6th Ed., Spon Press, New York

Kicinger, R., Arciszewski, T., and De Jong, K. A. (2005), *Evolutionary computation and structural design: a survey of the state of the art*, Computers & Structures, Volume 83, Issues 23-24

Koza, J.R., (1992), *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA

Logg, A. (2007), *Automating the Finite Element Method*, Archives of Computational Methods in Engineering, Volume 14, Number 2, pages 93 – 138

Lutz, M., & Ascher, D. (2004), *Learning Python*, 2nd Ed, O' Reilly Media Inc., Califonia

O'Neill, M. & Ryan, C. (2001), *Grammatical Evolution*, IEEE Transactions on Evolutionary Computation, Volume 5, Number 4, pages 349 - 358

O'Neill, M. & Ryan, C., (2003), *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*, Kluwer Academic Publishers, Massachusetts

O'Neill, M., Hemberg, E., Gilligan, C., Bartley, E., McDermott, J., & Brabazon, A. (2008), *GEVA: Grammatical Evolution in Java*, SIGEVOlution Summer 2008, Volume 3, Issue 2, pages 17-22

O'Neill, M., Hemberg, E., Bartley, E., Brabazon, A. & Gilligan, C. (2008) *GEVA- Grammatical Evolution in Java*. http://ncra.ucd.ie/GEVA

O'Neill M., Swafford J.M., McDermott, J., Byrne J., Brabazon A., Shotton E., McNally C., Hemberg M. (2009), *Shape Grammars and Grammatical Evolution for Evolutionary Design*, Genetic and Evolutionary Computation Conference GECCO 2009 ACM Montreal, Canada

O'Neill, M., McDermott, J., Swafford, J.M., Byrne, J., Hemberg, E., Shotton, E., McNally, C., Brabazon, A., Hemberg, M. (2010), *Evolutionary Design using Grammatical Evolution and Shape Grammars: Designing a Shelter*. International Journal of Design Engineering (volume in press)

Poli, R., Langdon, W. B., & McPhee, N. F. (2008), *A field guide to genetic programming*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, (With contributions by J. R. Koza)

Robot Structural Analysis Professional 2010 – Student version, available under a student licence from www.autodesk.com, Autodesk inc.

Rockey, K.C., Evans, H.R., Griffiths, D.W., & Nethercot, D.A. (1983), *The Finite Element Method, A Basic Introduction for Engineers*, Granada, St. Albans, Herts

Ross, C.T.F. (1985), *Finite Element Methods in Structural Mechanics*, Ellis Horwood Limited, Chichester

Ryan, C., Collins, J.J. & O'Neill, M. (1998), *Grammatical Evolution : Evolving Programs for an Arbitrary Language*, In W. Banzhaf, et al., editors, Proceedings of the First European Workshop on Genetic Programming, Volume 1391 of LNCS, pages 83–95,. Springer-Verlag, Paris

http://slffea.sourceforge.net/index.html

Stiny, G. & Gips, J. (1971), *Shape Grammars and the Generative Specification of Painting and Sculpture*, Presented at IFIP Congress 71 in Ljubljana, Yugoslavia, Republished in O R Petrocelli (ed.) *The Best Computer Papers of 1971* (Auerbach, Philadelphia, 1972), Pages 125-135

Structural Use of Hardwoods, [online], available at http://www.trada.co.uk/ , accessed 01-02-10

Sunley, J., & Bedding, B. (1985), *Timber in Construction*, Batsford/TRADA, London

Turner, M.J., Clough, R.W., Martin, H.C., Topp, L.C. (1956), *Stiffness and deflection analysis of complex structures*, Journal of the Aeronautical Sciences, Vol. 23, Pages 805 – 823

Weaver Jr., W. & Johnston, P.R., (1984), *Finite Elements for Structural Analysis*, Prentice-Hall Inc., Englewood Cliffs, New Jersey

Willis, M., Hiden, H., Marenbach, P., McKay, B., & Montague. G.A. (1997), *Genetic programming: An introduction and survey of applications*. In A. Zalzala, editor, Second International Conference on Genetic Algorithms in Engineering Systems: Innovations and

Applications, GALESIA, University of Strathclyde, Glasgow, UK, 1-4 September 1997. Institution of Electrical Engineers.

Yu, T. (2001), *Higherarchical Processing for Evolving Recursive and Modular Programs Using Higher-Order Functions and Lambda Abstraction*, Genetic Programming and Evolvable Machines, Vol.2, Iss.4, pages 345-380

Zhang, W.J., & Van der Werff, K. (1998), *Automatic communication from a neutral object model of mechanism to mechanism analysis programs based on a finite element approach in a software environment for CADCAM of mechanisms*, Finite Elements in Analysis and Design 28, pages 209-239

9. APPENDIX 1: Analysis_2.py

```
# This program will make a list of nodes and beam co-ordinates to be used by
# the structural analysis program SLFFEA. The intention is to create two
# separate lists: one which will consist of a numbered list of individual
# nodes, and the other of which will consist of a list of bar connections
# from numbered (indexed) node to numbered node.

# first we define our list of nodes and beams, along with other global
# variables

global nodeslist

def clear():

# We define a clear function to eusure we are starting from scratch; no
# values are retained from any previous runs of the program, it is cleared
# every time.

   global nodeslist
   nodeslist = []

clear()

def beam(x0, y0, z0, x1, y1, z1):

      global nodeslist

      a = (x0, y0, z0)
      b = (x1, y1, z1)

      nodeslist.append(a)
      nodeslist.append(b)

# This adds the newly created nodes to the existing nodeslist

      print "The list of nodes is: "
      print nodeslist

# This prints off the list of nodes.
```

10. APPENDIX 2: analysis.py

Note: line lengths have been altered for neatness

```
# This program will make a list of nodes and beam co-ordinates to be used by
# the structural analysis program SLFFEA. The intention is to create two
# separate lists: one which will consist of a numbered list of individual
# nodes, and the other of which will consist of a list of bar connections
# from numbered (indexed) node to numbered node. The program will then save
# these lists in a separate text file for use by the program SLFFEA.

from math import sqrt
import os
import subprocess
import re
import gui
import sys

class Analysis:

# define our list of nodes and beams, along with other global variables

global nodeslist
global beamslist
global fixedpoints
global udl
global loadelement
global stresslist
global strainlist
global momentslist
global Emod
global density
global area
global iy
global iz
global material
global maxtension
global maxcompression
global maxmoment
global new_nodes_list
global new_beams_list


def __init__(self):

# We set our maximum stress levels here, as taken from BS EN 338-2003:
# Structural Timber Strength Classes

    global maxtension
    maxtension = (1.8*10**7)

# Maximum permitted stress in tension, N/mm^2
    global maxcompression
    maxcompression = (2.3*10**7)

# Maximum permitted stress in compression, N/mm^2
def clear(self):
```

```
# We define a clear function to eusure we are starting from scratch; no
# values are retained from any previous runs of the program, it is cleared
# every time.

    global fixedpoints
    fixedpoints = []
    global loadelement
    loadelement = []
    global stresslist
    stresslist = []

def runanalysis(self):

# This will run the analysis program for us (behind the scenes), and will
# create a file with an extension (.obm) which will contain all the data
# recorded from the analysis.

print "\nRunning analysis...\n"

# this opens up the analysis program SLFFEA and pipes in the input file name,
# contained in the file "slfinput.txt". It then saves the display of the
# program into a separate file called "slfoutput.txt.

os.Popen('/home/michael/Desktop/SLFFEA/slffea-1.5/beam/beam/bm >
      /home/michael/Desktop/SLFFEA/slffea-1.5/beam/beam/slfinput.txt <
      /home/michael/Desktop/SLFFEA/slffea-1.5/beam/beam/slfoutput.txt',
      shell=True, stdout=subprocess.PIPE, stdin=subprocess.PIPE)

x = file("/home/michael/Desktop/SLFFEA/slffea-
1.5/beam/beam/slfoutput.txt","r")
print x

# Prints the display of the SLFFEA program on the screen

def showanalysis(self):

# This will fire up the SLFFEA beam analysis GUI so we can see a
# visualisation of the structure with its maximum and minimum stresses
# clearly displayed

print "\nShowing Analysis data\n"

# this opens up the SLFFEA beam analysis Graphic User Interface (GUI)
# which allows the user to see their analysed structure with all the
# maximum and minumum stresses present.

subprocess.Popen('/home/michael/Desktop/SLFFEA/slffea-
      1.5/beam/beam/bmpost > /home/michael/Desktop/SLFFEA/slffea-
      1.5/beam/beam/trussinput.txt < /home/michael/Desktop/SLFFEA/
      slffea-1.5/beam/beam/trussoutput.txt', shell=True,
      stdout=subprocess.PIPE, stdin=subprocess.PIPE)

y = file("/home/michael/Desktop/SLFFEA/slffea-1.5/beam/beam/
trussoutput.txt").read()

print y
```

```
# Prints the display of the SLFFEA program on the screen

def materialselect(self):

# This is where materials are defined: Steel or Timber

    global area
    global iy
    global iz
    global material
    global density
    global Emod

if material == "Timber":

# if the selected material is Timber
# default section size is 100mm x 200mm, easily changeable

    width = 100
    height = 200
    Emod = 10000000000        # Young's Modulus in N/m^2
    density = 5300            # Density in N/m^3
    area = (float(width))*(float(height))*10.0**(-6)
    iz = (((float(width))*(float(height)**3))/12)*10**(-12)
    iy = (((float(height))*(float(width)**3))/12)*10**(-12)

elif material == "Steel":

# otherwise, if the selected material is Steel
# default steel size is 254x146x43 UB, easily changeable

   size = "254x146x43"
   sections = file("/home/michael/geva_blender/blender/SteelTables.txt", "r")

 # Opens up the steel UB tables

   line = sections.readline()
   while line:
      if line.startswith(size):

# searches for the specified section size in the steel tables

         everything = line.split()
         density = 7850       # Density in kg/m^3
         Emod = 210000000     # Young's Modulus in N/m^2
         area = float(everything[8]) * 0.0001
         iy = float(everything[10]) * 10**(-8)
         iz = float(everything[9]) * 10**(-8)
         break
      line = sections.readline()
```

```
def searchanalysis(self):

# This will search the analysis results for the stresses data

   global stresslist
   global maxtension
   global maxcompression

x = file('/home/michael/geva_blender/GEVA_v1/bin/xxx.obm', 'r')

# opens the results file

line = x.readline()
while line:

   if line.startswith("element no. and nodal pt. no. with local stress
   xx,xy,zx and moment xx,yy,zz") or line.startswith("element no. and gauss
   pt. no. with local stress xx,xy,zx and moment xx,yy,zz"):

# searches for the stresses

      line = x.readline()
      while line != (' -10'):
         if line.startswith(' '):
            stresses = line.split()
            a = stresses[0:2]
            b = stresses[2:5]
            d = a + b
            stresslist.append(d)

# Adds the stresses to the stress list

         else:
            break
         line = x.readline()

# Moves on to read the next line

      line = x.readline()

# This next module removes the last item of the stresslist, which is "-10"
# (SLFFEA's end-of-input marker). This only happens if there is something in
# the stresslist in the first place.

if stresslist:
   stresslist.pop()     # Removes the last item
   for i, n in enumerate(stresslist):
      for x in range(2, 5):
         if eval(str(stresslist[i][x])) > 0:
            if abs(eval(str(stresslist[i][x]))) > maxtension:

# checks the xx stresses against the max tension

            print "Element ", stresslist[i][0], "fails in tension."
```

```
      if eval(str(stresslist[i][x])) < 0:
         if abs(float(stresslist[i][x])) > maxcompression:

# checks the xx stresses against the max compression

            print "Element ", stresslist[i][0], "fails in compression."

      else:
         print "Error: No analysis was performed. No stresses present!"

# Uh-oh, it didn't work!

def readfromfile(self, gen, num):

# After the beam function (from Analysis_2.py, see Appendix 2) is called in
# render.py, the list of beams and nodes is written to a new file, one file
# for each individual. The problem with these files is that they contain
# every node that has been generated so far in the GEVA session, which after
# a generation or two can run up to the tens of thousands. What needs to be
# done is to search these lists backwards to find the most recent additions.
# What this function does is search these lists one after another, and delete
# anything that has previously occurred in any previous instances. This
# ensures that each individual only contains the nodes relative to that
# individual, rather than itself and all previous individuals.

global new_nodes_list
nlist = []
new_nodes_list = []
node_counter = ()
new_node_count = ()
onelessnum = int(str(num)) - 1
onelessgen = int(str(gen)) - 1

for line in os.popen("tac /home/michael/geva_blender/blender/Individuals/" +
str(gen) + "/" + str(num) + ".txt"):

# reads the file backwards line by line. "gen" and "num" are taken from the
# current generation and individual that is being displayed on the Blender
# screen, ensuring that each individual is evaluated one after another.

   if line.startswith("[("):
      nlist = str(line)
      node_counter = len(eval(nlist))

# this finds the nodes list (the last line in the file to start with "[(")
# and stores it in Python's memory bank as "nlist".

      os.remove("/home/michael/geva_blender/blender/Individuals/" + str(gen)
+ "/" + str(num) + ".txt")

# once the nodeslist is stored in Python's memory, the previous file is
# deleted and a new, much smaller one is created in its place.

      xxx = file("/home/michael/geva_blender/blender/Individuals/" + str(gen)
+ "/" + str(num) + ".txt", "w")     # The new file is generated
      xxx.write("The node count is:\n" + str(node_counter) + "\n")
      xxx.write("List Of Nodes:\n" + str(nlist))
```

```
# The new file contains both the current list of nodes for that particular
# individual, and a count of the number of nodes generated so far in the
# GEVA session. It is essential to keep track of the total number of nodes,
# as after the first individual has been created, this program needs to
# remove all nodes belonging to previous individuals, which are printed each
# time a new individual is created. This means that in GEVA, each successive
# individual takes longer to generate than the one before it.

   if line.startswith("running beam method"):

# For any individuals in the first generation, but which are not equal to
# the first individual (if the individual number is not 0), the node count
# needs to be adjusted as explained above.

      if int(str(gen)) == 0 and int(str(num)) > 0:
         fin = open("/home/michael/geva_blender/blender/Individuals/" +
str(gen) + "/" + str(onelessnum) + ".txt", "r")

         line = fin.readline()

         while line:
            if line.startswith("The node count is:"):
               old_node_count = fin.readline()

# This is where the node count is updated to find the number of new nodes
# generated in the creation of the new individual.

                  new_node_count = node_counter - eval(str(old_node_count))

            if line.startswith("List Of Nodes:"):

# searches for the nodeslist

               nodes_list = fin.readline()
               new_nodes_wanted = eval(str(nlist)) # How many nodes we need
               nlist = new_nodes_wanted[-new_node_count:]

# The nlist takes the total list of nodes (stored in new_nodes_wanted) and
# only takes the most recently added nodes off the end of the list to form
# the new nodes list.

            line = fin.readline()

   os.remove("/home/michael/geva_blender/blender/Individuals/" + str(gen) +
"/" + str(num) + ".txt")

# The old file is then removed and a new one is written in its place

   xxx = file("/home/michael/geva_blender/blender/Individuals/" + str(gen) +
"/" + str(num) + ".txt", "w")
   xxx.write("The node count is:\n")
   xxx.write(str(node_counter) + "\n")
   xxx.write("List Of Nodes:\n")
   xxx.write(str(nlist))

   break
```

```
# The entire process is repeated again twice: the first time for individuals
# which are the first in a new generation, and the second for individuals
# which come after the first of a new generation. The process has to be
# adapted slightly for the first individual in a generation, as the node
# counter has to search in the previous generation folder and find the last
# created individual for the full nodeslist.

      if int(str(gen)) > 0 and int(str(num)) == 0:
         fin = open("/home/michael/geva_blender/blender/Individuals/" +
str(onelessgen) + "/2.txt", "r")
         line = fin.readline()

         while line:
            if line.startswith("The node count is:"):

# searches for the node counter

               old_node_count = fin.readline()
               new_node_count = node_counter - eval(str(old_node_count))
               print "The current number of nodes is: ", new_node_count

            if line.startswith("List Of Nodes:"):

# searches for the nodeslist

               print "found the previous nodeslist"
               nodes_list = fin.readline()
               new_nodes_wanted = eval(str(nlist))
               nlist = new_nodes_wanted[-new_node_count:]

            line = fin.readline()

   os.remove("/home/michael/geva_blender/blender/Individuals/" + str(gen) +
"/" + str(num) + ".txt")

# removes the old file and then replaces it with a new one

   xxx = file("/home/michael/geva_blender/blender/Individuals/" + str(gen) +
"/" + str(num) + ".txt", "w")
      xxx.write("The node count is:\n")
      xxx.write(str(node_counter) + "\n")
      xxx.write("List Of Nodes:\n")
      xxx.write(str(nlist))

   break

# This final module is for individuals after the first individual, in
# generations after the first generation

      if int(str(gen)) > 0 and int(str(num)) > 0:
         fin = open("/home/michael/geva_blender/blender/Individuals/" +
str(gen) + "/" + str(onelessnum) + ".txt", "r")
         line = fin.readline()

         while line:
            if line.startswith("The node count is:"):
               old_node_count = fin.readline()
```

```
                new_node_count = node_counter - eval(str(old_node_count))
                print "The current number of nodes is: ", new_node_count

            if line.startswith("List Of Nodes:"):

# searches for the nodeslist

                print "found the previous nodeslist"
                nodes_list = fin.readline()
                new_nodes_wanted = eval(str(nlist))
                nlist = new_nodes_wanted[-new_node_count:]

            line = fin.readline()

      os.remove("/home/michael/geva_blender/blender/Individuals/" + str(gen)
+ "/" + str(num) + ".txt")

# removes the old file and re-writes a new one

      xxx = file("/home/michael/geva_blender/blender/Individuals/" + str(gen)
+ "/" + str(num) + ".txt", "w")
      xxx.write("The node count is:\n")
      xxx.write(str(node_counter) + "\n")
      xxx.write("List Of Nodes:\n")
      xxx.write(str(nlist))

      break

   else:
      break

# The function ends
```

```
def writetofile(self, pop, ind):


# This will create the input file for use in the SLFFEA analysis program.
# First the list of global definitions must be imported.


global beamslist
global nodeslist
global fixedpoints
global roller
global loadelement
global materials
global density
global Emod
global area
global iy
global iz

allfixed = fixedpoints + roller     # This is a list of all fixed points

n = len(nodeslist)                  # This is the number of nodes
justf = len(fixedpoints)            # This is the number of fixed points
l = len(loadelement)                # This is the number of loaded elements
m = len(beamslist)                  # This is the total number of beams

# Next the file itself is created. The file is always titled "xxx" in this
# program, and thus gets over-written each time a new individual is analysed.
# If the user wishes to save an individual, they can simply open up the
# directory in which the file is saved (this directory is given at the end of
# the function) and change the file name.

testbeams = file('/home/michael/Desktop/SLFFEA/slffea-1.5/beam/beam/xxx',
'w')

testbeams.write('   numel numnp nmat nmode  (This is for a beam bridge)\n')
testbeams.write('     ' + str(m) + '    ' + str(n) + '  1   0\n')

# Material properties are written

testbeams.write('matl no., E mod, Poiss. Ratio, density, Area, Iy, Iz\n')
testbeams.write('0    ' + str(Emod) + '0.0000    ' + str(density) + '    ' +
str(area) + '    ' + str(iy) + '    ' + str(iz) + '\n')

# The list of beams gets written

testbeams.write('el no.,connectivity, matl no, element type\n')
for i, beam in enumerate(beamslist):
   testbeams.write(str(i) + '    ' + str(beam[0]) + '    ' + str(beam[1]) + '
' + '0' + '    ' + '1' + '\n')
testbeams.write('\n')

# The list of nodes gets written

testbeams.write('node no., coordinates\n')
for ii, node in enumerate(nodeslist):
```

```
   testbeams.write(str(ii)+ '    ' + str(node[0]) + '    ' + str(node[1]) + '
' + str(node[2]) + '\n')
testbeams.write('\n')

testbeams.write('element with specified local z axis: x, y, z component\n')
testbeams.write('-10')

# This is where the fixing points are defined. To allow for a roller at one
# end of the bridge, the list of fixed points is broken up into two sections.
# One gets set as having all fixities (x, y, z + rotations), whereas the
# other does without the x fixity.

testbeams.write('\nprescribed displacement x: node  disp value\n')
for w in range(justf):
   testbeams.write('    ' + str(fixedpoints[w]) + '    0.0\n')
testbeams.write('-10')
testbeams.write('\nprescribed displacement y: node  disp value\n')
for w in range(f):
   testbeams.write('    ' + str(allfixed[w]) + '    0.0\n')
testbeams.write('-10')
testbeams.write('\nprescribed displacement z: node  disp value\n')
for w in range(f):
   testbeams.write('    ' + str(allfixed[w]) + '    0.0\n')
testbeams.write('-10')

# For fully fixed structures, rotational fixities can be prescribed in the
# x, y and z directions at any node. In the bridge design instance, the
# restraints are taken as pinned at one end and a roller at the other (to
# allow for expansion of the bridge)

testbeams.write('\nprescribed angle phi x: node angle value\n')
testbeams.write('-10')
testbeams.write('\nprescribed angle phi y: node angle value\n')
testbeams.write('-10')
testbeams.write('\nprescribed angle phi z: node angle value\n')
testbeams.write('-10')
testbeams.write('\nnode with point load x, y, z and 3 moments phi x, phi y,
phi z\n')
testbeams.write('-10')

# This is where the loaded elements are assigned their loading, in this
# case 5kN/m

testbeams.write('\nelement with distributed load in local beam y and z
coordinates\n')
for w in range(l):
   testbeams.write('    ' + str(loadelement[w]) + '    0    -5000\n')
testbeams.write('-10')
testbeams.write('\nelement no. and gauss pt. no. with local stress vector xx
and moment xx,yy,zz\n')
testbeams.write('-10')
testbeams.close()

# Finally, the location of the new input file is revealed to the user.

print "\nA file has been created for you in the directory:
/home/michael/geva_blender/GEVA_v1/bin \n"
```

```
def fixed(self, pop, ind):

# This module finds the perfectly horizontal beams (in the y-direction only)
# and sets them as the elements over which loading is to be applied. The
# module then searches for the two elements with extreme x-values (i.e. the
# maximum and minimum x-values present) and sets the nodes at either end of
# those bars as the fixing points of the structure. It also generates the
# final nodeslist and beamslist for inclusion in the "writetofile" function.

global loadelement
loadelement = []
global beamslist
beamslist = []
global nodeslist
nodeslist = []
global fixedpoints
fixedpoints = [1, 2]     # The fixing points list is assigned two places
global roller
roller = [3, 4]          # The roller list is assigned two places

x = file("/home/michael/geva_blender/blender/Individuals/" + str(pop) + "/" +
str(ind) + ".txt", 'r')

# opens the results file

line = x.readline()
while line:
   if line.startswith("List Of Nodes:"):

# searches for the Nodes


      nlist = x.readline()
      nodeslist = eval(str(nlist))

# stores the list of nodes in the nodeslist

   line = x.readline()

beam_list = []
n_list = eval(str(nodeslist))

# this next module stores the list of beams as pairs of nodes

while n_list:
   n1 = n_list.pop(0)
   n2 = n_list.pop(0)
   beam = (n1, n2)
   beam_list.append(beam)

# Here a problem needs to be solved. GEVA creates its beams one at a time,
# resulting in a large number of nodes with quite a lot of repetition. While
# the input files for SLFFEA do not require repeated nodes to be omitted,
# they can clutter up the analysis data and obscure results. But simply
#removing the repeated nodes (by not adding them to the list in the first
# place), a problem is encountered in the creation of the beamslist. The
# beamslist relies of the original format of the nodeslist (i.e. every pair
```

```
# of nodes correspondes to a beam in GEVA), and so if the nodeslist is
# tampered with, the beam definitions are lost. What needs to be done is for
# the nodeslist to be created unhindered, and then a memory bank of both
# beams and repeated nodes to be created. Once the beams are stored (in
# beam_list, above), the nodeslist can be fixed so that repeated nodes are
# deleted (below).

temp_n_list = []
   new_n_list = eval(str(nodeslist))
      for i, n in enumerate(nodeslist):
         while new_n_list:
            n1 = new_n_list.pop(0)
            if n1 not in temp_n_list:
               temp_n_list.append(n1)

# Finally, the beamslist is updated to become the indexes of the nodes. Nodes
# which had formerly been repeated and subsequently deleted still exist in
# their original incarnations (i.e. the first time they appeared in the
# list), and so any repeated nodes in the beamslist are referenced back to
# their first appearance.

for i, n in enumerate(beam_list):
   n1 = beam_list[i][0]
   n2 = beam_list[i][1]
   n1_index = temp_n_list.index(n1)
   n2_index = temp_n_list.index(n2)
   pair = (n1_index, n2_index)
   beamslist.append(pair)

# The beamslist is then searched for any elements that have the same x and z
# co-ordinates at either end. This means that they are part of the horizontal
# walkway, and once found they are added to the list of elements to be loaded

   if n1[0] == n2[0] and n1[2] == n2[2] and n1[1] == 0 and n2[1] == -2.5 or
n1[1] == -2.5 and n2[1] == 0:
      loadelement.append(beamslist.index(pair))

yy = []
y = []
for i in loadelement:
   n = beamslist[i]
   m = beamslist.index(n)
   x = nodeslist[beamslist[i][0]][0]
   if x not in y:
      y.append(x)
      s = [x, m, n]
      yy.append(s)

# The walkway list is then ordered according to the x-values of the beams,
# such that the nodes of the first and last elements on the list are the
# fixing points of the bridge.

this = eval(str(yy))
this.sort(key=lambda stu:stu[0])

one = this[0][2]
two = this[-1][2]
```

```
fixedpoints[0] = one[0]
fixedpoints[1] = one[1]
roller[0] = two[0]
roller[1] = two[1]


def setmaterials(self, mat):

# Sets the material to be that selected in the drop-down menu in Blender

   global material

   material = mat


def hardcode(self, pop, ind):

# This is the function that is called when the "Analyse" button is clicked in
# the blender GUI. The functions below are called in succession.

self.materialselect()

# Runs the "Material Select" program

self.fixed(pop, ind)

# Finds the walkway and the fixing points of the bridge

self.writetofile(pop, ind)

# Writes the output to a file

self.runanalysis()

# Runs the SLFFEA analysis program

self.searchanalysis()

# Reads relevant data from analysis results

self.showanalysis()

# Displays the results of the analysis by running the SLFFEA post-processing
# GUI
```

11. APPENDIX 3: ROBOT TEST RESULTS

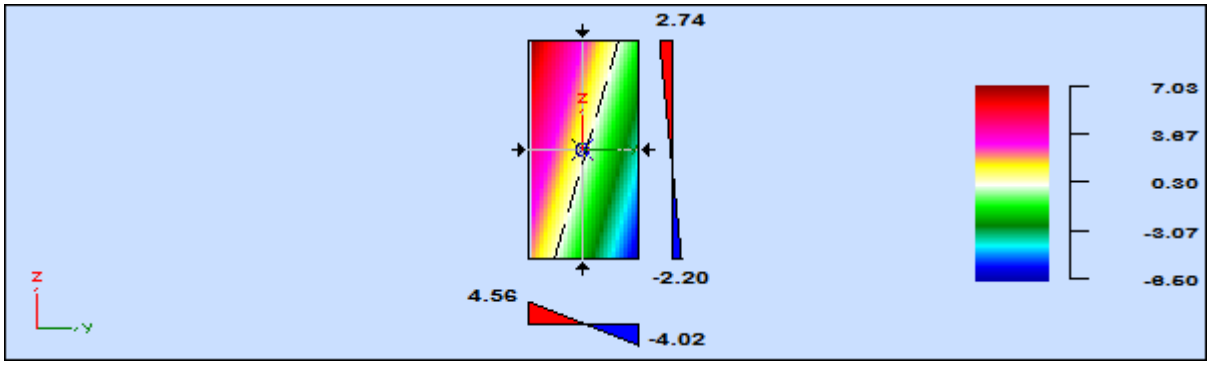
TEST 2: STRESS ANALYSIS IN BAR 14

Section : RECT_1

Element No. : 13

Length : 6708 mm

CROSS SECTION



Load case : "DL1"

Stress analysis type (hypothesis) : Normal

Internal forces taken into account : Fx Fy Fz Mx My Mz

Extreme stresses in the beam

	s_x max	s_x min	| t | max	s_i max
Stresses	28.88 MPa	-27.90 MPa	5.24 MPa	28.88 MPa
Relative position	0.00	0.00	0.09	0.86
Absolute position	0 mm	0 mm	3086 mm	0 mm

RESULTS IN THE SECTION

Section coordinates x/l = 0.50 (Relative) x = 3354 mm (Absolute)

Forces applied to the section

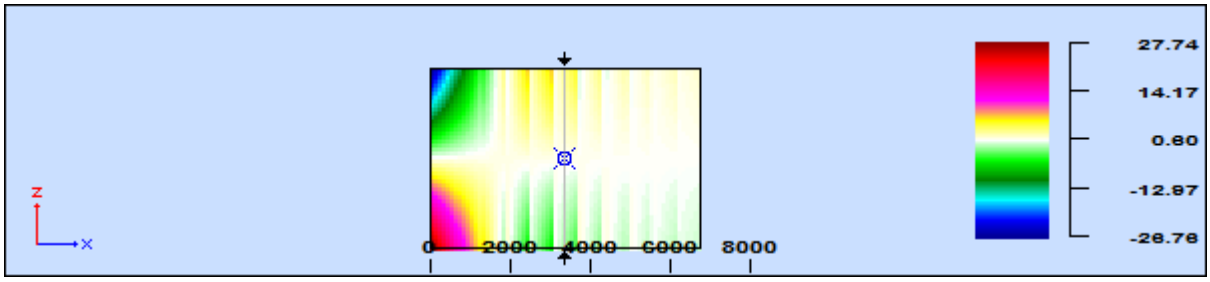
Fx	=	5.39 kN	Mx	=	-2.38 kN*m
Fy	=	-0.06 kN	My	=	1.65 kN*m
Fz	=	5.43 kN	Mz	=	1.43 kN*m

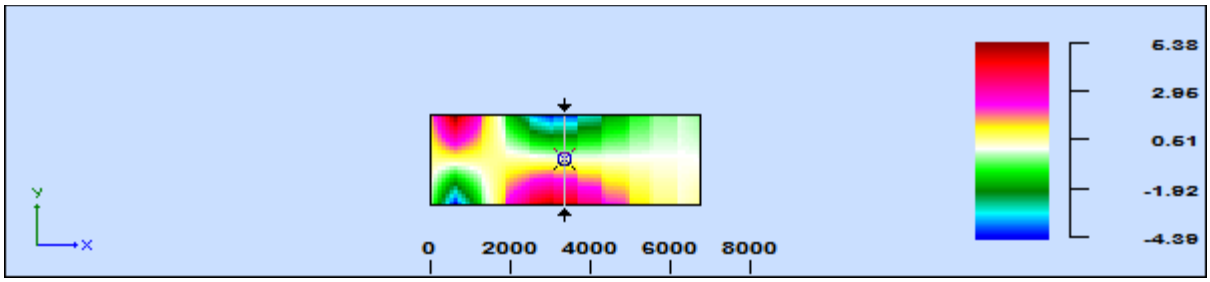
Extreme stresses in the section

	s_x max	s_x min	$\|t_{XY}\|$max	$\|t_{XZ}\|$max
Stresses	7.03 MPa	-6.50 MPa	3.83 MPa	5.23 MPa
Y local	-50 mm	50 mm	1 mm	-50 mm
Z local	100 mm	-100 mm	100 mm	2 mm

	$\|t\|$max	s_i max
Stresses	5.23 MPa	10.25 MPa
Y local	-50 mm	-50 mm
Z local	2 mm	16 mm

LONGITUDINAL SECTION





RESULTS IN THE SECTION

PLANE XZ	S_X max	S_X min	$\|t_{XZ}\|$max	S_i max
Stresses	27.74 MPa	-26.76 MPa	1.09 MPa	27.80 MPa
Relative position	0.00	0.00	0.00	0.00
Absolute position	0 mm	0 mm	0 mm	0 mm

PLANE XY	S_X max	S_X min	$\|t_{XY}\|$max	S_i max
Stresses	5.38 MPa	-4.39 MPa	5.24 MPa	10.16 MPa
Relative position	0.09	0.09	0.46	0.54
Absolute position	604 mm	604 mm	3086 mm	3622 mm