Structural Optimisation via Genetic Algorithms

by

Sophia Aletta Appelo

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Civil Engineering at Stellenbosch University

Department of Structural Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Mr. E. van der Klashorst

December 2012

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: ....

Abstract

Structural Optimisation via Genetic Algorithms

S.A. Appelo

Department of Structural Engineering,

University of Stellenbosch,

Private Bag X1, Matieland 7602, South Africa.

Thesis: MScEng (Civil)

December 2012

The design of steel structures needs to incorporate some optimisation procedure that evolves the initial design into a more economic final design, where this final design must still satisfy all the initial design criteria. A candidate optimisation technique suggested by this research is the genetic algorithm. The genetic algorithm (GA) is an optimisation technique that was inspired by evolutionary principles, such as the survival of the fittest (also known as natural selection). The GA operates by generating a population of individuals which 'compete' with one another in order to survive, or differently stated, in order to make it into the next generation. Each individual presents a solution to the problem. Surviving solutions which propagate through to the next generation are typically 'better' or 'fitter' than the ones that had died off, hence suggesting a process of optimisation. This process continues until a defined convergence criteria is met (e.g. specified maximum number of generations is reached), where after the best individual in the population serves as the ultimate solution to the problem.

This study thoroughly investigates the inner workings that drive the algorithm, after which an algorithm is presented to face the challenges of structural optimisation. This algorithm will be concerned only with sizing optimisation; geometry, topology and shape optimisation is outside the scope of this research. The objective of this optimising problem will be to minimise the weight of the structure, it is assumed that the weight is inversely propotional to the cost of the structure. The motive behind using a genetic algorithm in this study is largely due to its ability to handle discrete search spaces; classical search methods are typically limited to some form of gradient search technique for which the search space must be continuous. The algorithm is also preferred due to its ability to efficiently search through vast search spaces, which is typically the case for a structural optimisation problem.

The genetic algorithm's performance will be examined through the use of bench-marking problems. Benchmarking is done for both planar and space trusses; the 10 - and 25 bar truss problems. Such problems are typically analysed with stress and displacement constraints. After the performance of the algorithm is validated, the study commences towards solving real life practical problems. The first step towards solving such problems would be to investigate the 160 bar truss benchmarking problem. This problem will be slightly adapted by applying South African design standards to the design, SANS (2005). This approach is more realistic, when compared to simply specifying stress and displacement constraints due to the fact that an element cannot simply be assigned the same stress constraint for tension and compression; slenderness and buckling effects need to be taken into account. For this case, the search space will no longer simply be some sample search space, but will consist of real sections taken from the Southern African Steel Construction Handbook, SAISC (2008). Finally, the research will investigate what is needed to optimise a proper real life structure, the Eskom Self-Supporting Suspension 518H Tower. It will address a wide variety of topics, such as modelling the structure as realistically as possible, to investigating key aspects that might make the problem different from standard benchmarking problems and what kind of steps can be taken to over-come possible issues and errors.

The algorithm runs in parallel with a finite element method program, provided by Dr G.C. van Rooyen, which analyses the solutions obtained from the algorithm and ensures structural feasibility.

Uittreksel

Strukturele Optimisasie via Genetiese Algoritmes

S.A. Appelo

Departement Siviele Ingenieurswese,

Universiteit van Stellenbosch,

Privaatsak X1, Matieland 7602, Suid Afrika.

Tesis: MScIng (Siviel)

Desember 2012

Die ontwerp van staal strukture moet 'n sekere optimalisasie proses in sluit wat die aanvanklike ontwerp ontwikkel na 'n meer ekonomiese finale ontwerp, terwyl die nuwe ontwerp nog steeds aan al die aanvanklike ontwerp kriteria voldoen. 'n Kandidaat optimeringstegniek wat voorgestel word deur hierdie navorsing is die genetiese algoritme. Die genetiese algoritme (GA) is 'n optimaliserings tegniek wat geïnspireer was deur evolusionêre beginsels soos die oorlewing van die sterkste (ook bekend as natuurlike seleksie). Dit werk deur die skep van 'n bevolking van individue wat 'kompeteer' met mekaar om dit te maak na die volgende generasie. Elke individu bied 'n oplossing vir die probleem. Oorlewende oplossings wat voortplant deur middel van die volgende generasie is tipies 'beter' of 'fikser' as die individue wat uitgesterf het, dus word 'n proses van optimalisering word saamgestel. Hierdie proses gaan voort totdat 'n bepaalde konvergensie kriteria voldoen is (bv. 'n gespesifiseerde aantal generasies), waar na die beste individu in die bevolking dien as die uiteindelike oplossing vir die probleem.

Hierdie studie ondersoek die genetiese algoritme, waarna 'n algoritme aangebied word om die uitdagings van strukturele optimalisering aan te spreek. Hierdie algoritme het alleenlik te doen met snit optimalisering; meetkunde, topologie en vorm optimalisering is buite die bestek van hierdie navorsing. Die motief agter die gebruik van 'n genetiese algoritme in hierdie studie is grootliks te danke aan sy vermoë om diskrete soek ruimtes te hanteer; klassieke soek metodes word gewoonlik beperk tot 'n vorm van 'n helling tegniek waarvoor die soektog ruimte deurlopende moet wees. Die algoritme is ook gekies as gevolg van sy vermoë om doeltreffend deur groot soektog ruimtes te soek, wat gewoonlik die geval vir 'n strukturele probleem met optimering is.

Die genetiese algoritme se prestasie sal ondersoek word deur die gebruik van standaarde toetse. Standarde toetse word gedoen vir beide vlak en ruimte kappe, die 10 - en 25 element vakwerk. Sulke probleme word tipies met spanning en verplasing beperkings ontleed. Na afloop van die bekragtiging van die algoritme, word praktiese probleme hanteer. Die eerste stap in die rigting sou wees om die 160 element vakwerk toets probleem te ondersoek. Hierdie probleem sal effens aangepas word deur die toepassing van die Suid-Afrikaanse ontwerp standaarde, SANS (2005) aan die ontwerp. Dit is 'n meer realistiese benadering in vergelyking met net gespesifiseerde spanning en verplasing beperkings as gevolg van die feit dat 'n element nie net eenvoudig dieselfde spanning beperking vir spanning en druk toegeken kan word nie; slankheid en knik effekte moet ook in ag geneem word. In hierdie geval sal die soek ruimte nie meer net meer eenvoudig 'n sekere teoretiese soek ruimte wees nie, maar sal bestaan uit ware snitte wat uit die Suid Afrikaanse Konstruksie Handboek kom, SAISC (2008). Ten slotte sal die navorsing ondersoek instel na 'n standaard Eskom Transmissie toring en dit sal 'n wye verskeidenheid van onderwerpe aanspreek, soos om die modellering van die struktuur so realisties as moontlik te maak, tot die ondersoek van sleutelaspekte wat die probleem verskillend van standaard toets probleme maak en ook watter soort stappe geneem kan word om moontlike probleme te oor-kom.

Die algoritme werk in parallel met 'n eindige element metode program, wat deur Dr GC van Rooyen verskaf is, wat die oplossings ontleed van die algoritme en verseker dat die struktuur lewensvatbaar is.

Acknowledgements

I would like to express my sincerest gratitude to the following people and organisations:

Mr. E van der Klashorst for being my study leader and for your time, patience and support -
Dr. G.C. van Rooyen for helping with the technical side of programming and for providing the finite element method program -
Prof. A.A. Groenwold for insight into solving problems -
I thank my mother, C.E. Appelo, and my father, C. Appelo, for their support and faith in me -
I thank my uncle, H. Appelo, for proofreading this thesis -
Eskom for six years of financial support -
Centre for Development of Steel Structures for providing a laptop, internet access, printing quota and an additional computer monitor -
Centre for Development of Steel Structures for sponsoring a trip to the SAISC Steel Awards in Johannesburg -
South African Institute of Steel Construction for selecting me to attend the Steel Awards and all the events surrounding the awards -
The University of Stellenbosch for sponsoring a trip to Johannesburg to attend a course on the Design of Light Industrial Buildings and Connections presented by SAISC -
The University of Stellenbosch for sponsoring my attendance to the International Association for Bridge and Structural Engineering conference which was held in Bellville.

Dedications

This thesis is dedicated to Sophia van Zijl.

Contents

List of Figures

List of Tables

Listings

Nomenclature

Assume the following values, unless otherwise specified:

$E =$	2.1x10^5 MPa
$f_y =$	355 MPa
$G =$	77x10^3 MPa
$\phi_{st} =$	0.9

Variables *(All variables are subject to context)*

A	Area
b	Width
c_m	Fitness scaling constant
CP	Crossover points
C_r	Compressive capacity
C	Large constant or combination
d	Density
D	Deflection
E	Modulus of elasticity
e	An event
f	Stress, unless specified as a function
F	Force
g	Generation counter or inequality constraint function
G	Shear modulus of steel
GA	Genetic Algorithm
h	Equality constraint function
H	Schemata
I	Moment of inertia or specific individual

J	St. Venant's Torsion constant of a cross section
k	Effective length factor, unless otherwise specified
l	Substring length
L	Individual string length
m	Number of realisations for a specific schema
M	Number of unknowns or number of members in a truss
n	Population size or number of dimensions
N	Number of nodes
NFT	Near Feasibility Threshold
o	Order
p	Penalty function
p_c	Probability of crossover
p_d	Probability of being destroyed
p_m	Probability of mutation
p_s	Probability of survival
P	Permutation, unless otherwise specified
r	Real value or a subset
R	Radius of gyration
$\mathbb{R}^n$	Real number set with n dimensions
S	Search space
t	Time
T	Thickness
T_r	Tensile capacity
v	Number of constraints violated
V	Convergence velocity or global displacement in y direction
v_0	Principle $v-v$ axis coordinate of the shear centre with respect to the centroid of the cross section
u_0	Principle $u-u$ axis coordinate of the shear centre with respect to the centroid of the cross section
U	Global displacement in x direction
W	Width to thickness ratio or global displacement in z direction
z	Base 10 integer

$\mathbb{Z}^n$	Integer number set with n dimensions
Λ	Search parameter
δ	Defining length, unless otherwise specified
ϕ	Objective function
ϕ_p	Penalised objective function
σ	Stress
ψ	Composite constraint function
ζ	Fitness function

Vectors

$\mathbf{A}(t)$	Population at a time t
$\mathbf{V}$	Vector space

Subscripts

all	All/complete
$allow$	Allowable
ave	Average value
eff	Effective
$feas$	Feasible
g	Gross
i	Refers to an individual
max	Maximum value
min	Minimum value
off	Offline performance
on	Online performance
p	Penalised
r	Resistance
sum	Total sum of all the values in a set
v	With respect to the $v-v$ axis
u	With respect to the $u-u$ axis
y	Yield

Superscripts

s	Scaled

Part I

Contextual Information

Chapter 1

Background

1.1 Problem Statement

Structural optimisation poses many challenges; challenges such as dealing with discrete and extremely vast search spaces. An algorithm for structural optimisation, which can handle vast discrete search spaces, must be investigated and implemented. Solutions found must be better than just satisfactory. This study must serve as the preliminary phase to develop a tool with which real life structures can be optimised in an automated design fashion.

1.2 Objectives

The aim of this research is to introduce to this department a useful tool towards efficient structural design optimisation; to research the use of genetic algorithms as such a tool for robust structural optimisation. Although it is not intended to be viewed as a computer programming thesis, a large part of this thesis involves programming in order to investigate the algorithm. Analysis and design procedures will be integrated. An analysis iteration should therefore indicate a fragmentary improvement when compared to the previous iteration, as a result of the newly optimised design variables. The algorithm must be able to optimise structures which consist of truss/bar elements; an element with one degree of freedom in the element's axial direction. Such structures must have a defined geometry, topology and element shape definition. Structures to be optimised can be both planar or space trusses. The optimisation objective is to minimise the weight of the structure whilst adhering to displacment and stress constraints. The final step is to implement the South African design code to serve as constraint.

This study forms part of a larger research initiative which will eventually investigate a multi-objective problem; the reliability-based optimisation of steel structures. This study only considers the materials cost, however the larger scope will take into account the construction cost, the life cycle cost, sustainability and maintenance.

1.3 Motivation

The genetic algorithm, according to Rajeev and Krishnamoorthy (1992), is the best candidate for structural optimisation due to the efficient manner in which it handles discrete search spaces; most design variables in structural optimisation have a discrete nature. Moreover, solutions found by a genetic algorithm will be both mathematically and practically feasible.

Consider that a typical design process would start off with selecting initial member sizes based on experience, past knowledge or architectural requirements. This selection process typically results in an iterative procedure. The next step involves creating an analytical model that is an idealised model of the structure's shape, element sizing, topology and loading. This model will generate the structure's response, which in turn will again be used to determine element sizing that would satisfy the ultimate and serviceability limit state constraints. The number of conceivable design solutions exponentially increases with the number of design variables and the size of the search space for each of these design variables. Consider only a simple problem such as a ten bar truss and a discrete section list of, for example, 40 sections. The number of solutions for this search space is 10^{40}. This number is incredibly large; it is, for example, a few thousand times more than the number of estimated stars in our galaxy (Wagner, 2000). It would be unreasonable to expect an engineer with any number of years' experience to be able to choose the optimum truss from such a selection. However, there are a number of ways in which near optimal results for such a problem can be found within minutes, one of which is through implementing a genetic algorithm. Coello *et al.* (1994) claim that the genetic algorithm provides good solutions, even when compared to complex and specialised methods.

Furthermore, consider that structural optimisation can be viewed as a profitable tool and should become part of the standard design process. The increase in available computing power and the world's tendency towards efficient, efficacious and green designs are promoters of such an optimised design approach. Little bits of saving can accumulate to a significant quantity in large scale projects. With awareness comes the understanding that resources are scarce and in some cases even rapidly tending towards depletion. This calls for greener construction methods and using lower quantities of materials which are both efficient and economical. In the case of steel structures, one way of using lower quantities of materials can be achieved through sizing optimisation.

Chapter 2

Introduction

"This preservation of favourable variations and the rejection of injurious variations, I call Natural Selection." - Charles Darwin

Optimisation is a concept which humans seem to apply naturally in order to spend less energy, be as comfortable as possible and to minimise pain. The basic concept of optimised design is concerned with utilising the restricted obtainable resources in such a way as to maximise the profit or gain. Haftka and Gürdal (1992) described such a design as “the best feasible design according to a preselected quantitative measure of effectiveness”. In other words, an optimisation procedure aims at finding the best existing and available solution by seeking the ‘perfect’ trade off between all the given constraints. This trade off, or settlement in some cases, must result in the most propitious outcome for the given resources. This process of optimisation should occur within an economically and timely fashion and produce results that are better than just satisfactory.

It is believed that Galileo was the first person who was concerned with structural optimisation, as is apparent in his studies on the bending strength of beams. Other scientists such as Bernoulli and Lagrange, to name but a few, also aimed at finding the ‘best’ profiles for structural elements that would adhere to a set of strength constraints. Eventually a whole new discipline developed in engineering, commonly known as structural optimisation (Coello *et al.*, 1994). This is a study that is concerned with economical sizes which satisfy given constraints and requirements for design purposes.

In recent times, with the dawn of computers, engineers turned to automated structural design. This allowed for the same quantity of work to be done more accurately and in less time. The question however arises, to what degree of sophistication and complexity can computers aid in design? It seems the future of design aims at completely automating structural design. (Coello *et al.*, 1994)

This study will investigate a computer-based design approach for plane and space trusses with one dimensional elements, which must be optimised in a discrete fashion. Continuous optimisation methods prove to be inadequate for the sizing optimisation of steel trusses due to the nature of available steel sections which forms a discrete, rather than continuous set. Solutions are mostly not optimum for the case where member sizes from the continuous set are simply rounded to the nearest available steel profile section(Coello *et al.*, 1994). Groenwold *et al.* (1999) found that for the 160 bar problem, see section 12, the continuous minimum mass was found to be 1337.8kg. However, the minimum mass increased to 1420.7kg when these continuous solution values were rounded to the nearest available sections that are commonly manufactured. This is 60.9kg heavier than the discrete solution found by the genetic algorithm.

Most optimisation techniques which can handle discrete search spaces are limited to specific types of structures and therefore lack generality. Goldberg and Samtani (1986) were, evidently, the first to suggest and use the genetic algorithm as a tool for structural optimisation.

The genetic algorithm offers a solution for both the aforementioned challenges as it readily deals with discrete search spaces and is easily extended to deal with different types of structures which involve minimal adjustments to the algorithm (Coello *et al.*, 1994). The algorithm basically exists in two realms, the phenospace and the genospace, with a direct analogy to phenotype and genotype. Genetic operations occur in the genospace and function evaluations in the phenospace. The fitness function acts as a mediator between these two spaces. The way in which the design variables are encoded, the coding scheme, is also a link between the realms. The coding scheme serves as a means to map individuals from the genospace to the phenospace and vice versa (Krishnamoorthy *et al.*, 2002). It can be inferred that the procedure consists of a problem-dependent and a problem-independent part, with links between the two parts. This segregation of algorithmic parts is very useful as it enables for a core section of the algorithm to be programmed (for the problem-independent part), which never has to be adjusted again and can be applied repeatedly in the same manner for different types of problems.

A common problem between conventional optimisation techniques is their failure to differentiate between global and local optima. The simplest method with which other optima can be found in conventional search techniques, is through restarting the search at some random point and then to check whether the search leads to a new improved optima. This problem is amplified when the search space becomes discrete. For problems with many design variables, the probability of finding the optimum with such an approach decreases to a point where it will be necessary to do a complete exhaustive search, e.g. an enumerative search. For such cases the efficiency of the search drastically deteriorates to a point where such searches become completely impractical (Haftka and Gürdal, 1992). The genetic algorithm does not necessarily guarantee a global optimum solution, however near optimal solutions are found with relative ease (Erbatur *et al.*, 2000).

The genetic algorithm also differs from conventional methods in that it deals with a population of available solutions, instead of just one solution. The algorithm operates in a probabilistic fashion, rather than deterministically. Each unique individual within the population serves as a potential solution for the given problem, where these solutions are encoded as genes. A collection of genes forms a chromosome, and a collection of chromosomes forms an individual. The genetic algorithm process, along with its analogy to genetics, are thoroughly described later in the text.

Genetic algorithms have a limit to the number of design variables which it can effectively handle, if the encoding scheme is binary. The reason is due to the strings (individuals) becoming too large. Consider a problem with 5000 design variables, and string size of 10. The string size is dependent on the accuracy required for the problem for the case of continuous design variables and the number of discrete options for discrete design variables. For the case of discrete variables, the substring length must ensure that each point in the search space is accessible by the algorithm. Then, for the scenario above, the string length for one individual would be 50 000 bits long. This becomes a large encoding scheme when it is kept in mind that the algorithm uses a population of individuals. The upper bound to where the algorithm is no longer effective is still not certain; the main limitation to this bound will be the amount of available computing power and will therefore not be a set value.

2.1 Outline

The study will commence with a quick glance at optimisation in general, where after an extensive literature review is provided in order to give context to the algorithm. Therefore, the algorithm is first approached from a theoretical point of view with test functions and artificial landscapes. The focus then shifts to discrete optimisation, which incorporates a finite element analysis. The algorithm is then validated by means of benchmarking problems, first only taking into account stress and displacement constraints. The benchmarking problems are the standard 10 bar plane truss and 25 bar space truss. These problems serve to validate and illustrate the usefulness of the algorithm. The study then adapts the algorithm in order to optimise the 160 bar truss problem, which will implement the South African code of design, instead of prescribed constraints. The section list for this case changes from some standard benchmarking list, to the equal leg angle section list in the Southern African Steel Construction Handbook (SAISC, 2008). This problem, with 38 design variables, serves to illustrate the power of the algorithm. Finally, a real life structure is investigated and future research is discussed. The purpose of discussing the future research is to provide insight as to why this research is important. In other words, this chapter will highlight the relevance of this investigation with regard to the basis that it has established, which could lead to a whole series of other applications.

Part II

Literature Review

Chapter 3

Towards Optimisation

There is often no single approach that guarantees an optimised solution for a given problem; therefore there is a wide variety of optimisation techniques developed for solving different kinds of problems. Classical optimisation is synonymous to mathematical programming, some examples of such techniques are calculus methods, geometric and quadratic programming (Rao, 2009).

An engineering system typically consists of a set of quantities, or variables. Some of these variables are pre-assigned parameters, however other variables are free to change in order to produce a better system. Such variables can be grouped into vectors which form a design space, or a search space. Each point in such a space is a design point which represents a solution. Solutions can be both feasible or infeasible (possible or impossible). Solutions to engineering problems typically lie embedded within regions which are surrounded by infeasible solutions within search spaces that are so large, it is unfathomable (Rao, 2009). There are therefore three main components in an optimisation procedure: the design variables, objective function and constraints.

Design variables are those parameters within the search which are adjustable, that would eventually allow for the structure to be optimised. Therefore, these parameters could offer a set of solutions for a given problem. This set could be useful in the case where different solutions need to be considered due to reasons such as financial implications, practicability of construction and time constraints, to name but a few. Design variables cannot be assigned arbitrarily, they have to adhere to a set of requirements in order to produce a solution which is acceptable, or possible; i.e. lie within the feasible region. Such requirements are termed design constraints.

Design constraints are restrictions to the given problem in order to ensure feasible and acceptable outcomes. These restrictions can also be viewed as requirements or limitations.

The objective function specifies criteria for the optimisation process and is ruled by the nature of the design problem. It serves as a filter in order to find solutions. In cases where there are more than one criterion, the problem metamorphoses into a multi-objective optimisation problem.

Therefore, to summarise, an optimisation procedure varies design variables in order to obtain the peaks within the objective function whilst adhering to the limitations of the design constraints.

In the case of classical structural optimisation, constraints would typically take on the form of prescribed stresses and/or displacements and the objective function could for example describe the weight of the structure, where an objective could be to minimise the weight, which indirectly minimises the cost of the structure.

3.1 An Illustrative Optimisation Problem

The following optimisation problem, taken from Spillers and MacBain (2009), serves as an illustrative introduction towards optimisation:

Consider the structure shown in figure 3.1. The aim of this optimisation problem is to vary the height (H) and the diameter (d) of the two members in such a way that the structure is as light as possible whilst still being able to carry the load. Furthermore, the stresses that develop in the members must not exceed the yield stress (f_y) and the members are not allowed to buckle. In other words, the objective of the problem is to minimise the weight of the truss whilst adhering to stress and buckling constraints. Equations 3.1.1 to 3.1.4 follow from basic structural engineering principles.

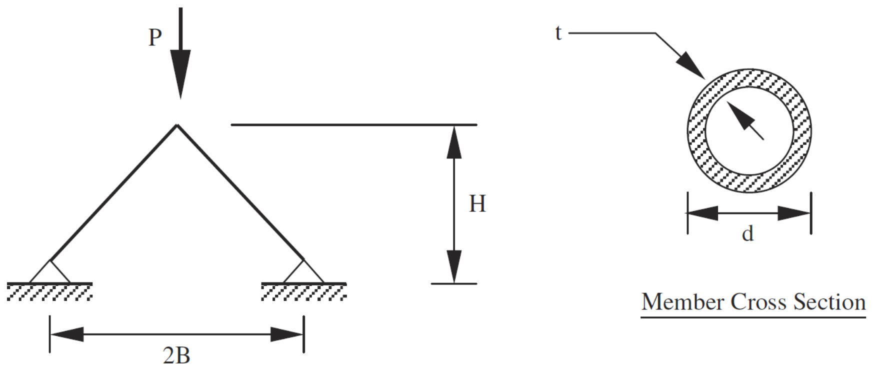


Figure 3.1: Two bar plane truss problem (Spillers and MacBain, 2009)

The second moment of inertia is:

$$I = \frac{\pi}{64}\left[(d+t)^4 - (d-t)^4\right] = \frac{\pi t d}{8}\left(d^2 + t^2\right) \tag{3.1.1}$$

The force in a member:

$$F = \frac{P}{2}\frac{\sqrt{B^2 + H^2}}{H} \tag{3.1.2}$$

The stress in a member:

$$\sigma = \frac{F}{A} \tag{3.1.3}$$

The buckling stress in a member:

$$\sigma_{cr} = \frac{\pi^2 EI}{AL^2} \tag{3.1.4}$$

The volume of the truss is simply $V = 2AL = 2\left(\pi dt\right)\sqrt{H^2+B^2}$, where L is the total length of the two bars. Therefore, the objective function to be minimised is:

$$\phi = 2\left(\pi dt\right)\sqrt{H^2+B^2} \tag{3.1.5}$$

Where the problem is subjected to stress and buckling constraints:

$$g_1 = \frac{P}{2}\frac{\sqrt{H^2+B^2}}{H}\frac{1}{dt\pi} - f_y \leqslant 0 \tag{3.1.6}$$

$$g_2 = \frac{P}{2}\frac{\sqrt{H^2+B^2}}{H}\frac{1}{dt\pi} - \frac{\pi^2 E\left(d^2+t^2\right)}{8\left(H^2+B^2\right)} \leqslant 0 \tag{3.1.7}$$

Figure 3.2 offers a graphical solution to the problem. The contour lines are different volumes for the structure for varying heights and diameters. Each contour represents one constant volume. The optimised solution is shown where the stress and buckling constraints intersect. One can, in this case, simply read off the height and diameter for the optimised structure from the graph. The marked green region illustrates other solutions that satisfy the constraint criteria.

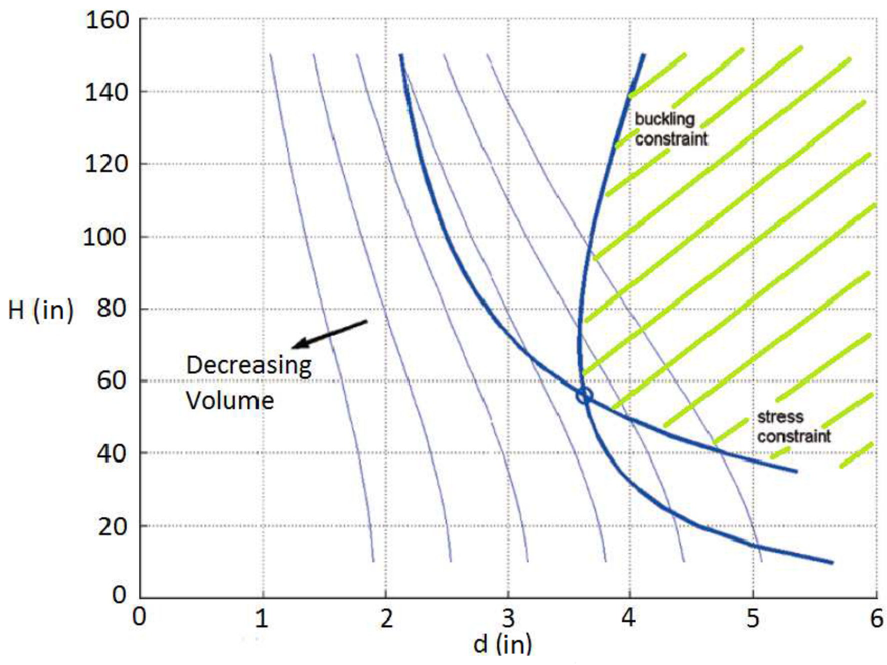


Figure 3.2: Graphical solution of illustrative problem (Spillers and MacBain, 2009)

It should be noted that the lightest structure is not necessarily the cheapest structure. In some cases the fabrication costs, wastage, repetition of elements and so forth, might make a simpler, but heavier design a more economical design.

3.2 Standard Formulation

Find $\mathbf{X} = \left\{ \begin{array}{c} x_1 \\ x_2 \\ \vdots \\ x_n \end{array} \right\}$ which minimises the function $\phi(\mathbf{X})$ such that

$$g_j(\mathbf{x}) \geq 0, \quad j = 1, ..., n_g \tag{3.2.1}$$

$$h_k(\mathbf{x}) = 0, \quad k = 1, ..., n_h \tag{3.2.2}$$

where $\mathbf{X}$ is an n-dimensional vector that contains the design variables (named the design vector), $\phi(\mathbf{X})$ is the objective function, $g_j(\mathbf{X})$ is the inequality constraints and $h_k(\mathbf{X})$ the equality constraints. The formulation above is a constrained optimisation problem, there are simply no constraints for the case of an unconstrained optimisation problem (Rao, 2009). Haftka and Gürdal (1992) suggest normalisation in order to remove boundless variations. For example, consider the constraint:

$$g = \sigma_{allow} - \sigma \geq 0 \tag{3.2.3}$$

The numerical outcome of the above is dependent on the stress units, for which reason the outcome may be great or small. The magnitude of the outcome can be controlled with normalisation:

$$\bar{g} = 1 - \frac{\sigma}{\sigma_{allow}} \geq 0 \tag{3.2.4}$$

This method will be applied to the penalty functions, discussed later in the thesis.

3.3 General Comments on Search Spaces

Design variables can be divided into 2 categories, continuous ($\mathbf{X} \in \mathbb{R}^n$) or discrete ($\mathbf{X} \in \mathbb{Z}^n$). $\mathbb{R}$ refers to real numbers, $\mathbb{Z}$ refers to integers and n, in this case, refers to the number of design variables or dimensions to a problem. Certain optimisation models might contain a mixture of continuous and discrete design variables (Rothlauf, 2011). A search space is defined by its design variables. Search spaces can therefore be divided into continuous and discrete spaces. A continuous one dimensional search space between the numbers 1 and 4 may be represented as follows:

$$S = \{x | 1 \leqslant x \leqslant 4\} \tag{3.3.1}$$

A discrete one dimensional search space can be represented as shown in equation 3.3.2.

$$S = \{1, 2, 3, 4\} \tag{3.3.2}$$

A random experiment in statistics is such an experiment where, even though the procedure is repeated in an identical manner each time, the outcomes will typically vary from trial to trial. The set of outcomes that can be obtained from the random experiment is termed a sample space. An event is the occurrence of some subset of the sample space, denoted as e. The union of two events are denoted as $e_1 \cup e_2$ and is illustrated in figure 3.3.

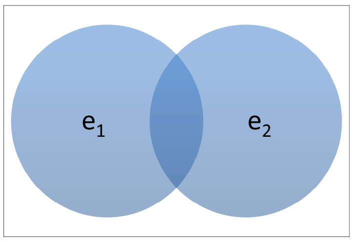


Figure 3.3: Two events in a sample space

In this figure the rectangular area represents the sample space and all areas that are blue form part of the events. The intersection of these two events can be denoted as $e_1 \cap e_2$ and is represented only by the darker region in figure 3.3 (Montgomary and Runger, 2007). A search space can be seen as a sample space and solutions as different events. For example, the event e_1 can represent the event where all the solutions have stresses in the structure within the allowable stress range and the event e_2 can be seen as the event where all the solutions have displacements within the serviceability requirements. All feasible solutions are therefore represented by the area $e_1 \cap e_2$. The solution found by an optimisation technique must be within this feasible region.

Traditionally, the optimum of a function is found where the gradient is equal to zero. This optimum would be accepted if it lies within the feasible region, as discussed above. For example, take into consideration a two dimensional function with one independent variable is illustrated in figure 3.4. Here the optimum for y is easily found by equating the derivative of the function to zero, the optimum is indicated by the dashed line. The difficulty with a discrete search space, as will be thoroughly explained later in the text, is that it does not contain any gradients with which to find the optimum.

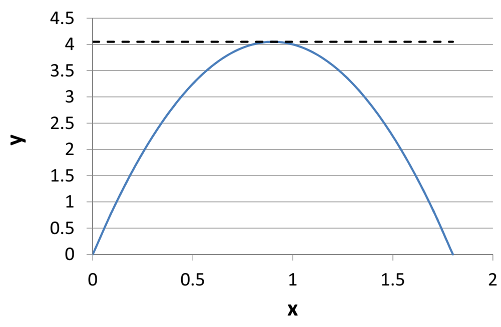


Figure 3.4: Two dimensional line with peak at zero gradient

Counting the number of solutions in a search space provides insight into the scale of structural optimisation problems. Consider the following problem: Given ten different element sizes and a structure consisting of 4 elements, how many sets of four elements can be selected from the list of ten sections that are all of different size? Let the numbers 1,...,10 represent different section sizes. For this case, assume that the order in which sets are formed is considered, therefore $\{1,2,3,6\} \neq \{1,2,6,3\}$. According to equation 3.3.3, 5040 permutations can be found.

$$P_r^n = n \times (n-1) \times (n-2) \times ... \times (n-r+1) = \frac{n!}{(n-r)!} \tag{3.3.3}$$

Where P is the number of permutations of subsets of r elements selected from n different elements (Montgomary and Runger, 2007). Now consider the same problem, however without considering the order in which sets can be formed. According to equation 3.3.4, 210 combinations can be found.

$$C_r^n = \binom{n}{r} = \frac{n!}{r!(n-r)!} \tag{3.3.4}$$

Where C is the number of combinations that has subsets of size r that can be selected from a set of n elements (Montgomary and Runger, 2007). Note that there are fewer combinations than permutations, as some combinations are equal. For example $\{1,2,3,4\} = \{3,2,4,1\} = \{3,4,1,2\}$ and so forth. The difference between combinations and permutations is therefore that order is not considered for combinations. However, for a structural optimisation problem, different elements (design variables) are allowed to have the same size and the order is of significance, as a different order would produce a new structure. Therefore, finding a solution in the form of $\{3,3,1,1\}$ would be acceptable, if it be feasible, and $\{3,3,1,1\} \neq \{1,1,3,3\}$. In other words, each element now has ten sizes to choose from, instead of only from a remaining set once a size is removed. The search space size now suddenly increases with almost 5 000 times, from the original 210 combinations to $4^{10} = 1\,048\,576$.

Finally, it is important to note that search space sizes increase exponentially as the number of elements in a structure and the number of sizes to choose from increase. A structure with 10 elements will have a search space size of 10 billion, approximately 9 500 times larger than the search space of

4 elements. 9 500 times is quite a large increase when it is kept in mind that the number of elements was only increased by 6 and the number of sizes to choose from remained the same. Considering that 10 elements is a small number for a realistic structure and therefore increasing the number of elements to one hundred. This increase will result in a search space size that is 95 trillion times larger than the original. To put this into perspective, if one times increased is equivalent to an increase in distance of 1 mm, then the total increase in length would be approximately 11 times the distance from Earth to Pluto and back.

3.4 Complexity

Computer optimisation algorithms can be grouped according to their *difficulty*. Difficulty is defined by the least amount of computation time needed to solve a problem. The amount of computation time needed to solve an n dimensional problem is a function of *time* and *space complexity*. Time complexity simply refers to the amount of time needed to execute a problem, this is typically expressed by the number of iterations and steps to convergence criteria. Space complexity refers to the amount of physical memory needed to run a problem. Therefore, problem difficulty increases as time and space complexity increase. Complexity adds another dimension to the optimisation problem; not only is a problem difficult to solve based on its search space and nature, but also due to limited physical capabilities of a computer. Complexity classes ranges from class P to NP Hard, where P in this case is the abbreviation for polynomial and NP is the abbreviation for non-deterministic polynomial time (Rothlauf, 2011).

3.5 Structural Optimisation

Structural optimisation can typically be divided into 3 main groups, refer to figure 3.5: (Auer, 2005)

- Topology Optimisation
  - Adjusting the element-node connectivity in order to establish an optimal layout.
- Size Optimisation
  - Adjusting element sizes.
- Shape Optimisation
  - Shape optimisation of the structure concerns changing the shape of the structure without changing the topology. Element shape optimisation is concerned with finding the best profiles for elements.

This research is concerned with sizing optimisation.

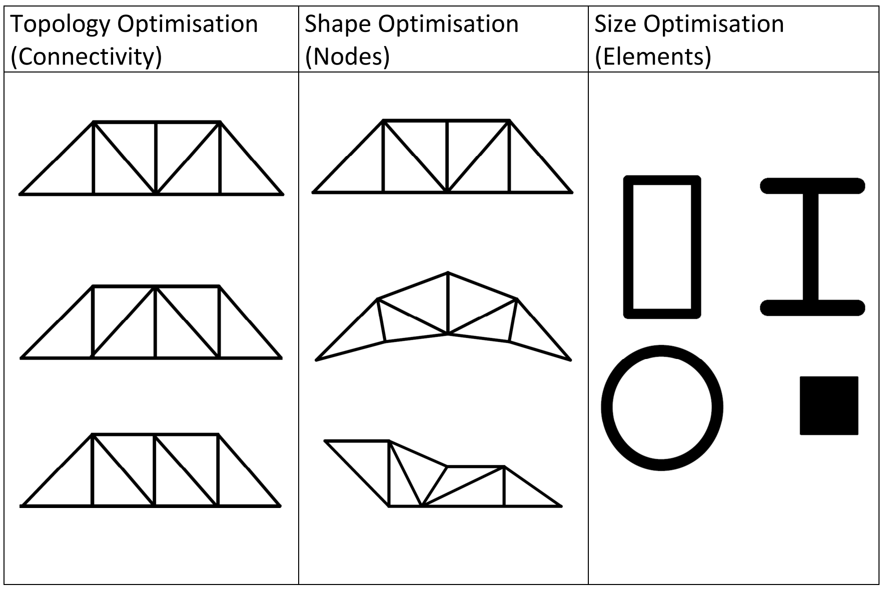


Figure 3.5: Structural Optimisation according to Auer (2005)

Chapter 4

Genetic Algorithms

"Natural selection is a mechanism for generating an exceedingly high degree of improbability" - Sir Ronald Aylmer Fisher

The genetic algorithm owes its existence to John Holland, who's research aims were to thoroughly understand and describe the methods of natural adaptation and then to design an artificial system that operates in the same way (Haftka and Gürdal, 1992).

4.1 Introduction

The goal of optimisation has typically been to find the true optimum; it was concerned with whether a method was converging rather than to explicitly focus on the process of betterment (which seems to be the case in nature). For example, human nature suggests that perhaps perfection is too much to accomplish, but instead it might be enough just to be better relative to others. This form of optimisation seems to take on a whole new set of priorities compared to conventional optimisation. As Goldberg (1989) puts it, the essential objective of optimisation is improvement, the actual optimum is of much less significance in a sophisticated complex system.

The genetic algorithm can be seen as a heuristic method in the sense that the algorithm 'learns' as it gains 'experience'. In other words, previous information is 'remembered' to a certain degree throughout the search and is therefore not completely lost as the algorithm continues to search through the search space. It is based on the same principle of recessive genes; recessive genes might not display themselves physically, but they are still carried by the individual (their information is not lost).

4.2 What is a Genetic Algorithm?

The genetic algorithm is an optimisation technique which searches through a given search space by imitating the processes of natural selection. As the search loops through the iterations, new generations

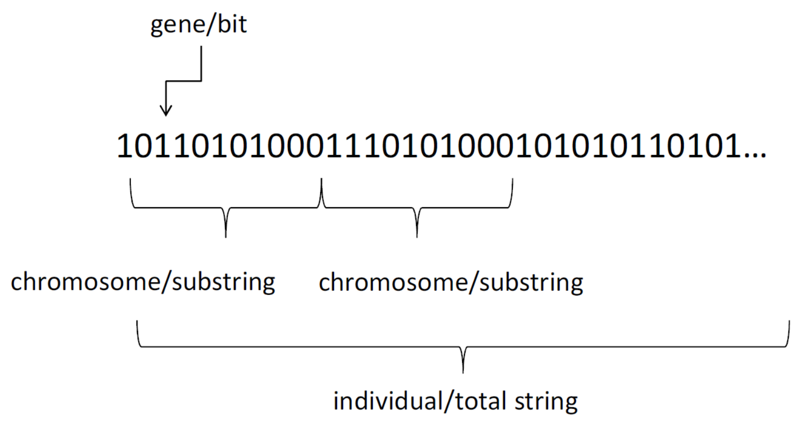


Figure 4.1: Relationship between genetics and computer encoding

of artificial offspring are produced by combining the surviving individuals from the previous generation in a systematic yet randomised interchange of information. This process is accompanied with the occasional random new fragment to keep the search diverse and to steer away from obtaining local optima (Goldberg, 1989). The algorithm generally consists of initial random guesses for a solution of a given problem and a means of finding the better solutions from that initial population (Coley, 1999). The algorithm is founded upon five ideas derived from Darwin's evolution theory: selection, variation, recombination, population and heredity. Each idea is assimilated into the algorithm in order to simulate natural selection (Auer, 2005).

4.2.1 Analogy to Genetics

The terminology used in the study of genetic algorithms is a muddle of the natural and the artificial due to the fact that genetic algorithms stemmed from both natural genetics and computer science. In nature, chromosomes consist of genes which can take on a number of values called alleles, where a collection of chromosomes form an individual. Individuals are the total genetic design of an organism, where the complete genetic package is called a genotype. The complete genetic package in its environment is called a phenotype. In an artificial system, these chromosomes are represented by substrings, where a gene could refer to the bit with an allele taking on the value zero or a one (for the case of binary encoding). Each substring represents an unknown or a dimension of the fitness function. All the substrings combine to form a total string with its natural counterpart being an individual. A collection of individuals form a population and a generation refers to a population at a specific point in time or artificially to the iteration number in the loop. The genotype is called a structure (Goldberg, 1989).

The algorithm optimises a problem through the use of the fitness function. This function has the analogy of being the predator or lack of resources which will govern the probability of a creature, with a specific fitness, to survive. The stronger creatures will have a lower mortality rate on average, when compared to the weaker ones. With a higher probability for stronger creatures to survive comes a higher probability for them to reproduce (stronger) offspring for the new generation. This automatically results in some form of optimisation.

4.2.2 How it the Genetic Algorithm Different to More Traditional Methods?

Instead of working with the actual parameters, the genetic algorithm works with an encoded parameter set, where the natural parameter set is encoded into a string with finite length. In search, the genetic algorithm searches from a population of points simultaneously, in contrast to traditional methods that search from a single point which consecutively corrects a particular solution (Goldberg, 1989). The algorithm uses a fitness function instead of derivatives or similar traditional means. It is based on probabilistic, rather than deterministic guidelines by using stochastic handlers (Coley, 1999).

4.3 Why Use a Genetic Algorithm?

A genetic algorithm could always be outperformed by other methods if sufficient information on the search space is provided. However, to get hold of such information can prove to be nearly as challenging as finding an answer to the problem itself (Coley, 1999).

The algorithm is powerful. Goldberg and Samtani (1986) illustrated that a genetic algorithm can search a vast search space and achieve very near optimal results by only considering an infinitesimal portion of points in comparison to the whole search space. The authors gave context to the power of the algorithm, after performing a ten bar truss benchmarking problem, by saying that the performance of the benchmarking problem was equivalent to searching the world for the best person (population at that time was 4.5 billion) by only interviewing 26 persons before making a decision.

The algorithm is robust even in complex search spaces. It handles a fine balance between efficacy and efficiency, that is it has the ability to fulfill its intended purpose at minimal waste or cost. Robust systems minimise or even completely avoid costly redesigns (Goldberg, 1989).

4.3.1 Advantages

- The genetic algorithm is powerful in its search for betterment, even though the essentials of the algorithm are computationally straightforward (Coley, 1999).
- It is flexible in the sense that it can be applied to a wide variety of problems; examples of such applications are image processing, water networks and spacecraft trajectories (Coley, 1999).
- The algorithm is robust in the sense that it steers the search through the search space, sidestepping the traps set by local optima (Coley, 1999).
- Features of the algorithm such, as self-guidance and self-repair (which are essential to efficient and efficacious optimisation), are scarcely present in the most complex artificial systems (Goldberg, 1989).

- The genetic algorithm is not vitally constrained by limiting assumptions for the search space, examples of such assumptions concern continuity and derivatives (Goldberg, 1989).
- It can quickly scan a vast solution set (Goldberg and Samtani, 1986).
- Bad proposals do not affect the end solution negatively as they are simply discarded.
- The inductive nature of the genetic algorithm means that it does not have to know any rules of the problem - it works by its own internal rules. This is a useful characteristic for complex or loosely defined problems.

4.3.2 Disadvantages

- Even though the biggest driving force behind the genetic algorithm is the evolutionary principles upon which it rests, this is also its biggest limitation. Jaber *et al.* (2006) explain that given evolution's inductive nature, it seems that life does not necessarily evolve towards a good solution, it merely evolves to survive, it simply evolves away from that which does not work. This can result in an 'evolutionary dead end'. Similarly, the genetic algorithm is still always at risk of finding local optima, however it has built-in operators to prevent such outcomes.
- The algorithm may require a large number of iterations, which can become computationally expensive
  - An increase in the number of design variables results in an exponential increase in the number of iterations required
- The performance of the GA is highly dependent on selecting the correct parameters, such as scaling constant and mutation probability (these are discussed later in the text)
  - The algorithm needs to be calibrated for the problem which it must solve
- Most of these disadvantages are common to most modern optimisation techniques

4.3.3 Comparison to Traditional Search Methods

In order to explain the preference to use an evolutionary algorithm or to elaborate on why to use a genetic algorithm, consider the following methods:

4.3.3.1 Calculus-Based Methods

These methods can be subdivided into two categories, direct and indirect. The indirect methods search for local optima through solving sets of non-linear equations obtained by equating the objective function's gradient to zero. Therefore, for a given unconstrained and smooth function, obtaining a probable peak begins with limiting the search to points of zero gradients in all directions. The direct

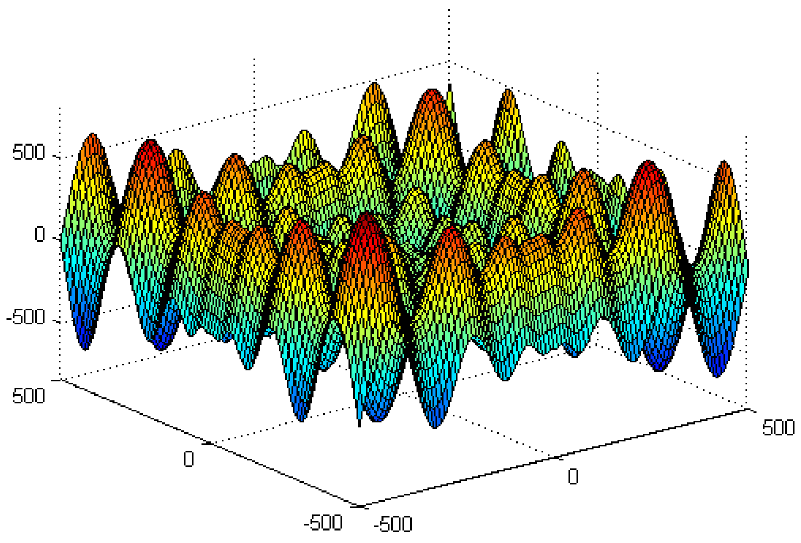


Figure 4.2: Schwefel's Function for two independent variables in 3 dimensions

methods, also known as hill climbing methods, operate by jumping onto the function and are then guided by the local gradient in order to find local optima (Goldberg, 1989).

The calculus-based methods show lack in robustness in that their search is local in scope, the solutions found by these methods are restricted to the proximity of the current point. Consider the function shown in figure 4.2, it is clear that the locality of the scope could produce a false optimum. A random restart mechanism or some other means need to be implemented in order to overcome this deficiency, however this does not necessarily prove to be effective.

Another drawback of the calculus-based methods is its dependence upon existing derivatives with prescribed gradients. Even with the allowance of numerical approximation, this defect can be seen as a great weakness due to the fact that many realistic search spaces have little regard for derivatives and smooth functions (Goldberg, 1989). The calculus-based methods were therefore not considered for this research.

4.3.3.2 Enumerative Search Methods

The idea behind these methods is quite simple; considering a discretised infinite or finite search space, the algorithm searches through the objective function values for all the points in the space one by one (Goldberg, 1989). Although the straightforward approach is appealing, such methods cannot be used for the purpose of this study purely because it is inefficient or impractical.

Realistic search spaces are simply too large. Consider a problem with 10 unknowns, where each unknown needs an accuracy of 1%. This problem will need 100^{10} estimations. Assuming that a computer can compute 2.5 billion estimations per second, then it would take 1268,39 years to complete the run.

4.3.3.3 Random Search Methods

Completely randomised methods were not considered in this study, also due to their lack of efficiency. In the long run, these random search methods' efficiency compares to that of the enumerative search methods. However, take note that these methods do not refer to randomised techniques. The genetic algorithm incorporates a randomised technique which arbitrarily guides the search through the search space. It might seem odd to use a randomised technique for a directed search procedure, but this occurrence is abundant in nature with good results (Goldberg, 1989).

4.3.4 Other Non-Classical Methods

The genetic algorithm was prefered above methods such as Particle Swarm Optimisation and Ant Colony Optimisation simply due to examples in literature which state that the genetic algorithm is a good candidate for structural optimsation (Coello *et al.*, 1994).

Nanakorn and Meesomklin (2001) highlight characteristics from the algorithm which makes it ideal for structural optimisation:

- The solution in a structural optimisation problem is global
- The design variables are typically discrete
- The optimisation problem is constrained
  - The algorithm cannot be directly applied to constrained problems, however it can be indirectly applied by means of penalty functions (see section 4.6).

4.4 How Do Genetic Algorithms Work?

Genetic algorithms operate by handling strings. Collections of strings have different names, depending on the function of the strings. These names range from a population to an individual, where a population contains a number of individuals and an individual contains a number of chromosomes. There are a number of ways to code these data structures, one of the simplest ways is through binary numbers, where an allele value would either be a 1 or a 0, these refer to bit values.

The foundation of the algorithm rests upon a few main operators, these are discussed below.

4.4.1 Selection

Selection needs to be able to distinguish, not only the fit from the unfit, but also the fit from the fitter or the 'good' from the 'very good'. The reason why selection cannot simply take all the top

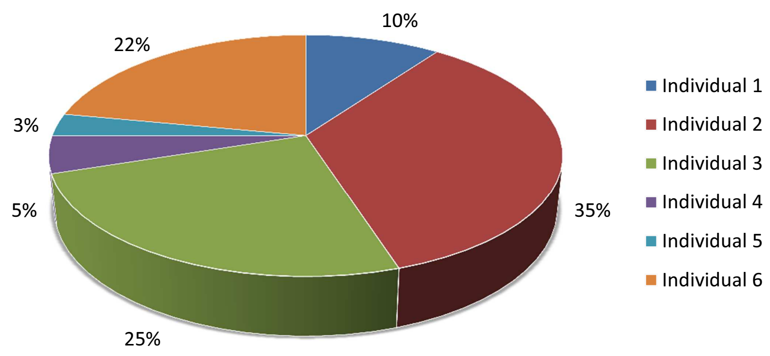


Figure 4.3: Roulette wheel selection illustration

performing individuals from a population is due to the fact that it will cause the algorithm to converge prematurely or in the natural sense, loose important diversity. Selection is a method that tries to imitate natural selection by awarding better performing (fitter) individuals a higher probability to be selected, thereby giving these individuals a greater chance to pass on their information to the next generation (Coley, 1999).

4.4.1.1 Roulette Wheel Selection

One of the simplest ways of selection is using a biased roulette wheel analogy, also known as 'fitness-proportional selection'. Roulette wheel slot sizes are attributed to individuals in a population in relation to their fitness. The circumference of the circle must sum to the total sum of fitnesses for all the individuals. Each slot, as shown in figure 4.3 for a populatio of 6, is sized in such a way that the percentage represents the ratio of that individual's fitness to the total population fitness. 'Spinning the wheel' is done by simply generating a random number and multiplying it with the population fitness. Individual fitnesses are then added one by one until the roulette wheel value is reached, this can also be visualised as the slot in which the ball finally stops. Fitter individuals therefore have a greater probability to be selected due to the larger slot sizes that they were awarded (Coley, 1999).

An individual's probability to be selected is:

$$p_{select,i} = \frac{\zeta_i}{\Sigma\zeta} \tag{4.4.1}$$

There are numerous other methods with which selection can be implemented, for example tournament selection.

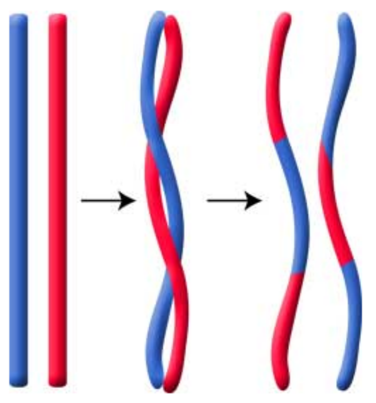


Figure 4.4: Crossover (Venter, 2012)

4.4.2 Elitism

Fitness-proportional selection cannot guarantee that the fittest individuals of a population will propagate through to the next generation, but in fact may (with a very small probability) fail to select them all together . The most general case would be the occasional oversight of the fittest individual. Even though this might be valuable for some problems, as it allows for faster exploration of the search space, it might be a drawback for others. A genetic algorithm must handle a reasonable balance between exploration and exploitation. Greater exploitation speeds up the algorithm, but decreases the probability of finding the true optimum. Elitism speeds up the algorithm by allowing elite members to pass through to the next generation without being subjected to selection and thereby not losing important information. This individual will also not be touched by crossover or mutation (Coley, 1999).

4.4.3 Crossover

Crossover is analogous to reproduction in that it permits the exchange of information to form new offspring. Crossover only takes place with a given probability called the crossover probability. Two parent strings that are selected through selection undergo crossover once it has been established that crossover must indeed occur. Crossover takes place at a random location on the string. For example, given that the crossover location is 4 for the following, the children will look as follows:

Parent 1: 1 1 0 1 | 0 1 1	Child 1: 1 1 0 1 1 1 0
Parent 2: 1 0 0 1 | 1 1 0	Child 2: 1 0 0 1 0 1 1

The illustration above is an example of one point crossover; however, crossover can occur at a number of crossover locations. Crossover promotes exploration of the search space.

4.4.4 Mutation

Mutation allows, usually with a very low probability, an occasional small random change in an individual (string). It is a strategy to avoid premature loss of information and convergence to local optima (Goldberg, 1989).

It operates by visiting every bit within a string and changing a 1 to a 0 or a 0 to a 1 for a given prescribed probability. This prescribed probability is (as with crossover probability) problem dependent, with a higher probability for some and a lower probability for others. Mutation rates are typically in the order of 0.001; mutating on average 1 bit for every 1000 bits visited. Coley (1999) suggests the following mutation rates:

$$\frac{1}{n\sqrt{L}} \leq p_m \leq \frac{1}{L} \tag{4.4.2}$$

where n is the number of individuals in the population and L is the total string length.

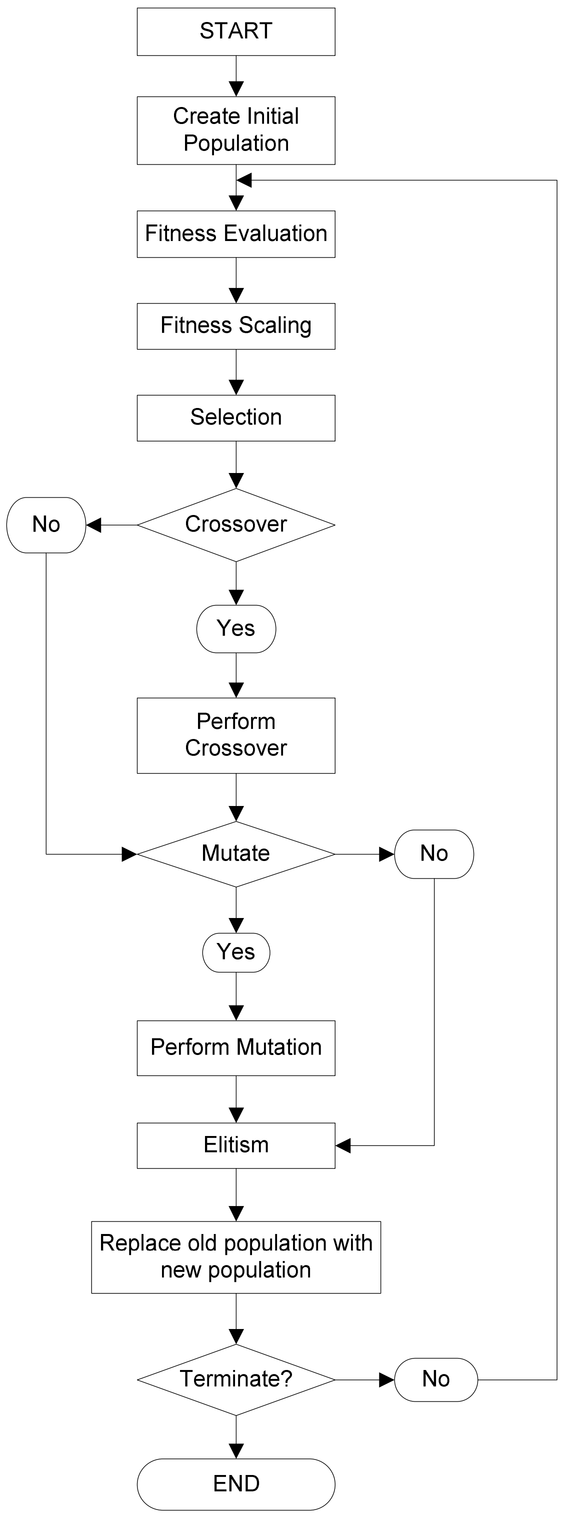


Figure 4.5: The genetic algorithm basic flow diagram

4.5 Object and Fitness Functions

A genetic algorithm is a maximisation procedure, however, in numerous optimisation problems the objective is naturally better expressed as a minimisation function (i.e. of some expense) rather than a maximisation function (i.e. of some profit or utility). Nonetheless, the problem cannot simple be converted to a maximisation problem by taking the negative of the objective function as the algorithm cannot operate on negative values. Even if an objective function is more naturally expressed as a maximisation function, some check still needs to be built into the function in order to ensure positive and feasible outcomes. This function is called a fitness function, a function which converts the objective function into some function that the genetic algorithm can understand. In other words, the objective function must be mapped to some fitness function which the algorithm can use.

4.5.1 Decoding Problems

Binary strings can translate into some integer value, for example 1011 is 11. However, for continuous problems this integer value, 11, must be converted into a real value. Alternatively, real-valued parameters can be used within the genetic algorithm itself, however this will involve changes to the basic operators of the algorithm. The general way to achieve this transformation for a fixed string length is through linear mapping. This linear mapping procedure is used in test functions (Coley, 1999).

- Convert the binary representation to an integer of base 10 and name this integer z
- Transform the integer to a real number through linear mapping

$$r = mz + c \tag{4.5.1}$$

  - m and c refer to the position and dimensions of the space

- Solve two simultaneous equations to obtain m and c

$$r_{\min} = mz_{\min} + c \tag{4.5.2}$$

$$r_{\max} = mz_{\max} + c \tag{4.5.3}$$

  - The minimum value a binary string can take is 0000...0 = 0

$$\therefore z_{\min} = 0$$

  - The maximum value a binary string can take is:

$$z_{\max} = 2^l - 1 \tag{4.5.4}$$

- From the two simultaneous equations:

$$m = \frac{r_{\max} - r_{\min}}{z_{\max} - z_{\min}} \tag{4.5.5}$$

  - Substituting values for $z_{\max}$ and $z_{\min}$

$$m = \frac{r_{\max} - r_{\min}}{2^l - 1} \tag{4.5.6}$$

- Rearrange equation 4.5.2 to obtain:

$$c = r_{\min} - m z_{\min} \tag{4.5.7}$$

  - But $z_{\min} = 0$

  $\therefore c = r_{\min}$

- The transformation equation can therefore be written as:

$$r = \frac{r_{\max} - r_{\min}}{2^l - 1} z + r_{\min} \tag{4.5.8}$$

- $\therefore$ 1011 for a range of 1 $\leqslant$x$\leqslant$ 10 and a substring length of 4
  - $z = 11$
  - $r = \left(\frac{10-1}{2^4-1}\right) 11 + 1 = 7.6$

The next thing to consider is accuracy; the next integer after 11 is 12, which translates to binary as 1100. There is no other number between 1011 and 1100. Transforming 12 to the real set gives $r = 8.2$ for a substring length of 4. It is clear that this poses a fundamental accuracy problem, given that there is an infinite amount of numbers between 7.6 and 8.2. The only known techniques of improving upon the accuracy are by increasing the string length and reducing the search space size.

4.5.1.1 Multi-Parameter Problems

An individual for a multi-parameter or multi-dimensional problem consists of more than one chromosome, therefore more than one substring. Substrings are simply concatenated to form a string (Coley, 1999). Other than considering the genetic reproduction analogy, defense for such an approach is that operators (section 4.4) operate on individuals and not on chromosomes (complete strings, not substrings). In other words, crossover takes places between individuals and not chromosomes.

$$L = \sum_{j=1}^{M} l_j \tag{4.5.9}$$

M is the number of unknowns, l is the substring length and L is the total string length.

Substrings need not be of similar length, which allows for accuracy fine tuning for specific parameters.

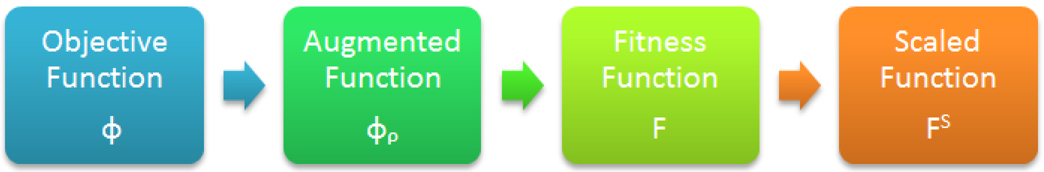


Figure 4.6: Approach to fitness (Galante, 1996)

4.5.2 Approach to Fitness

A means of describing the population fitness is to consider its level of saturation. Saturation describes where the population fitness lies with respect to the best fitness thus far (Galante, 1996).

$$\text{saturation} = \frac{\sum_{i}^{n} \zeta_i}{n\zeta_{\max}} 100 = \frac{\bar{\zeta}}{\zeta_{\max}} 100 \tag{4.5.10}$$

where ζ is the fitness function value. The objective is to minimise the structure's weight, where after the objective function is penalised and is therefore transformed into an augmented function (figure 4.6) called the penalised objective function. The objective of the algorithm is to maximise the fitness, however, due to reasons explained in section 4.5.3 the objective changes to maximising the scaled fitness. In general, the objective function value for a given individual solution can be expressed as:

$$\phi_i(\mathbf{x}) = \sum_{j=1}^{M} dA_j L_j \tag{4.5.11}$$

where M is the number of members in the truss (structure) and $\mathbf{x}$ is a possible solution vector to the problem. d is the density of the material, A is the area of an element and L is the length of an element. Constraints for stress and displacement are typically expressed as:

$$\frac{\sigma_M}{\sigma_{allow}} - 1 \leqslant 0 \tag{4.5.12}$$

$$\frac{D_N}{D_{allow}} - 1 \leqslant 0 \tag{4.5.13}$$

where subscript *allow* indicates the allowable, M the number of members, N the number of nodes and D the deflection.

4.5.2.1 Static Fitness

Goldberg (1989) suggests obtaining a fitness function value by subtracting the objective function value from a very large constant, C. This constant is typically in the order of 10^5.

$$\zeta_i = C - \phi_{i,p} \tag{4.5.14}$$

Galante (1996)'s approach differs from equation 4.5.14 by the implementation of a relative rate between individuals that are maintained as initially expressed by the objective function and a large constant value.

$$\zeta = \frac{C}{\phi_{i,p}} \tag{4.5.15}$$

4.5.2.2 Dynamic Fitness

The foremost shortcoming of the static fitness approach, as was done by Goldberg (1989) and Rajeev and Krishnamoorthy (1992), is that the convergence behaviour could possibly be dependent on C, the large constant value. The objective function value, ϕ, might exceed C for the case where the value of C was chosen too small, which will result in a negative fitness. Normalisation and choosing a larger value for C can correct such an outcome. Oppositely, if C is assigned too large a number, then chromosomes might be assigned similar fitnesses even though their objective function values vary. For example, consider a large constant value of 1 000 000. If it be that the objective function values range from 1 to 10, then the magnitude of the large constant value and that of the objective function values differs too much. The fitnesses assigned to the individuals would all be in the range of 999 990 to 999 999, which excludes 0.99999% of the fitness scale. Consider that, in this case, an objective function value of 1 should supposedly represent a poor fitness value, however, this transformation does not resemble the degree of poor performance of the given value. A solution to the aforementioned, other than fitness scaling (refer to section 4.5.3), is to incorporate a dynamic factor method in which case the fitness is a function of maximum and minimum objective function values for each generation and the specific individual's objective function value under consideration. This approach will ensure that the individual with the highest objective function value (lowest fitness) will be assigned a proportional value to that of the lowest objective function value (Krishnamoorthy *et al.*, 2002).

$$\zeta_i = \phi_{\max} + \phi_{\min} - \phi_{i,p} \tag{4.5.16}$$

Toropov and Mahfouz (2001) suggest a similar function, however the maximum and minimum objectives should be penalised as well. Hence the function develops into:

$$\zeta_i = \phi_{\max,p} + \phi_{\min,p} - \phi_{i,p} \tag{4.5.17}$$

$\phi_{\max,p}$ is the maximum penalised objective value, $\phi_{\min,p}$ is the minimum penalised objective value and $\phi_{i,p}$ is the penalised objective value of individual i, refer to section 4.6. This approach requires the population fitness values to be sorted, where after all the individuals with a fitness below the average fitness value are killed off. Therefore only the upper fittest part of the population remains. Now a new fitness is defined for each individual based on the new highest and lowest fitness values:

$$\zeta^{new} = \phi^{new}_{\max,p} + \phi^{new}_{\min,p} - \phi_{i,p} \tag{4.5.18}$$

The approach magnifies the distances between different top performing individuals, in the same way a map with a smaller scale would emphasise the distance between two places by supplying more information. This method could, however, lead to premature search convergence due to the fact that search loses much diversity.

Coello *et al.* (1994) suggests a fitness that is inversely proportional to the objective function value:

$$\zeta_i = \frac{1}{\phi_{i,p}\left[1000v + 1\right]} \tag{4.5.19}$$

For this case v is the number of constraints violated for a specific solution. v would be zero for the case of no constraint violation, hence the fitness function would be reduced to the inverse of the structure's weight. It is clear that the fitness would decrease as the number of constraint violations increase. A constant of a thousand was found to work best for the ten bar truss problem. Nanakorn and Meesomklin (2001) had the same approach, however not including a factor of a 1000 or the v term. Both approaches reward the same level of punishment for all solutions violating a given number of constraints. Therefore, it could be argued that solutions which are better performing than others are treated too severely and poorer solutions are not penalised enough.

In general, the objective function would be some function of the structure's weight, as it is deduced that the weight of the structure is directly proportional to its cost, hence the cost is indirectly optimised. However, Raj and Kalyanaraman (2005) incorporated actual costs in their objective function by considering material and fabrication costs. Joint costs are dependent on the number of joints or nodes, the number of individuals connected to the joint and the magnitude of forces transferred by the joint. Hence the constraints include material strength, fatigue strength- and deflection limit and buckling strength.

$$\min\ \phi_i = \left(\sum_{k=1}^{N_m} A_k d_k L_k\right) C_{st} + \left(\sum_{j=1}^{n_j}\sum_{r=1}^{n_{mj}} c_r\right) \tag{4.5.20}$$

C_{st} is the cost of steel per kN, n_{mj} is the number of members that connects at joint/node j and c_r per member added to the joint based on the accompanying force. For this case the augmented objective function is given as:

$$\phi_{i,p} = \phi_i\left(1 + P_{i,c}\right) \tag{4.5.21}$$

with ϕ_i as the objective function obtained in equation 4.5.20.

$$P_{i,c} = \sum_{j=1}^{m} \left(\left(C_{j,stress} P_{j,stress} \right) \left(C_{j,deflection} P_{j,deflection} \right) \left(C_{j,implicit} P_{j,implicit} \right) \right) \tag{4.5.22}$$

$P_{i,c}$ incorporates the penalised constraint violations of the individual solution i, for all its members, j, with C_{stress}, $C_{displacement}$ and $C_{implicit}$ representing the constraints violations and P_{stress}, $P_{displacement}$ and $P_{implicit}$ their associated penalty factors.

Krishnamoorthy *et al.* (2002) used the following function for a specific load case:

$$\phi_{p,i}(x) = \left(\sum_{i=1}^{M} d_i A_i L_i \right) \left(1 + \sum_{j=1}^{M+N} k_j c_j^2 \right) \tag{4.5.23}$$

$$c_j = \begin{cases} \max\left(0, \frac{\sigma_j}{\sigma_{j,allow}} - 1 \leqslant 0 \forall j \in [1, M]\right) \\ \max\left(0, \frac{d_{j-M}}{d_{j-M,allow}} - 1 \leqslant 0 \forall j \in [M+1, N]\right) \end{cases} \tag{4.5.24}$$

where k_j is the penalty coefficient, L_i is the length of member i, M the number of members in the structure, N the number of nodes, d the density of the material and A_i the area of member i. Due to string length being directly proportional to the number of design variables, large convergence delays and loss of important information can be expected for large number of design variables. To compensate for this drawback, a method of member grouping is proposed, in which case certain members assume the same size, hence leading to shorter string lengths and a reduced search space size (member grouping will be thoroughly discussed later in the text). Another benefit of this approach is that it allows for the design to stay symmetrical, which is good for constructability of the structure and ensures the structure can handle reversed load conditions, for example wind load from the opposite side as was done in the analysis. The objective function now evolves into:

$$\phi_{i,p}(x) = \left(\sum_{k=1}^{NG} A_k \sum_{i=1}^{M_k} d_i L_i \right) \left(1 + \sum_{j=1}^{M+N} k_j c_j^2 \right) \tag{4.5.25}$$

Member grouping for smaller structures can be done a priori, however Krishnamoorthy *et al.* (2002) suggest member grouping strategies for larger structures due to inaccuracies regarding grouping which could lead to suboptimal outcomes.

4.5.3 Fitness Scaling

It could happen that a few highly fit individuals are created prematurely in the run, causing its offspring to drown other individuals in subsequent generations. This will lead to a huge loss in diversity, producing offspring close to a manner of cloning, which could potentially result in a local optimum. There needs to be some form of a steady state or balance of the power of the highly fit individuals in the early and later stages of the algorithm. In other words, the highly fit individuals must be prevented from hijacking the algorithm in its initial stages, but needs to be able to apply adequate selection pressure to the algorithms in its final stages. Fitness is therefore scaled in order to maintain a

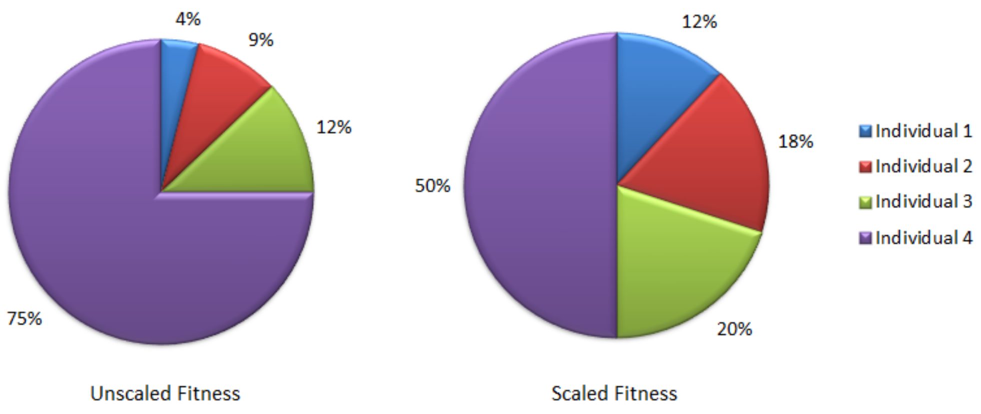


Figure 4.7: Unscaled and scaled fitness roulette wheels

saturation level of about 50% (see equation 4.5.10) which will ensure proper exploitation of the search space and to steer away from premature convergence.

Linear fitness scaling is a method which scales the fitness of individuals to the proximity of the average population fitness. This implies that a certain ratio between the number of highly fit selected individuals and the number of individuals selected with average fitness will be kept at a reasonable proportion, which would be nearly constant. Conventional values for this constant are between 1 and 2; where a value of 2 implies that about twice the number of highly fit individuals will propagate through to the next generation compared to the number of individuals with average fitness. To accomplish this, dynamic scaling of individuals' fitnesses would need to take place by pulling fitnesses closer together in the initials stages and then pushed apart in the later and final stages of the algorithm. The linear transformation:

$$\zeta_i^s(g) = a(g)\zeta_i(g) + b(g) \tag{4.5.26}$$

- ζ_i is the actual fitness of a particular individual
- ζ_i^s is the scaled fitness for that particular individual

It is assumed that the average fitness of a population stays constant:

$$\zeta_{ave}^s(g) = \zeta_{ave}(g) \tag{4.5.27}$$

Additionally:

$$\zeta_{\max}^s(g) = c_m(g)\zeta_{ave}(g) \tag{4.5.28}$$

- c_m is the fitness scaling constant

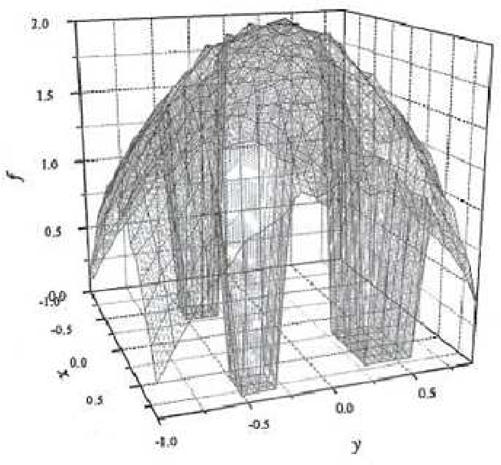


Figure 4.8: Constraints depicted as areas with zero fitness (Coley, 1999)

- $\zeta^s_{\max}$ is the scaled fitness of the fittest member

$$a(g) = \frac{(c_m - 1)\zeta_{ave}(g)}{\zeta_{\max}(g) - \zeta_{ave}(g)} \tag{4.5.29}$$

$$b(g) = (1 - a(g))\,\zeta_{ave}(g) \tag{4.5.30}$$

Implementing the linear scaling can result in negative fitnesses. One way to overcome this, is to set $c_m = 0$ for such cases (Coley, 1999).

There are also other scaling methods such as Sigma Truncation and the Power Law Scale.

4.6 Constraints and Penalty Functions

Constraints split the search space into feasible and infeasible segments. A constraint can be visually understood as regions within a search space where no fitnesses can be allocated, refer to figure 4.8.

4.6.1 Constraint Handling

The genetic algorithm performs best for unconstrained problems (Gahsemi *et al.*, 1999). Problems which are not heavily constrained are quite easily dealt with, the chromosome is decoded and the fitness function awards a fitness to it. The fitness is simply zeroed for cases where there are constraints.

Even though the aforementioned approach seems appealing, it would be ineffective for densely constrained problems and produce many solutions which will simply be discarded. Even if it was not the case for a densely constrained problem, infeasible solutions may carry valuable information which should not be cast-off. A genetic algorithm can however not be directly applied to constrained problems. In order to use the algorithm for engineering applications, the problem must be transformed from a constrained to an unconstrained problem (Coley, 1999). Nevertheless, any final solutions that are obtained from the genetic algorithm need to satisfy all the prescribed constraints in order for it to be feasible. A constraint can partly be classified by its criticality and difficulty. The criticality of a constraint can be described by the degree to which it needs to be satisfied. A constraint is typically formulated as absolute, where it may indeed be more 'soft'. A genetic algorithm allows for the use of 'soft' constraints through the implementation of penalty functions, see figure 4.9. A penalty function acts as a sort of punishment for violating a constraint by decreasing the fitness of the guilty individual. The amount of decrease is in relation to the severity of the violation. The penalty function must not disrupt that equilibrium of exploitation and exploration. The algorithm allows for constraints to be violated, where after it probabilistically selects the best solutions from a population of solutions. The penalty function operates by decreasing the fitness of infeasible solutions relative to the severity of the constraint violation. The difficulty of a certain constraint is directly related to the ratio of the feasible area to that of the sample space area. An increased ratio will result in a lower difficulty. The difficulty of a problem is however also related to the number of constraints (Smith and Coit, 1995).

There are various ways in which a penalty function can be implemented (Yeniay, 2005):

- Death penalty
- Static penalty
- Dynamic penalty
- Annealing penalty
- Adaptive penalty
- Co-evolutionary penalty
- Segregated GA

Penalty methods can typically be divided into 3 groups (Smith and Coit, 1995):

- The first group is called barrier methods, in which case only feasible solutions will be considered.
- The second group consists of partial penalty functions, where penalties only apply to areas which are near the feasibility margin.
- The last group of penalty functions contains global penalty functions. These functions consider the whole sample space (which includes the complete infeasible region).

Penalty methods can crudely be grouped into four strategies with their advantages and disadvantages (Gen and Cheng, 1996):

- Rejecting approach
  - Rejects all infeasible solutions
- Repairing approach
  - Needs a repair procedure
- Modifying genetic operators approach
  - Problem specific with specialised operators
- Penalising approach
  - Converts a problem which is constrained into an unconstrained problem

The first three strategies never generate infeasible solutions which are advantageous; however it has the disadvantage of not searching the infeasible regions as well, which is typically most of the search space. A general requirement for good penalty functions include penalties which concern distance from feasibility, rather than just simply keeping count of the constraints violated. Penalties incorporating such a requirement are better performing. The relationship between feasible and infeasible solutions is important as the penalty value should correspond to this amount. The penalty method is either a function of (Gen and Cheng, 1996):

- The distance from a single infeasible solution
- The relative distance of all current infeasible solutions
- The adaptive penalty term

Combinatorial optimisation uses the Lagrangian Relaxation method (some alteration to the same idea) in which case the difficult constraints are briefly relaxed. Control is kept with an adjusted objective function which keeps the search from completely drifting away from the feasible region (Smith and Coit, 1995).

The standard optimisation formulation is adapted as follows to include penalty (Yeniay, 2005):

$$\phi_{i,p}(\mathbf{x}) = \begin{cases} \phi(\mathbf{x}), & \text{if } \mathbf{x} \in S_{feas} \\ \phi(\mathbf{x}) + \psi(\mathbf{x}), & \text{otherwise} \end{cases} \tag{4.6.1}$$

where $\psi(\mathbf{x})$ is the penalty applied. For the case where no constraints are violated, $\psi(\mathbf{x}) = 0$. $\phi(\mathbf{x})$ is the objective function. S_{feas} refers to the feasible region.

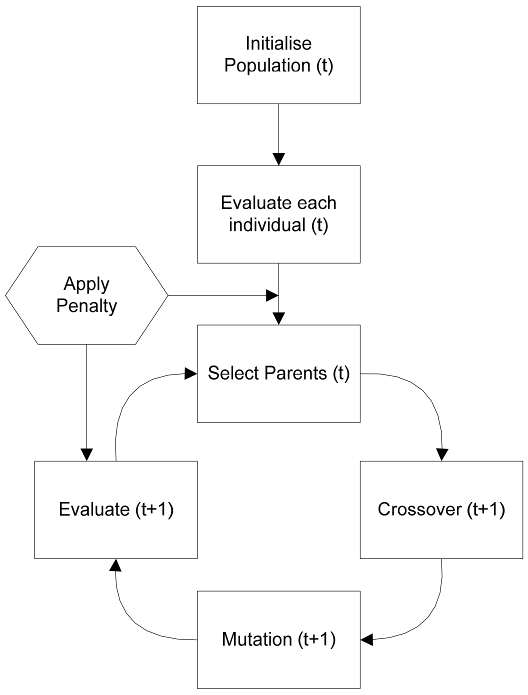


Figure 4.9: Transforming the genetic algorithm from an unconstrained to a constrained problem solver

Another method is a multiplicative function:

$$\phi_{i,p}(\mathbf{x}) = \begin{cases} \phi(\mathbf{x}), & \text{if } \mathbf{x} \in S_{feas} \\ \phi(\mathbf{x})\psi(\mathbf{x}), & \text{otherwise} \end{cases} \tag{4.6.2}$$

For this case, when there is no constraint violation $\psi(\mathbf{x}) = 1$. The better overall performer has been observed to be the additive function. Penalty functions can further be divided into two types: interior and exterior, however the exterior function is generally more preferred. For more information on interior penalty functions, refer to Rao (2009). The motivation behind this preference has to do with the fact that the exterior penalty needs not to be initiated within the feasible region (Yeniay, 2005).

4.6.2 The Exterior Penalty Function

$$\phi_p(\mathbf{x}) = \phi(\mathbf{x}) + \left[\sum_{i=1}^{q} r_i G_i + \sum_{j=q+i}^{m} c_j L_j\right] \tag{4.6.3}$$

G_i and L_j are functions of the constraints $g_i(\mathbf{x})$ and $h_j(\mathbf{x})$. r_i and c_j are penalty parameters. Generally:

$$G_i = \max\left[0, g_i(\mathbf{x})\right]^{\beta} \text{ with } \beta = 1 \text{ or } 2 \tag{4.6.4}$$

$$L_j = |h_j(\mathbf{x})|^{\gamma} \text{ with } \gamma = 1 \,\text{or}\, 2 \tag{4.6.5}$$

The magnitude of the penalty is dependent on r_i and c_j.

4.6.3 Death Penalty Function

In this case the penalty function simply discards any unfeasible solutions.

$$p(\mathbf{x}) = \infty \text{ with } \mathbf{x} \in S - S_{feas} \tag{4.6.6}$$

This method is only effective for a convex search space, see figure 10.2. This approach will be ineffective for highly constrained problems. Two approaches are thoroughly described by Homaifar and Kuri Morelas with Quezada in Yeniay (2005).

4.6.4 Static Penalty Function

The penalty parameters are independent of the generation counter and are kept constant throughout the search. Before the search commences, users must define degrees of violation. (Yeniay, 2005)

$$\phi_p(\mathbf{x}) = \phi(\mathbf{x}) + \sum_{i=1}^{m} C_i \delta_i \text{ where } \begin{cases} \delta_i = 1, & \text{if constraint } i \text{ violated} \\ \delta_i = 0, & \text{if constraint } i \text{ satisfied} \end{cases} \tag{4.6.7}$$

C_i = enforced constant on the violation of constraint i. This category of penalty functions has proven to be less effective when compared to penalisation techniques whose degree of penalty depends on the distance to the feasibility. These penalisation techniques assume that this distance defines accurately the closeness of the solution to the feasible region and that this distance value is significant to the solution fitness (Smith and Coit, 1995).

$$\phi_p(\mathbf{x}) = \phi(\mathbf{x}) + \sum_{i=1}^{m} C_i \psi_i^{\kappa} \text{ where } \psi_i = \begin{cases} \delta_i g_i(\mathbf{x}), & \text{for } i = 1, ..., q \\ |h_i(\mathbf{x})|, & \text{for } i = q+1, ..., m \end{cases} \tag{4.6.8}$$

κ is typically 1 or 2, C_i is determined through scaling or experimentally and g and h are the inequality and equality constraint functions (Smith and Coit, 1995). δ remains as defined in equation 4.6.7.

4.6.5 Dynamic Penalty Function

The main shortcoming of static penalty functions is in the difficulty of determining C_i. The static penalty functions also have contradictory aims in the sense that it allows for exploration in the infeasible regions, however it needs the ultimate solution to be feasible. One way to lessen the difficulties of the improved static search is by incorporating a dynamic feature to the penalty. In this case, the severity of the penalty increases with an increasing distance between the problem outcome and the feasibility region. In this case extremely infeasible solutions might be present during the initial stages of the search, where after extreme penalties will be applied in order to advance the solution to the feasible region and then decreasing the penalty (Smith and Coit, 1995).

$$\phi_p(\mathbf{x},g) = \phi(\mathbf{x}) + \sum_{i=1}^{m} s_i(g)\psi_i^{\kappa} \text{ where } \psi_i = \begin{cases} \delta_i g_i(\mathbf{x}), \text{ for } i = 1,...,q \\ |h_i(\mathbf{x})|, \text{ for } i = q+1,...,m \end{cases} \tag{4.6.9}$$

Caution should be exercised with $s_i(g)$, where $s_i(g)$ is a function of the constraints. For the case where $s_i(g)$ is too merciful the ultimate solution might be infeasible and for the case where it is too severe the solution might be a local optimum as a result of premature convergence. It is suggested to assume $s_i(g) = (C_i g)^{\alpha}$ with $\alpha = 1$ or 2 (Smith and Coit, 1995). Jounes and Houck used a value of $C = 0.5$. Kazarlis and Petridis also formulated a penalty approach, however slightly altered, refer to Yeniay (2005). The problem with all the aforementioned dynamic approaches is the constants which these approaches incorporate. These constants typically have no physical meaning and are simply chosen after it was empirically observed that they produce the best outcome.

4.6.6 Adaptive Penalty Function

The adaptive penalty, as described by Hadj-Alouane and Bean in Smith and Coit (1995):

$$\phi_p(\mathbf{x},g) = \phi(\mathbf{x}) + \sum_{i=1}^{M} \lambda_g\psi_i^{\kappa} \text{ where } \psi_i = \begin{cases} \delta_i g_i(\mathbf{x}), \text{ for } i = 1,...,q \\ |h_i(\mathbf{x})|, \text{ for } i = q+1,...,m \end{cases} \tag{4.6.10}$$

with

$$\lambda_{g+1} = \begin{cases} \lambda_g\beta_1, \text{ if previous N}_\text{f} \text{ generations have infeasible best solution} \\ \lambda_g/\beta_2, \text{ if previous N}_\text{f} \text{ generation have feasible best solution} \\ \lambda_g, \text{ otherwise} \end{cases} \tag{4.6.11}$$

where $\beta_1 > \beta_2 > 1$. M refers to the number of members, g to the current generation number and κ is typically 1 or 2. These constants need to be selected, it might prove difficult to select a good value.

$$\phi_p(\mathbf{x},g) = \phi(\mathbf{x}) + \lambda(g)\left[\sum_{i=1}^{q} g_i^2(\mathbf{x}) + \sum_{j=q+1}^{m} |h_j(\mathbf{x})|\right] \tag{4.6.12}$$

For every generation g, update:

$$\lambda(g+1) = \begin{cases} \left(\frac{1}{\beta_1}\right)\lambda(g), \text{ if Case 1} \\ \beta_2\lambda(g), \text{ if Case 2} \\ \lambda(g), \text{ otherwise} \end{cases} \tag{4.6.13}$$

- Case 1
  - All the best performing individuals of the last g generations are feasible.
- Case 2

  - All the best performing individuals of the last g generations are not feasible.

- Case 3
  - The best performing individuals of the last g generations are a mixture of feasible and unfeasible solutions.

The drawback of this approach concerns defining β_1 and β_2.

4.6.7 Near Feasibility Threshold

The near feasibility threshold (NFT) is the verge where the search can be considered as 'getting warmer'. The penalty function promotes the algorithm to search within the feasibility region and in the near feasibility threshold of the feasible region and discourages search elsewhere. NFT according to Smith and Tate as explained in (Smith and Coit, 1995):

$$\begin{aligned} &\phi_p(\mathbf{x}, g) = \phi(\mathbf{x}) + (\zeta_{feas}(g) - \zeta_{all}(g)) \sum_{i=1}^{m} \left(\frac{\psi_i}{NFT_i} \right)^{\kappa} \\ &\text{with } \psi_i = \begin{cases} \delta_i g_i(\mathbf{x}), \text{for} i = 1, ..., q \\ \left| h_i(\mathbf{x}) \right|, \text{for} i = q+1, ..., m \end{cases} \end{aligned} \tag{4.6.14}$$

$\zeta_{all}(g)$ is the current best solution which is not penalised and $\zeta_{feas}(g)$ is the current best solution which is feasible. These terms serve as adaptive scaling and amalgamate with the near feasibility threshold of iteration i (Smith and Coit, 1995).

$$NFT = \frac{NFT_0}{1 + \Lambda} \tag{4.6.15}$$

Λ is the search parameter which modifies the near feasibility threshold by taking the search history into account. The function will result in a static near feasibility threshold for the most elementary case where the Λ parameter is zero. This parameter can be described as a function of time during the search, for example for generation g, $\Lambda = f(g) = \lambda g$. With $\lambda > 0$ the penalty will increase as the threshold region decreases. A greater λ results in a greater increase in penalty, therewith integrating adaptive and dynamic elements into the search (Smith and Coit, 1995).

4.6.8 Segregated Genetic Algorithm

This algorithm makes use of two distinct penalty parameters in two parallel populations. The main objective of this approach is to eliminate problems concerning premature convergence or no convergence at all due to too low/high penalty parameters. This is accomplished through selecting a low value for the first penalty parameter and a high value for the second in order to achieve a simultaneous convergence approaching from both the feasible and infeasible regions (Yeniay, 2005).

4.6.9 General Comment on Penalty Functions

The overall disadvantage of penalty function methods is concerned with choosing a suitable set of penalty parameters, however penalty functions are decidedly the best approach when dealing with non-constrained optimisers such as genetic algorithms (Yeniay, 2005).

4.7 Why Do Genetic Algorithms Work?

It can be showed that there are specific string configurations that lead to higher fitness or better performance for certain given problems. Two important steps in genetic optimisation are to seek for similarities amongst individuals and to find a connection between these similarities and better performance (Goldberg, 1989).

4.7.1 Schema Theory/Similarity Theory

One string on its own is of no significance; this is due to the fact that only similarities between high performing strings can help navigate the search. The question is therefore, how can a string resemble strings of other string sets with similarities at specific string locations? The answer is through schemata. A schema is a description for a subgroup of strings that has certain similarities (Goldberg, 1989). For the sake of discussion, consider binary encoding {0,1} with a wild card character * which can represent either a zero or a one. Therefore, for a schema to match a given string, every 0 must match with a 0 at a specified location, the same for every 1 and the * can match with either a 1 or a 0. For example, the following scheme:

$\rightarrow$ 0 0 1 * 1 matches { 0 0 1 0 1, 0 0 1 1 1 }

Take note of the fact that the * is merely a device to represent other symbols, this symbol itself is not specifically used in the genetic algorithm. There are m^l different strings of length l for a given character set of m elements, with $(m+1)^l$ schemata. The question that surfaces is, why consider the schemata which will in effect increase the search space rather than just all the different string different strings? For example, a string with length 10 has $2^{10} = 1024$ possible strings (for binary encoding), why then consider $3^{10} = 59049$ schemata instead? Consider that individual strings only provides pieces of information compared to the oceans of new information that is contributed by similarities which will contribute to a more efficient search. The 'magnitude' of this additional information is associated with the number of unique schemata within a given population (Goldberg, 1989).

Schemata are not all of the same magnitude. For example, a schema of 1** is much greater than a schema of 11*, as the one encapsulates a much larger part of the search space. The basic operators (selection, crossover and mutation) have different effects on schemata. Fitter schemata will have on

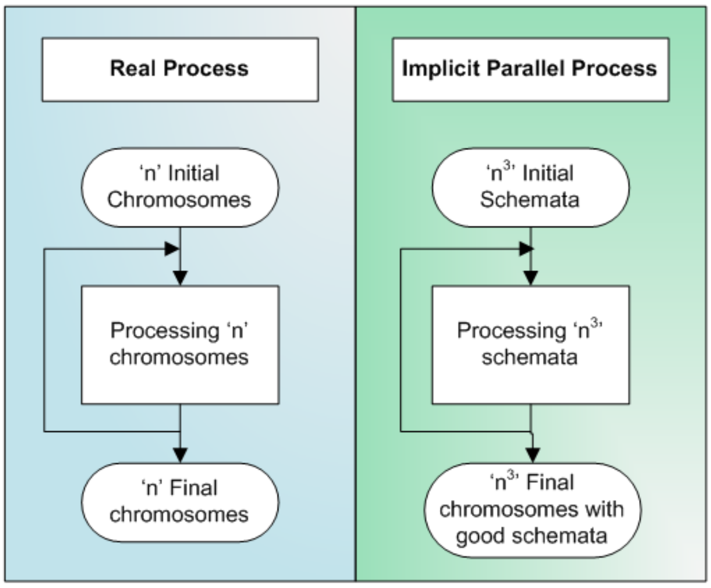


Figure 4.10: Real and implicit parallel process (Galante, 1996)

average more surviving individuals, due to fitter individuals having a higher probability to be selected. However, selection alone does not contribute new points to the search space. One of two things can happen when crossover occurs; it can either leave the schemata intact or destroy it to form a new schema. Consider the following schemata:

$\rightarrow$ 0 * * * 1 and * * * 0 1

The first of the two schemata will probably be destroyed with crossover, compared to the second schema which has a higher chance to remain intact. Therefore, the shorter the defining length of schemata, the higher the probability of survival after crossover. Mutation does not play a significant role in the survival of schemata as it occurs in such low frequencies. The above will be explained in more detail in section 4.7.2. Schemata which are very fit and of short defining length are called 'building blocks' and are propagated through the generations. This occurs with no special memory or bookkeeping, where this whole procedure is called 'implicit parallelism', see figure 4.10 (Goldberg, 1989).

4.7.1.1 Similarity Templates as Hyper-Planes

Consider the bit space from a geometric viewpoint for $l = 3$. Schemata of order three form cube corners and schemata of order 2 form the lines between these corners, refer to figure 4.11. Genetic algorithms can be seen jumping through hyper-planes in the search of betterment (Goldberg, 1989).

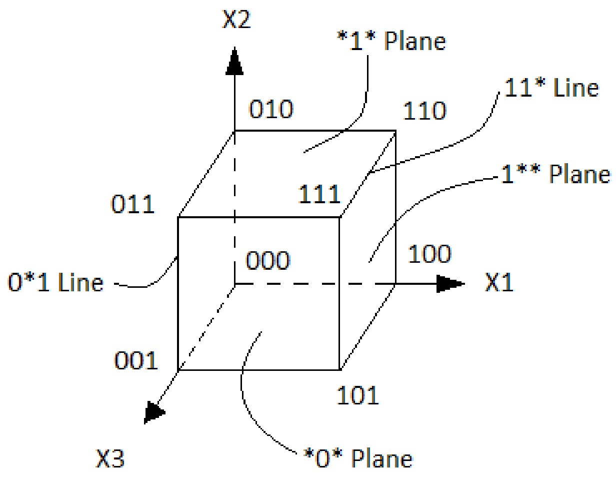


Figure 4.11: Hyper-Planes (Goldberg, 1989)

4.7.2 Fundamental Theorem of Schemata

This section will take a closer look at the growth and decay of schemata within a population subjected to selection, crossover and mutation.

Let $\mathbf{A}(t)$ refer to a population of strings at a specific time, therefore to a specific generation. Also, let schema H have a vector space $\mathbf{V} = \{0, 1, *\}$. For example:

$\rightarrow \quad H$ = *10*0**

There are 3^l schemata for a string with length l for the case of binary representation and in general as already mentioned, $(k+1)^l$ schemata for alphabets of k elements.

4.7.2.1 Types of Schemata

Different schemata are not tantamount; some schemata are better defined than others. For example, 111*0** is more specific than 1******. Additionally, some schemata have a greater span length over the string, compare 1*0**** and 1****0*. This introduces two new concepts, order and defining length. Order is symbolised as $o(H)$, where it denotes the number of fixed positions in a string. Example:

$\rightarrow \quad o(10**01**) = 4$

Defining length is symbolised as $\delta(H)$, where it denotes the span of a schema and is calculated by subtracting the location of the last fixed position from the location of the first fixed position. Example:

$\rightarrow \quad \delta(10**01**) = 5$

The properties of the schemata provide a means to interpret the effect of selection, crossover and mutation on a population (Goldberg, 1989).

4.7.2.2 Effect of Selection

Assume that there are m realisations of a specific schema H within $\mathbf{A}(t)$ at a given time t. This is denoted as $m = m(H,t)$. A string A_i has a probability of $p_i = \frac{\zeta_i}{\Sigma\zeta_j}$ to be selected. At a time $t+1$, for selection with replacement from $\mathbf{A}(t)$, there will be $m(H,t+1)$ schemata.

$$\therefore m(H,t+1) = \frac{m(H,t)\cdot n \cdot \zeta(H)}{\Sigma\zeta_j} \tag{4.7.1}$$

where $\zeta(H)$ represents the average fitness for schema H at time t.

But the average population fitness is $\zeta_{ave} = \frac{\Sigma\zeta_j}{n}$.

$$\therefore m(H,t+1) = m(H,t)\frac{\zeta(H)}{\zeta_{ave}} \tag{4.7.2}$$

From the above, it can be deduced that a schema's growth is dependent on the proportion of average schema fitness and average population fitness. In other words, when the schemata fitness is higher than the average population fitness, then selection will be biased towards that particular schemata by awarding it more individuals at time $t+1$. In this case the schemata will grow. The opposite effect will occur to schemata with average fitness lower than the population average fitness, where the schemata will start to die off. All schemata for a particular population are processed simultaneously, or in parallel (Goldberg, 1989).

4.7.2.3 Effect of Crossover

The algorithm requires a crossover mechanism, because selection does not support exploration of the search space. String $A = 111|1000$ might have the following shemata:

$\rightarrow \quad H_1 = *1*|***0 \qquad \delta(H_1) = 5$

$\rightarrow \quad H_2 = ***|1\,0** \qquad \delta(H_2) = 1$

It is clear that schema H_1 is destroyed with a probability of

$$p_d = \frac{\delta(H_1)}{l-1} = \frac{5}{6} \tag{4.7.3}$$

and has a survival probability of

$$p_s = 1 - p_d = \frac{1}{6} \tag{4.7.4}$$

For the case of H_2, $p_d = \frac{1}{6}$ and $p_s = \frac{5}{6}$. Generally, a schema survives when the crossover site is located outside of $\delta(H)$. Therefore, for single crossover with a probability of p_c:

$$p_s \geqslant 1 - p_c \frac{\delta(H)}{l-1} \tag{4.7.5}$$

Therefore, the effect of crossover on a schema can be described such that the shorter the schema's defining length, the greater is its probability to survive and reproduce (Goldberg, 1989).

4.7.2.4 Combined Effect of Selection and Crossover

The combined effect of selection and crossover on the expected schema H in generation $t + 1$ (assuming that selection and crossover are independent) is:

$$m(H, t+1) \geqslant m(H, t) \frac{\zeta(H)}{\zeta_{ave}} \left[1 - p_c \frac{\delta(H)}{l-1}\right] \tag{4.7.6}$$

In this case, the effect of schema is clear, the survival not only depends on average fitness, but also defining length. Schemata with above average fitness with short defining lengths will grow exponentially (Goldberg, 1989).

4.7.2.5 Effect of Mutation

Mutation occurs with a probability of p_m. All the fixed positions of a schema must survive for the schema itself to survive. In other words, the schema survives when all $o(H)$ fixed positions survives, where each allele has a survival rate of $1 - p_m$ (Goldberg, 1989).

$$\therefore p_s = (1 - p_m)^{o(H)} \tag{4.7.7}$$

(The survival rate for $p_m << 1$ is estimated as $1 - o(H) \cdot p_m$).

4.7.2.6 Overall Effect

The expected number of samples for a schema H is:

$$m(H, t+1) \geqslant m(H, t) \frac{\zeta(H)}{\zeta_{ave}} \left[1 - p_c \frac{\delta(H)}{l-1} - o(H) p_m\right] \tag{4.7.8}$$

Finally it can be concluded that schema with short defining length, low order and above average fitness will be awarded with an increasing amount of individuals (Goldberg, 1989).

4.7.3 Building Block Hypothesis

The complexity of the problem is reduced by the use of schemata; rather than constructing high performance strings, the hypothesis aims to actively build improving strings from the best fragmentary solutions of the former (refer to section 3.4). These best partial solutions are known as building blocks. A building block is a schema which offers a good solution to a sub-problem, as explained in section 4.7.2.6. It can be seen as analogous to genes within the genetic framework (Rothlauf, 2011). The building block hypothesis has showed promising results for a range of problems, including noisy multimodal and combinatorial optimisation problems (Goldberg, 1989).

4.7.3.1 Intra-Building Block Difficulty

This type of difficulty is related to the locality of the search. The problem difficulty increases if the nature of the search space is of such that it leads the search away from the optimum. This is also known as the *deceptiveness* of a sub-problem. A problem is deceptive to the order k if all schemata of order lower than k have a lower fitness compared to the rest, even though they hold fragments of the fittest solution (Rothlauf, 2011).

4.7.3.2 Inter-Building Block Difficulty

Genetic algorithms are a form of recombination-based search; this simply implies that the greater problem is decomposed into sub-problem. Simpler sub-problems are solved instead of solving one extremely complex problem. Such sub-problems can be solved independently, given that the problems were decomposed correctly. However, it might occur that the some sub-problems contribute more to the objective than others, which results in inter-building block difficulties. Additionally, interdependencies arise when problems cannot be effectively disintegrated into perfectly separate sub-problems (Rothlauf, 2011).

4.7.3.3 Extra-Building Block Difficulty

Noise can add difficulty to a problem by altering the objective values. The recombination-based search will make poorer decisions as noise is introduced to the problem. Non-stationary problems cause a similar problem as solutions have dissimilar valuations at different points in time (Rothlauf, 2011).

4.7.3.4 Berthke's and Holland's Walsh-Schema Partitioning Coefficient Transforms

Methods devised to analyse the Building Block Hypothesis can be grouped into two categories based on their approach; the application of dynamic or static methods. The dynamic approach, in alliance with the Minimal Deceptive Problem produces decent results for small problems (the actual approach will not be discussed here). On the other hand, the static approach determines schema averages through transformation methods, which is used to judge the Building Block Hypothesis. In other words, to establish whether high performing schemata of short defining length and of low order propagate through the generations in order to combine and create improved schemata which is longer and of higher order (Goldberg, 1989).

Chapter 5

Test Functions

A Test function, also known as an artificial landscape, has the objective to analyse the performance of a genetic algorithm. The outcome of these functions can be used to systematically rectify and fine tune the internal settings of the algorithm. Therefore, once a genetic algorithm for a specific problem is coded, the algorithm can be 'tested' by checking whether it produces the expected outcome of the function. Internal settings are unique to each problem, these can therefore be adjusted. While these test functions are of great value, they are of little significance to 'real' world problems. A genetic algorithm should be tested with a set of test functions in order to cover various essential landscapes, each with their own features. This set will test different aspects of the algorithm (Coley, 1999):

- Functions with scalable dimensions
  - The function should be able to adjust the number of unknowns if it would be desired
- A unimodal, continuous function, to gain insight to the algorithm's convergence velocity
  - A single peak function, refer to figure 5.1
- Test the algorithm's performance with the absence of a local gradient
  - A step function, refer to figure 5.2
- Test the algorithm's performance when faced with complexity with a multimodal function
  - A multi peak function, refer to figure 5.3

5.1 De Jong's Test Functions

De Jong realised the great value of controlled experimentation with genetic algorithms in neat problem domains. He rid the genetic algorithm of all frills, together with its environment and performance criteria to expose its sheer fundamentals. This allowed him to carry out experiments which aided in

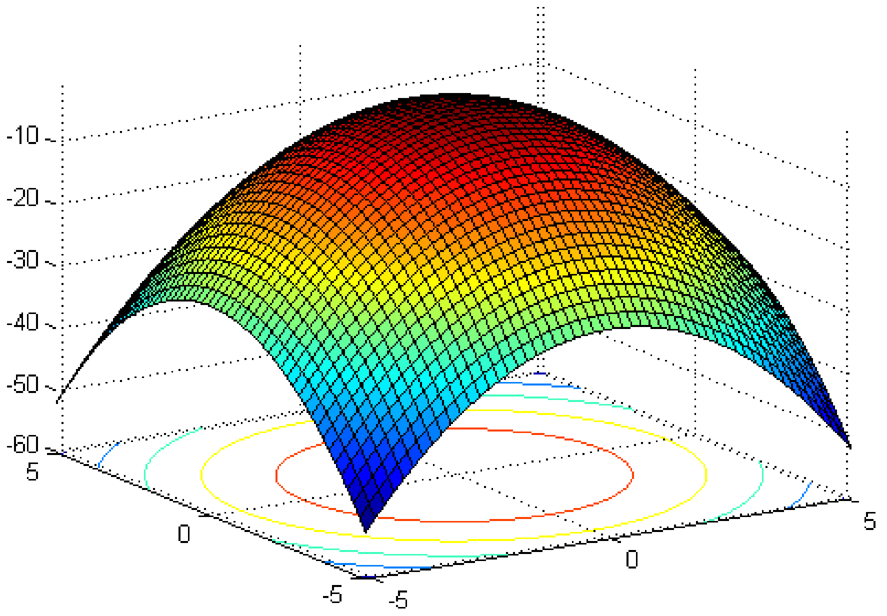


Figure 5.1: Griewank's function with 2 independent variables

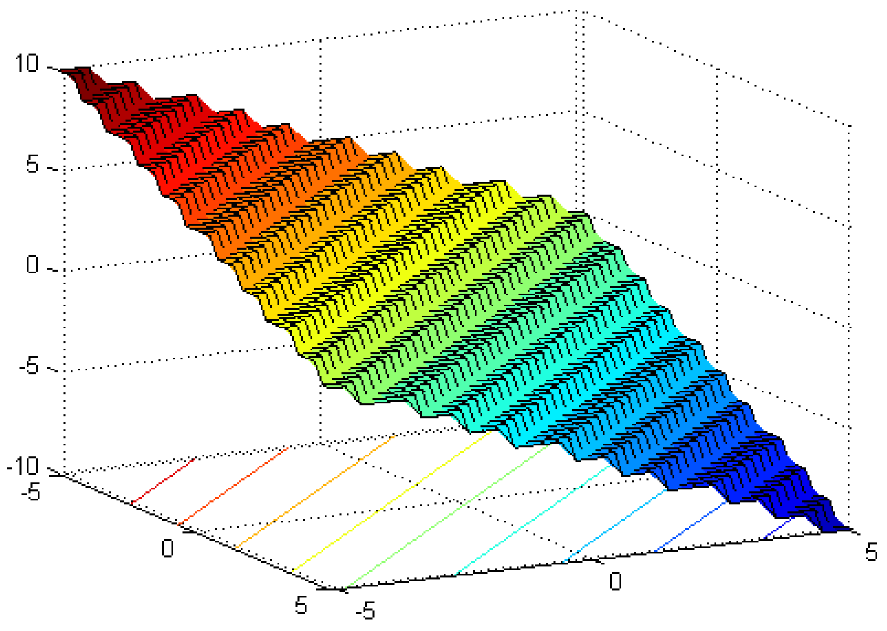


Figure 5.2: Step function with 2 independent variables

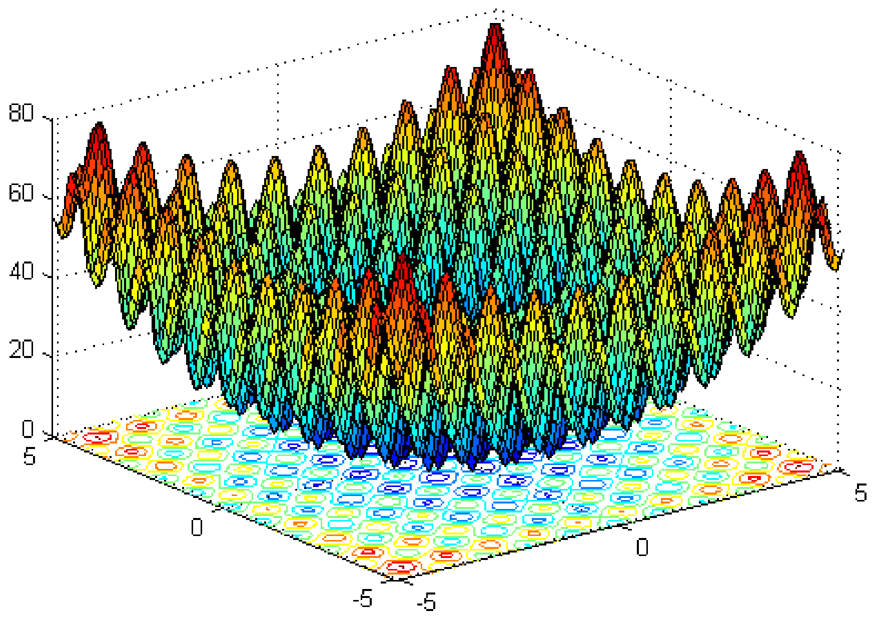


Figure 5.3: Rastrigin's function with 2 independent variables

the further development of genetic algorithm research and its uses. De Jong created a test environment which dealt with minimisation of 5 problems. The functions he used encompassed the following features (Goldberg, 1989):

- Continuous and discontinuous
- Quadratic and non-quadratic
- Convex and concave
- Low and high dimensionality
- Unimodal and multimodal
- Deterministic and stochastic

Refer to Appendix 17.5 for test functions and artificial landscapes.

5.2 Measuring Performance

De Jong carried out much work with regard to the genetic algorithm's performance. De Jong had two gauges for performance which he called online and offline performance. The offline performance (ζ_{off}) refers to the continuous average fitness of the fittest population member (ζ_{max}) and gauges performance (Goldberg, 1989):

$$\zeta_{off}(g) = \frac{1}{g}\sum_{j=1}^{g}\zeta_{\max}(j) \tag{5.2.1}$$

The online performance (ζ_{on}) is simply the average fitness thus far in the algorithm and measures online performance (Coley, 1999):

$$\zeta_{on}(g) = \frac{1}{g}\sum_{j=1}^{g}\left[\frac{1}{N}\sum_{i=1}^{N}\zeta_i(j)\right] \tag{5.2.2}$$

The convergence velocity is given as (Coley, 1999):

$$V = \ln\sqrt{\frac{\zeta^{\max}(g=G)}{\zeta^{\max}(g=0)}} \tag{5.2.3}$$

5.3 De Jong's Conclusions

De Jong constructed a few models (Goldberg, 1989):

- The Reproductive Plan
- The Elitist Model
- The Expected Value Model
- The Elitist Expected Value Model
- The Crowding Factor Model
- The Generalised Crossover Model

These models will not be discussed here, for more information refer to Coley (1999).

5.3.1 Towards Population Size

These studies indicated that larger populations result in improved offline performance, which results in better convergence. This increase in offline performance is due to a bigger pool of diverse schemata that is accessible by the algorithm. However, with an increased population size the online performance is poorer in the early stages of the algorithm. Smaller populations are more agile which results in higher initial online performance (Goldberg, 1989).

5.3.2 Towards Mutation Rate

An increased mutation rate can help maintain diversity by resisting premature allele loss. Too high a mutation rate will however affect the run negatively, resulting in a decrease in offline and online performance. Offline performance begins to mirror random search performance when the mutation rate becomes too high. A mutation probability of 0.5 is simply random search; this is irrespective of the values of the crossover probability and population size (Goldberg, 1989).

5.3.3 Towards Generation Gap

De Jong found that non-overlapping population models provided better results for optimisation, with the major influencing factor being the offline performance (Goldberg, 1989).

5.3.4 Towards Crossover

De Jong also performed tests on crossover probability. De Jong suggested that a crossover probability of 0.6 provides a good balance between offline and online performance. Later studies suggested higher crossover rates with improved selection methods.

The generalised crossover model showed that there was a relation between the number of crossover points (CPs) and performance, increasing the number of points decreases both offline and online performance. The number of distinct operators involved in this process offers an explanation to the observation. For one point crossover, there is a set of l -1 operators. CP $=$ 2 has $\binom{l}{2}$ combinations to select different CPs. Generally there are $\binom{l}{\text{CP}}$ combinations. This implies that as CPs increases, the number of combinations decreases, resulting in a lower probability for selecting a specific operator during a specific cross. This leads to increased mixing and a decrease in structure. In other words, the process becomes random and a significant increase in the loss of important schemata (Goldberg, 1989).

5.3.5 Towards Elitism

De Jong came to the conclusion that elitism promotes local search by sacrificing some degree of global perspective (Goldberg, 1989).

All of the deductions above will be taken into account when running the algorithm in order to achieve the best possible results.

Chapter 6

Advanced Operators

There are a variety of ways to improve the performance and robustness of the algorithm, or to make it more problem specific. The methods below offer solutions to difficulties found in real life problems.

6.1 Combinatorial Optimisation

For many real life problems, the aim of optimisation is not to optimise a simple chain of real valued parameters, but to determine an ultimate ordered output or list as in the case of the Travelling Salesman Problem. In this problem, a salesman has to travel the shortest route between a collection of cities and has to visit each one (ideally, a city should not be visited more than once). Structural design is also an example of a combinatorial optimisation problem (Coley, 1999).

The biggest challenge with combinatorial optimisation problems and genetic algorithms is the potential for the algorithm to choose infeasible tours due to crossover and mutation. For illustration purposes, refer to figure 6.1 where each dot represents a city.

Possible tours might be:

$\rightarrow$ f c e g | a b d h

$\rightarrow$ a b f g | c d h e

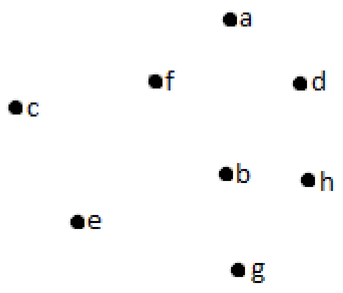


Figure 6.1: Combinatorial optimisation: Find the shortest distance between the cities

For a single point crossover at for example the indicated point, the following children are obtained:

$\rightarrow$ f c e g c d h e

$\rightarrow$ a b f g a b d h

It is clear that both these children are infeasible, as the routes that they describe visit some cities more than once and others not at all. Therefore, the crossover operator needs to be changed so that it will only generate feasible results. A city can only be visited once in the case of strings with fixed lengths. Coley (1999) explains that there are a number of techniques to deal with the crossover problem, one of them being to simply proceed with crossover as usual and then to discard any infeasible outcomes. However, the aforementioned is not a very effective technique. It should be noted that the location of gene and its allele value are not unrelated. In other words, the location and value of a bit are both independently significant. In fact, order is the only thing that is of importance for the travelling salesman. Preferably, crossover and mutation must operate in such a way as to both produce feasible results and combine building blocks that produce fitter offspring (Coley, 1999).

6.2 Niches and Species

Niches and species can be used to locate alternative solutions. To find the best solution for a problem that is large and complex might be to find an answer that is only in the proximity of the true global optimum. Even so, some problems need a series of solution options. For these problems the options which dwell in the vicinity of the optimum need to be found. In these cases it is highly probable that such solutions are separated by 'bad' regions. Therefore, contrary to the norm, the intent here is to find local optima. However, an interesting question arises, why seek local optima when any point close to the global optimum is highly likely to have a higher fitness? To answer the question, consider the following example presented in Coley (1999):

Consider a structural problem where x is the slant of the roof and f is some inverse cost function, then it can be understood that each optima represents a noteworthy solution, refer to figure 6.2. These solutions are indeed good, even though they are not the best. They offer a number of financial schemes for a variety of different roof constructions. If cost was the only constraint, then the global optimum x^* would have been the best. However, for any additional constraints such as specifications on the slant (enforced by the practicability of construction, requirements from the client or visual qualities) then any of the other solutions (x_1, x_2, x_3) could be of interest, even though they are more expensive (have a lower fitness). Granting that there are a number of solutions in the vicinity of the optimum that are less expensive than the local optima, their proximity might be too limiting on the slant of

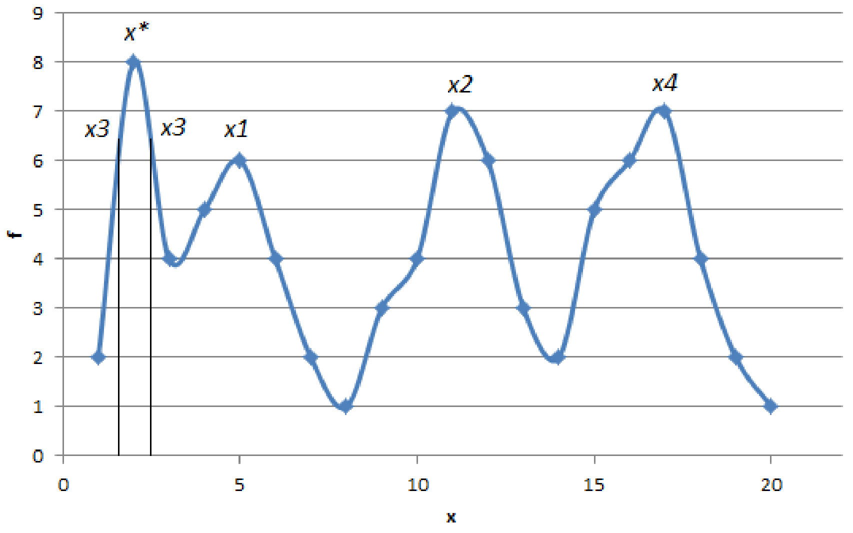


Figure 6.2: Local optima (Coley, 1999)

the roof and could therefore proof to be infeasible. On way of finding these local optima in complex search spaces, is through niches and species. In the natural sense, to subdivide a search space into niches by species (subgroups of a population) is a common phenomenon. When it comes to genetic algorithms, the niches imply some form of a fitness sharing and the species imply limits and restrictions on mating partners. Partners who will be able to breed must be of certain resemblance and be related to a satisfying extent (Coley, 1999).

6.2.1 Sharing

Consider two gambling machines and a certain number of players. If both machines pay out the same amount within the same time frame then players can divide themselves equally to play on these machines, where each player will receive maximum prize money (given that the money won at a machine is distributed equally amongst the players, the money is shared). However, in the case where one machine pays out more than the other in similar time intervals, then more players should move over to that machine so that each will still receive maximum prize money. It is obvious that if the players were to stay as they were in the first case, then one half of the players will receive more prize money than the other half. If it was just a free for all and no sharing was involved, then all the players would sooner or later end up at the machine that pays out the most money. The players that have to share their winnings at the machine that pays out less will learn that even though they might not win as much money in total, they still receive the same amount individually because there are less players to share with. In this case it is sensible to form a niche (Coley, 1999).

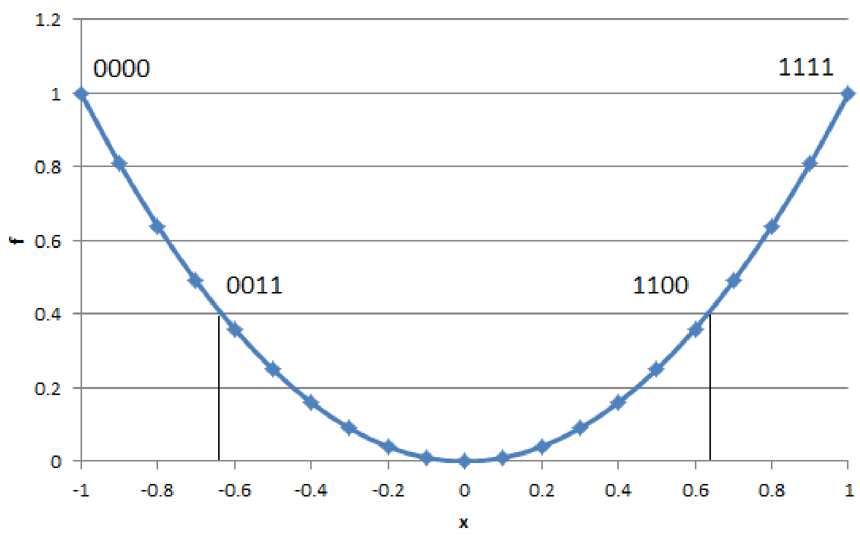


Figure 6.3: Speciation (Coley, 1999)

6.2.2 Species

Normally, mating does not occur between differing species. Thus far, the discussions on the genetic algorithm have not considered such restrictions. There might be an advantage to consider species in the algorithm, consider the following strings with single point crossover:

$\rightarrow$ 0 0 | 0 0 = -1

$\rightarrow$ 1 1 | 1 1 = 1

For both of these points on the x axis, the fitness is $\zeta(x) = 1$, refer to figure 6.3. However, crossover of these two highly fit strings produces the following:

$\rightarrow$ 0 0 1 1

$\rightarrow$ 1 1 0 0

The fitnesses of the offspring is now $\zeta(x) = 0.4$ (nowhere near optimal), see figure 6.3. Even though both the parents performed very well, their children performed poorly. The parents' failure to produce highly performing offspring lie in the fact that they are from different locations in the landscape, in this case it makes sense to allow only for parents to mate with individuals of their own liking.

6.3 Hybrid Algorithms

Genetic algorithms are good at tackling large and complex search spaces, but finding the true optimum proves to be challenging for it. Genetic algorithms have very good performance in the initial stages of the search, but the performance decreases later in the search as localised search begins. It makes the genetic algorithm the perfect means to start a search and to navigate through the space to locate the near optimal solutions. The genetic algorithm would mostly find the optimum given that it had enough time. However, time is usually not enough and is a very big factor in optimisation methods. Therefore it is worthwhile to consider some form of collaboration between those methods that perform well at the start and process the bulk of the search space, and those methods that are perfect at the end to lead the search to the optimum in the final moments. These end methods are typically more traditional methods. In other words, use the genetic algorithm to find the hill and a more specialised traditional method to climb it, thereby forming a hybrid algorithm. The easiest way to construct such a hybrid algorithm is to make the solution from the genetic algorithm, which was obtained after certain criteria were met, the starting groundwork for the traditional method, in which case a real valued vector would be used. The traditional method is chosen based on its ability to solve the specific problem. It is also possible to continue the final stages of the search in binary code. For example, the search can climb the local hill by mutating each bit in the string separately and then reassessing its fitness. The mutation is only regarded in the search if the fitness has increased. Another way is through addition or subtracting of 1 from the binary string and then yet again only regard the operation if it has improved the fitness. This is done for all the unknown parameters. If it should prove to be beneficial, then such methods can be applied at any point in time during the run and not only at the end of the search. However, it would be a mistake to desert the genetic algorithm too soon in a complex and difficult search space, as it can result in a fallacious solution. Other techniques include the use of heuristics, in which case child strings inherit certain traits to speed up the search. Another way to speed up the search is by only using approximated fitness estimations initially (Coley, 1999).

6.4 Additional Advanced Operators

There are numerous other operators which can be applied to the algorithm, examples of such are:

- Advanced mutation
- Dominance and diploidy
- Abeyance
- Inversion

These are simply mentioned for completeness sake and will not be discussed here. For more information refer to Coley (1999) and Goldberg (1989).

Part III

Implementation

Chapter 7

Modelling

In general, a structure is a system of nodes and elements, where nodes are connected by elements. A plane truss is a system of elements or members that are pin connected, where the whole structure lies within a single plane. There are no moments at joints due to pin connections which results in one degree of freedom in the axial direction. The applied forces must be in-plane forces for such a plane structure. Distributed beam loads may by represented by statically equivalent loads at the appropriate nodes for analysing purposes. This type of analysis, which is only subjected to nodal loads, will only produce axial member forces in tension or compression. On the other hand, a space truss can have members in any direction in space, not just members in one plane. This type of truss tolerates forces from any direction, however, the type of element remains the same (Coello *et al.*, 1994).

The model may be subjected to loading and constraints, once it has been properly defined. The type of model governs the type of forces it can carry.

This study commences with the use of truss elements. This results in a truss type structure, with one degree of freedom in the axial direction of the element, refer to figure 7.1. Loading may only be applied at the nodes. Fixity may only be specified in terms of translational restraints, as all nodes are pin connected. The truss structural element needs only a specified cross sectional area. The length of

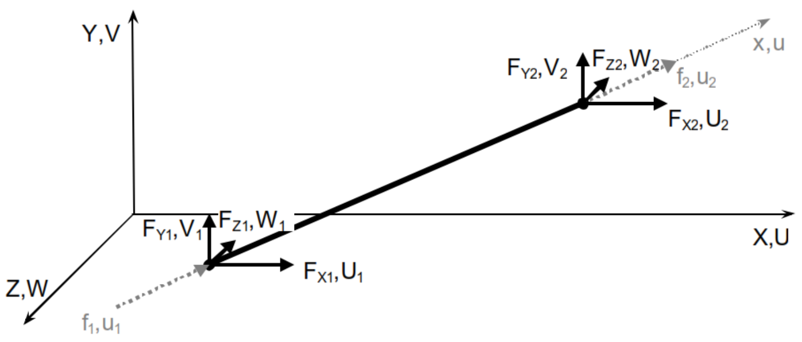


Figure 7.1: The truss element in space (Auer, 2005)

the element is determined by its nodal coordinates. Figure 7.1 illustrates a truss element within a three-dimensional space, with local and global reference systems and degrees of freedom. Bold and capital annotation represents the global coordinate system and faint and lowercase annotation represents the local system. UVW refers to global displacement directions, XYZ are the global coordinate directions. The numbers 1 and 2 refer to the two end nodes of an element.

7.1 Genetic Parameters

The benchmarking problems were performed with genetic parameters as specified in table 14.1, unless otherwise indicated. No sensitivity studies on these parameters are presented here as it is not directly aligned with the main aim of this thesis. Multiple runs were performed to establish which parameters result in the best outcome.

Table 7.1: Genetic parameters

Parameter	**Value**
Crossover probability	0.85
Mutation probability	0.005
Population size	50
Maximum number of generations	5000
Scaling constant	1.5
Number of crossover points	1
Elitism	TRUE
Selection with replacement	TRUE

7.2 Mapping the Structure to an Individual

Each individual in a population offers a solution to the problem, where an individual consists of a collection of chromosomes which describes it (refer to section 4.2.1). Hence, the algorithm would offer a 100 solutions for a population of a 100 individuals. These solutions typically converge to the same value late in the search when a near optimum has been found. A specific chromosome in an individual refers to a specific element in the structure. This chromosome contains information such as the element's cross sectional area and orientation in space, to name but a few. The binary string length of a chromosome is dependent on the size of the section list from which the algorithm can select discrete member sizes and other information. The binary string must be of such a length as to ensure that it can decode in a manner which allows the search to access every entry in the section list. Figure 7.2 illustrates the relationship between chromosomes and truss members. This figure also illustrates the concept of grouping (explained in section 7.3), where (for example) all red members are represented by the same chromosome. In such a way the whole structure is translated from an engineering model

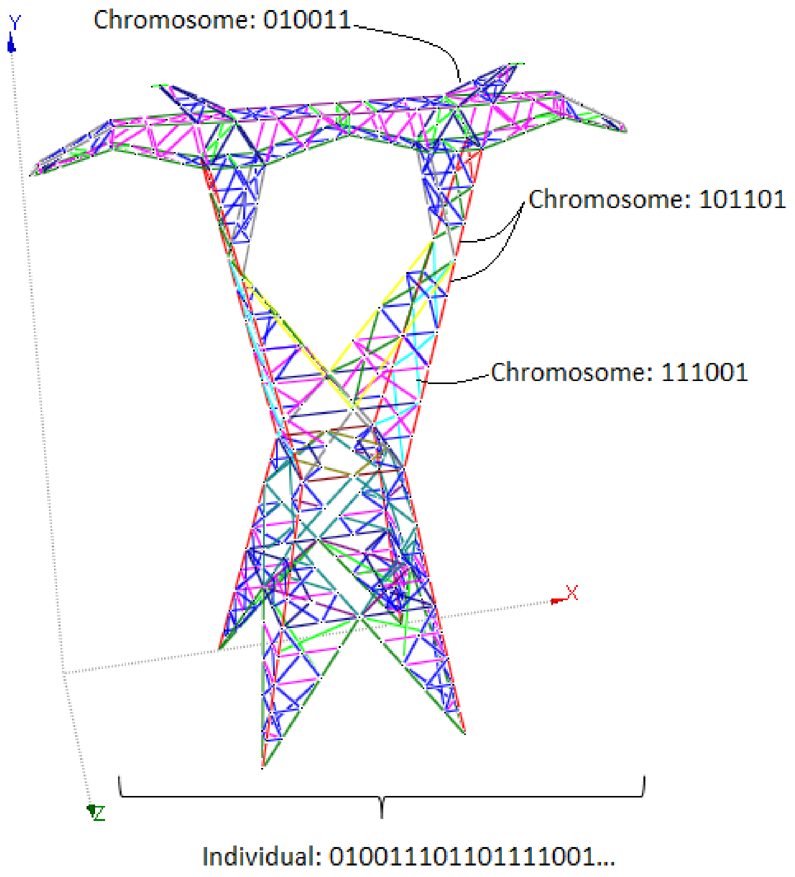


Figure 7.2: Converting the engineering model into a genetic model

into a binary genetic model consisting of individuals, chromosomes and genes. This conversion allows for the algorithm to apply all its genetic operators discussed in section 4.4.

7.3 Grouping

Each member in a structure can be directly mapped to a new chromosome, however this might not always be a practical, or even the best, approach. For certain cases it might be vital to retain symmetry within the structure due to reasons such as practicality and simpler construction methods. In such a case, certain members should be exactly the same in order to produce a symmetric structure. Another motivation for symmetry is to accommodate reversed loading; for instance, wind might blow from the opposite direction than originally described by the model loading and hence the structure needs to be designed for this reversed loading case as well. A means of achieving symmetry is through grouping. Grouping reduces the number of design variables for a given problem, therewith reducing the search space size and computational time required to execute the algorithm. Grouping might, for some cases, be the only way to solve a problem, even if symmetry is not required. The reason for this drawback is limited computation time. Consider a section list of 40 sections; this requires a minimum string length of 6. This will result in an individual string length of 3000 for a structure of 500 elements or 500 design variables. Also take into account that the genetic algorithm does not operate on an individual, but on a population of individuals. It would be beneficial to decrease the number of design variables through means of grouping. This study did not investigate the upper bound of the number of design variables that a genetic algorithm can handle effectively due to computation time restrictions.

Member grouping can either be decided a-priori, or performed by the algorithm through a grouping strategy. In this study grouping is defined by the user. It could be argued that the solution might be suboptimal due to the predefined grouping order. The predefined grouping order might result in a case where, given that the optimal structure was known, members of that optimal structure with different cross sections are placed in the same group. In other words, the user grouped members into the same group that should not be in the same group. In this way the optimal structure cannot be found by the algorithm.

A grouping strategy should be as such that the final solution contains the smallest number of cross sections with as much as possible search space reduction (Togan and Daloglu, 2008). A simple strategy suggested in Togan and Daloglu (2008) initially involves assigning the same cross sectional areas to all the members in the structure. An analysis is performed on this structure, where after the internal forces are divided into groups based on the magnitude of these forces obtained from the outcome of the analysis. An initial round of grouping is performed by grouping elements with forces of similar magnitudes. Tension and compression members fall into different groups. Members with zero force or a very small force are placed in a separate group. This method could be refined by grouping tension and compression members by different criteria. Tension members are still grouped by their internal axial forces, however compression members are grouped based on their slenderness ratio. The genetic algorithm is therefore only aware of the number of chromosomes (the number of groups), where the finite element analysis is aware of all the members in the structure. The more criteria exerted on a grouping strategy, the better the outcome will be. This is due to the fact that a group can only perform as well as its weakest member. The lightest structure will be produced for the case of no grouping, given that symmetry is not required and that there is enough computation time to accomplish such a solution.

According to (Togan and Daloglu, 2008), grouping has the following advantages:

- Search space reduction
- Increased probability of finding the true optimum
  - This advantage is mainly based on the fact that there is very likely not enough computation time to solve for every element in a large realistic structure
- Enhanced algorithm performance

– Due to shorter string lengths

7.4 Comments on CPU Time

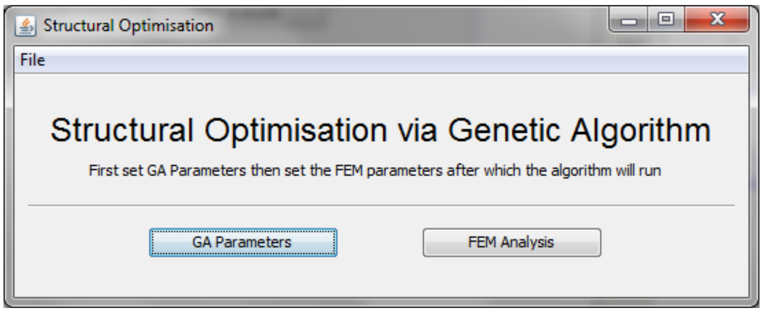
The analysis of the space truss, when compared to a planar truss, requires more CPU time as there are more global directions, even though the space truss could potentially have less design variables (which results in a smaller search space). This is due to the additional unknown forces that could potentially act at a node. The algorithm requires an analysis whenever a design variable or member of the structure was modified, in order to calculate the fitness and performance of the structure based on the outcome of the analysis. The algorithm can establish feasibility once the analysis is done. An analysis has to be performed for each individual in the population for every generation, before and after modifications to the element. Therefore, for a population of a 100 and 5 000 generations, the program would perform a million finite element analyses. Keep in mind that a population of 100 is still relatively small, greater populations might be needed for cases where greater exploration and diversity are required.

7.5 Deflection Criteria

The algorithm makes use of Table D.1 - Maximum deflections at serviceability - SANS (2005) for the case where deflections are not prescribed. The structure, for this case, is assumed to be an industrial type building, where its span is open to the interpretation and engineering judgement of the user. The structure is penalised as a whole, instead of penalising individual nodal displacements for each element, as is done for stress violations. This is done by assigning the maximum nodal displacement in the structure as the whole structure's displacement. The maximum allowable deflection that a structure may undergo is assumed to be 1/180 of the 'span' length (if no deflection limit is prescribed).

7.6 User Input Required to Run the Program

At start up, the program asks two inputs, the genetic parameters and the actual model, refer to figure 7.3. The genetic parameters are simply a list of parameters which the algorithm will need, refer to figure 8.5. The model, however, has a few steps which need to be completed. Users communicate the structure that they want to model through an Excel spreadsheet, hence the user needs to provide the file path to this document. The input must be in exactly the same format as shown in figure 7.4, this includes units (forces in Newton and nodal coordinates in meters). Columns J and K in figure 7.4 are element definitions. The element number associated with this definition is in column I. Grouping is defined from column L onwards. In figure 7.4 there are 7 groups; group number 2 contains elements number 2,3,4 and 5.

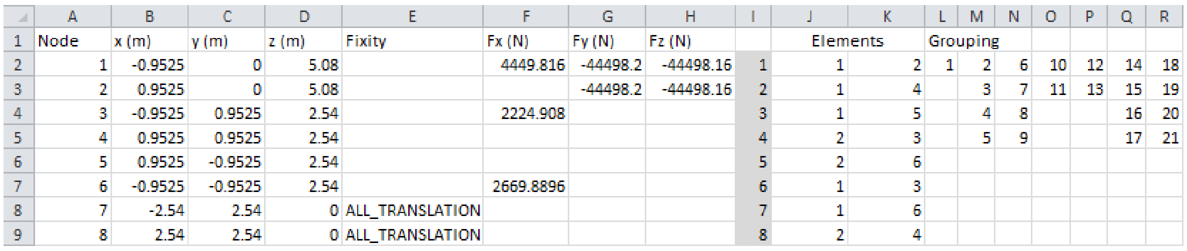


Structural Optimisation via Genetic Algorithm

First set GA Parameters then set the FEM parameters after which the algorithm will run

Figure 7.3: Program start-up

Next, the user must supply a section list which the algorithm can use, this is also done in an Excel spreadsheet. For this case however, the user has to indicate which columns in the sheet it must read. The navigation tab allows for a user to specify which columns are assigned to which section property. For example, a user can specify that the area column is column 5. This allows for the user to create any section list with any number of elements.

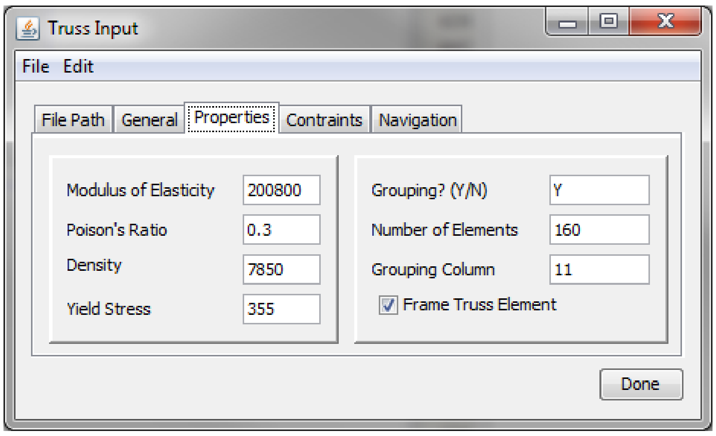


	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Node	x (m)	y (m)	z (m)	Fixity	Fx (N)	Fy (N)	Fz (N)		Elements		Grouping						
2	1	-0.9525	0	5.08		4449.816	-44498.2	-44498.16	1	1	2	1	2	6	10	12	14	18
3	2	0.9525	0	5.08			-44498.2	-44498.16	2	1	4		3	7	11	13	15	19
4	3	-0.9525	0.9525	2.54		2224.908			3	1	5		4	8			16	20
5	4	0.9525	0.9525	2.54					4	2	3		5	9			17	21
6	5	0.9525	-0.9525	2.54					5	2	6							
7	6	-0.9525	-0.9525	2.54		2669.8896			6	1	3							
8	7	-2.54	2.54	0	ALL_TRANSLATION				7	1	6							
9	8	2.54	2.54	0	ALL_TRANSLATION				8	2	4							

Figure 7.4: Excel spreadsheet user input for creating the model

The general tab contains fields such as the number of entries in the section database, the number of nodes and whether the structure is 2 or 3 dimensional. These input parameters are not only important for the finite element model, but also because they enable the program to read the spreadsheets accurately. These values, together with the number of design variables (obtained from the genetic parameters) and information on grouping, completely navigate the algorithm through the model input spreadsheet. Lastly, the user can define the structure's span, refer to section 7.5.

The properties tab contains input fields such as the modulus of elasticity, Poisson's ratio, density and the steel's yield stress, see figure 7.5. It also allows for the user to indicate whether the structure was grouped, in which case the actual number of elements for the structure must also be provided. This extra input value is necessary, as the number of chromosomes and the number of elements will no longer be the same value if grouping in implemented. Lastly, this tab allows for the user to indicate whether frame elements must be used instead of truss elements, refer to section 15.2.4.



Modulus of Elasticity	200800
Poison's Ratio	0.3
Density	7850
Yield Stress	355
Grouping? (Y/N)	Y
Number of Elements	160
Grouping Column	11
Frame Truss Element	✓

Figure 7.5: User input for the material properties, grouping information and type of element to be used

The constraints tab consists of two parts. The one part is for the case where stress and displacement constraints are simply assigned predefined values. This part is activated for the case of benchmarking problems. The other part creates the option of implementing the South African design code, SANS (2005).

Chapter 8

The Program

The program is an object oriented program which was written in Java and can be divded into two distinct parts; optimisation of the design problem and structural modelling and analysis. These two parts are completely separate, they are even coded in separate packages.

8.1 Approach to Implementing Structural Optimisation

The implementation process, as illustrated in figure 8.1, starts with designing the structure (the design parameters are generated), after which solutions from the optimisation process are analysed and evaluated. The structural variables are adjusted according to the outcome of the analyses. In this study, the design process (selection of design variables), optimisation and evaluation will be performed by the genetic algorithm and the analyses of the solutions generated will be performed by the finite element method program. The finite element method program discretises a structure and operates by solving systems of equations. Just as the finite element method moved structural analysis away from functions towards discrete values at nodes, so too, in contrast to earlier techniques, did evolutionary algorithms move optimisation away from searching for that optimum analytical function to rather searching for optimum values in a discretised search space. The finite element program needs to analyse the problem repeatedly throughout the optimisation procedure, it is therefore recommended to use a relatively crude finite element method model in order to be computationally effective. Once the programs have looped and are now at the second generation or beyond, the outcome of the finite element program will serve, together with the objective function, as a guide through the search space in the quest to find the optimal structure. This outcome is in the form of internal element forces and a structural displacement. The algorithm will then commence with the redesign. The two programs will, in such a way, work together toward a common goal; they will run concurrently until optimisation conditions are met, or a given number of loops were performed (generation counter).

The program makes allowance for the use of both classes 3 and 4 members (refer to section 12.7 and figure 12.3) for the implementation of the South African design code. Limiting the search to class

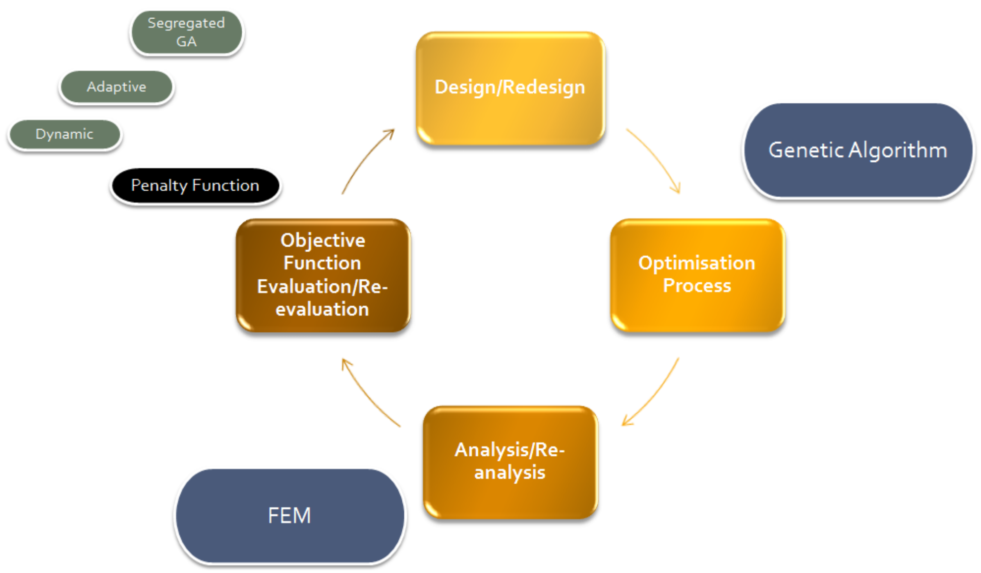


Figure 8.1: The design process

3 elements would only produce a near optimal result for the constrained search space of class 3 only sections, however it could result in a suboptimal solution for a realistic steel structure where no such limitations are necessary on the search space. The program was only coded for equal leg angle sections in order to avoid unnecessary complexities, such as shifted shear centres.

8.2 Characteristics of the Program

The number of sections that the user can consider in the search is not limited; the user can simply compile a master section list which contains all the desired sections. The same is applicable to loading. Unique test lists can also be compiled. Load cases are considered separately. The user must create a new combined load case, for the case where load cases need to be combined. The program terminates after the specified number of generations has been executed.

8.3 Pseudo Code

This section provides a step by step description of the algorithm. Figure 8.2 is a graphical illustration of the complete program.

```
Step 1:      Set the parameters
Step 2:      Generate initial population
Step 3:      Decode the chromosomes
                 Refer to section 4.5.1
```

```
Step 4:    Create finite element models
Step 5:    Analyse the finite element models
Step 6:    Obtain initial element forces and the largest displacement for
           each individual
Step 7:    Begin genetic algorithm:
               for $g$ = 1 to $number\ of\ generations$
Step 8:    Evaluate the objective
               $\phi_{p,i}$ for $(i = 1, 2, ..., n)$
               Refer to section 8.6
Step 9:    Find the fittest individual with regard to $\phi_p$
Step 10:   Find the weakest individual with regard to $\phi_p$
Step 11:   Evaluate fitness
               $\zeta_i$ for $(i = 1, 2, ..., n)$
               Refer to section 8.6
Step 12:   Obtain statistics
               $\zeta_{sum} = \sum_{i=1}^{n} \zeta_i$
               $\zeta_{ave} = \frac{\zeta_{sum}}{n}$
               for $i$ = 1 to $n$
                   if $(\zeta_i > \zeta_{\max})$
                       then $\zeta_{\max} = \zeta_i$
                   end
               end
Step 13:   Scale fitness
               $\zeta_{sum} = 0$
               if $(c_m \neq \zeta_{\max}$ and $\zeta_{\max} \neq \zeta_{ave})$
                   then $a = \frac{(c_m - 1)\zeta_{ave}}{\zeta_{\max} - \zeta_{ave}}$
                           $b = (1 - a)\,\zeta_{ave}$
                       for $i$ = 1 to $n$
                           $\zeta_i^s = a\zeta_i + b$
                           if $(\zeta_i^s < 0)$
                                   $\zeta_i^s = 0$
                           end
                           $\zeta_{sum} = \zeta_{sum} + \zeta_i^s$
                       end
                   $\zeta_{ave} = \frac{\zeta_{sum}}{n}$
               end
Step 14:   Select
               create random number $rouletteWheel$ = random number * $\zeta_{sum}$
                   $sum$ = 0
                   while ($sum < rouletteWheel$ and $i < n$)
```

```
                    $sum$ = $sum$ + $\zeta_i$
                    increment $i$
                end
            individual $i$ will be selected
Step 15:    Crossover
            if (random number $\leqslant p_c$)
                perform crossover, refer to section 8.6
            end
Step 16:    Mutate
            For each individual $I$:
                for $k$ = 1 to $L$
                    if (random number $\leqslant p_m$)
                        then if ($I_k$ = 0)
                                then $I_k$ = 1
                        end
                        else
                                $I_k$ = 0
                        end
                    end
                end

Step 17:    Create finite element models from new population
Step 18:    Analyse the finite element models
Step 19:    Obtain initial element forces and the largest displacement
            for each individual
Step 20:    Elitism
            if elitism is $true$
                if ($\zeta_{max,old} > \zeta_{max,new}$)
                    place the fittest individual of the old population
                    at a random position in the new population
                end
            end
Step 21:    Update the temporary population's attributes after modifications
            Different force and displacement values
Step 22:    Replace the old population with the new population
            $temporary\ population \rightarrow current\ population$
Step 23:    $g = g + 1$
```

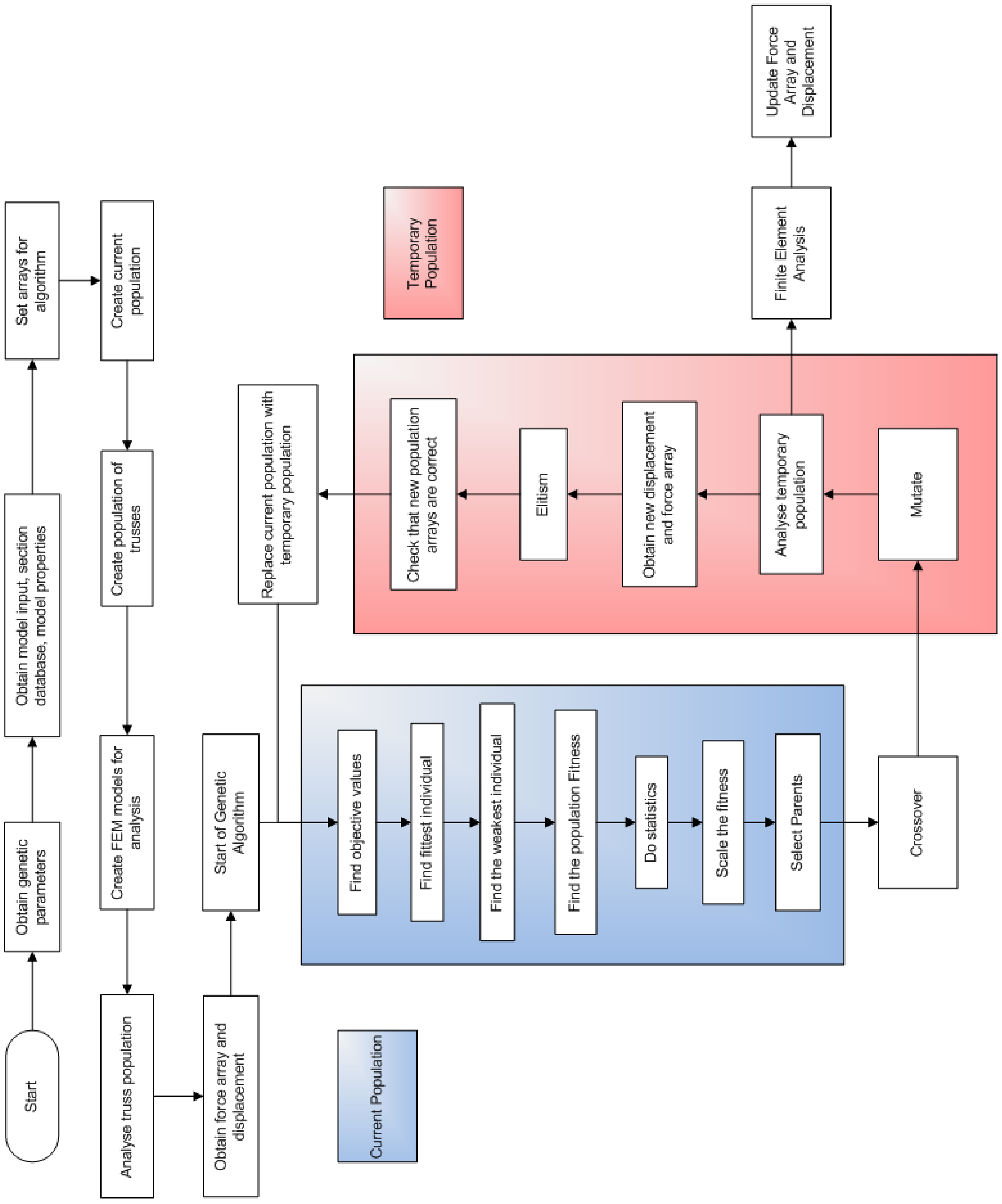

Start
Obtain genetic parameters
Obtain model input, section database, model properties
Set arrays for algorithm
Create current population
Create population of trusses
Create FEM models for analysis
Analyse truss population
Obtain force array and displacement
Start of Genetic Algorithm
Find objective values
Find fittest individual
Find the weakest individual
Find the population Fitness
Do statistics
Scale the fitness
Select Parents
Crossover
Mutate
Analyse temporary population
Finite Element Analysis
Update Force Array and Displacement
Obtain new displacement and force array
Elitism
Check that new population arrays are correct
Replace current population with temporary population
Current Population
Temporary Population

Figure 8.2: Flow diagram of the combined genetic algorithm and finite element method

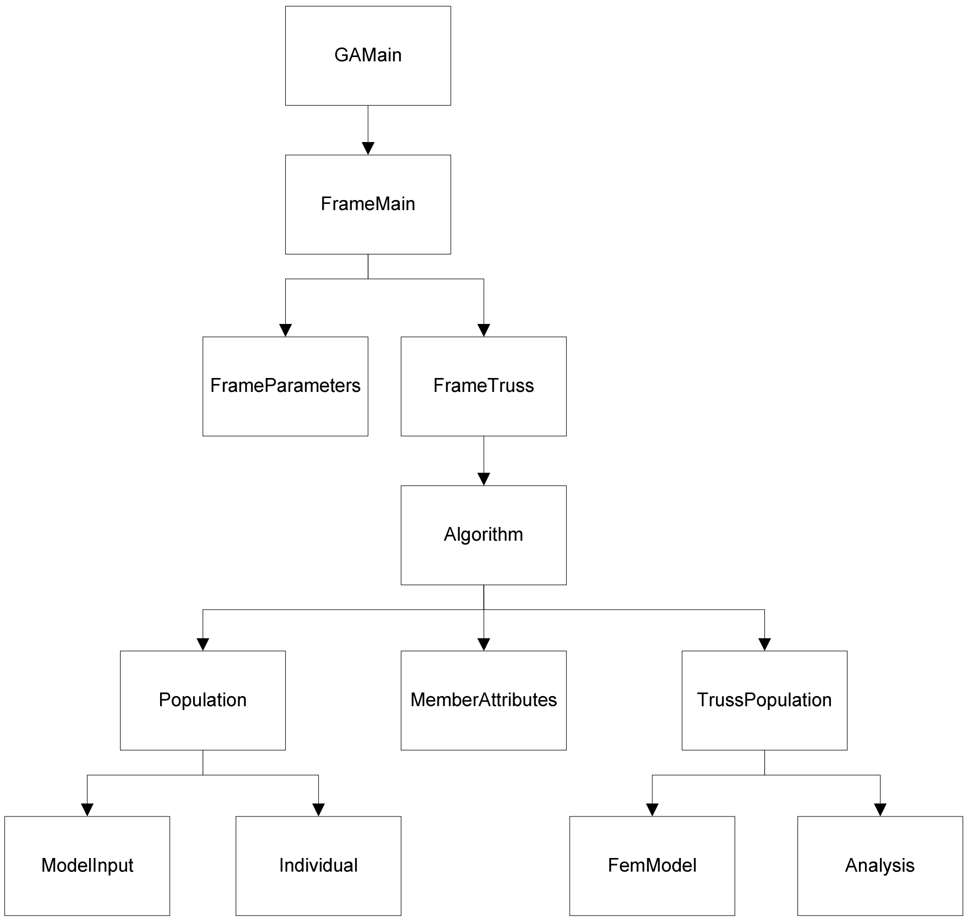


Figure 8.3: Basic structure of the program

8.4 Towards the Finite Element Method

An analysis can be linear or non-linear. Secondary (or P-Delta) effects are ignored for the case of a linear analysis. On the other hand, the whole structure is in equilibrium for its deformed state in the case of a nonlinear analysis, therefore the secondary effects are taken into account. Elastic buckling can result from secondary effects. A nonlinear analysis will be able to indicate whether buckling has occurred by either not converging or resulting in extreme post buckling displacements. A linear analysis will not be able to detect buckling. This implementation will make use of a linear finite element analysis, secondary effects are not explicitly taken into account by the analysis program, however buckling is taken into account when the fitness values are calculated for the case where the South African design code is implemented.

8.5 Discussion of Essential Classes

Figure 8.3 illustrates the relationship between all the different program classes.

8.5.1 Population and Individual

The algorithm operates with two populations, the *current population* and the *temporary population*, refer to figure 8.2. The current population's function is to carry information from the one generation to the next. All changes which are made to the current population are stored in the temporary population. The temporary population will replace the current population only after all the modifications to the individuals are complete. A population is an array of individuals and an individual is an array of chromosomes. Each chromosome is a new design variable for a given structure. However, the individual is simply an integer array which is initially populated at random, where after it is adapted by the algorithm. A binary encoding scheme was used for the individuals, therefore the integers used for populating the arrays were only 0 and 1. Refer to Appendix 17.5 for complete code extracts.

8.5.2 Truss Population

The truss population contains a population of finite element models, named *FemModels*. A *FemModel* is an object which has attributes such as material, load, support, node and element. These attributes help to model the actual structure. This class acts as an interface between the genetic algorithm and the finite element method program.

8.6 Notes on Functions

Only a few selected functions will be discussed and special features will be highlighted, such features may in some cases simply specify which approach the algorithm implemented.

```
public void setArrays(FrameParameters gaParam, ... )
```

This method is activated before the algorithm is started. It serves to read all the excel input files' data into arrays. Arrays are created instead of real-time reading from file because real-time reading takes an excessive amount of time. These arrays contain model information and section properties and will remain unchanged throughout the run. This method creates:

- Arrays to be used in the objective function
  - The radius of gyration array
  - St. Venant's torsion constant of cross section array
  - The thickness array
  - The cross sectional area array
  - The lengths of all members array
  - The distance to shear centre array

  - The moment of inertia array
- Arrays to store output from the finite element analysis
  - The force (double) array
  - The displacement array
- Arrays to communicate with finite element analysis
  - The *femIndexArray*
  - The *femElementArray*

The force matrix, which will only be populated after the finite element analysis is run, is initialised here. The force double array is a matrix, as each individual i will have a force for each chromosome/element e, $\mathbf{F} = F_{i,e}$ for $i = 1, .., number\,of\,individuals$ and $e = 1, ..., number\,of\,elements$. The displacement array is also initiated here; however it is simply a vector as each individual only has one overall displacement value. Only the largest nodal displacement in the structure will be used in the penalty function, where the structure will be penalised as a whole, refer to section 7.5. In other words, each chromosome is assigned a force and each individual is assigned a deflection.

The *femIndexArray* is used only for grouping, refer to section 7.3. It acts as a mapping device from the individual (which will only know the number of design variables/chromosomes) to an array which the analysis will use (which will be the size of the actual number of elements in the structure). The genetic algorithm is only ‘aware’ of the chromosomes, the finite element method program is ‘aware’ of the whole structure. The *femElementArray* is simply a means for the algorithm to determine which element is part of which group.

```
public double findObjectiveFunctionValue(Individual individual, ...)
```

This method does not simply determine the weight of the structure (as the objective is to minimise the weight), but also enforces a penalty on individuals with constraint violations. The function is divided into two sections, the first section calculates the penalised objective function value (ϕ_p) based on the design code. The second section is a set of simpler checks for the case of prescribed constraints in order to execute benchmarking problems. The first section is then further subdivided into two sections, one which performs calculations for circular hollow sections and the other for equal angle sections. These calculations are only performed for the number of design variables and not the number of elements in the structure, hence saving computation time.

8.6.1 Objective Function Value Calculation: Equal Leg Angle

8.6.1.1 Classification: Equal Leg Angle Sections

The SANS 10162-1 code classifies a section as either class 3 or 4, depending on its width to thickness ratio:

$$\frac{b}{t} \leqslant \frac{200}{\sqrt{f_y}} \tag{8.6.1}$$

for the case where this condition holds, the section can be classified as class 3.

8.6.1.2 Check for Slenderness

First the algorithm establishes whether a member is in tension or compression, where after the member is checked for slenderness according to SANS 10162-1. The maximum slenderness ratio for members in compression shall not exceed 200 (SANS, 2005).

$$\frac{KL}{R} \leqslant 200 \tag{8.6.2}$$

The maximum slenderness ratio for members in tension shall not exceed 300 (SANS, 2005).

$$\frac{KL}{R} \leqslant 300 \tag{8.6.3}$$

The member is immediately penalised if these conditions are not met. The penalty parameter is a variable declared at the start of the function. The penalty is increased for cases of constraint violations and will be updated as the function continues through all the checks. For pinned connections the effective length is simply taken to be the length of the element. In this case the penalty for individual i for a slenderness violation of element e in generation t is as follows:

$$\begin{gathered} \frac{KL}{R} = 300 \\ \therefore g_{1,e}(t) = \frac{L_e}{300R_e} - 1 \end{gathered} \tag{8.6.4}$$

A thorough background to penalty functions is provided in Section 4.6. The function can also check for redundant members, however these elements are simply defined for elements which carry no force. Such elements might be needed to avoid mechanisms and it should only be classified as redundant if it be redundant for all the relevant load cases. For the case of compression, the penalty for individual i simply changes to:

$$\therefore g_{2,e}(t) = \frac{L_e}{200R_e} - 1 \tag{8.6.5}$$

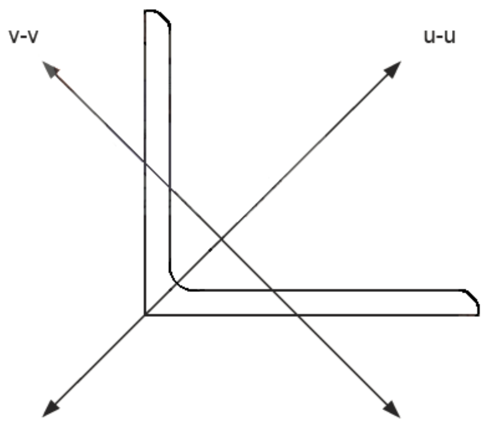


Figure 8.4: Equal leg angle section

8.6.1.3 Determine Capacity of the member

The allowable force for each member is calculated according to SANS 10162-1 for both tension and compression in order to establish whether a particular element has a constraint violation.

Tension

The tensile resistance of a member was taken as:

$$T_r = \phi_{st} A_g f_y \text{ with } \phi_{st} = 0.9 \tag{8.6.6}$$

The tensile resistance of connections are not taken into account and therefore also not their respective net effective areas.

Compression

The equal leg angle section is singly symmetric (see figure 8.4), therefore for torsional or torsional-flexural buckling f_e was taken as the lesser of f_{ex} and f_{eyz}.

Torsional or Torsional-Flexural Buckling:

$$f_{ey} = \frac{\pi^2 E}{\left(\frac{L_u}{R_u}\right)^2} \tag{8.6.7}$$

$$\overline{R}_0^2 = u_0^2 + v_0^2 + R_u^2 + R_v^2 \tag{8.6.8}$$

$$f_{ez} = \frac{GJ}{A\overline{R}_0^2} \tag{8.6.9}$$

$$\Omega = 1 - \frac{u_0^2 + v_0^2}{\overline{R}_0^2} \tag{8.6.10}$$

$$f_{eyz} = \frac{f_{ey} + f_{ez}}{2\Omega}\left(1 - \sqrt{1 - \frac{4 f_{ey} f_{ez} \Omega}{\left(f_{ey} + f_{ez}\right)^2}}\right) \tag{8.6.11}$$

$$f_{ex} = \frac{\pi^2 E}{\left(\frac{L_v}{R_v}\right)^2} \tag{8.6.12}$$

Flexural Buckling:

$$\lambda = \sqrt{\frac{f_y}{f_e}} \tag{8.6.13}$$

f_e is taken to be the lesser of f_{ex} and f_{eyz}.

$$C_r = \phi_{st} A_g f_y \left(1 + \lambda^{2n}\right)^{\frac{-1}{n}} \tag{8.6.14}$$

with $n = 1.34$ and $\phi_{st} = 0.9$

According to table 3 in SANS (2005) an element is of class 4 if condition 8.6.15 does not hold and might therefore require an area reduction.

$$\frac{b}{t} \leqslant \frac{200}{\sqrt{f_y}} \tag{8.6.15}$$

For the case where $W \leqslant W_{lim}$ no area reduction is necessary.

$$W = \frac{b}{t} \tag{8.6.16}$$

$$W_{\text{lim}} = 0.644\sqrt{\frac{kE}{f}} \quad \text{with} \;\; k = 0.43 \tag{8.6.17}$$

f is a reduced calculated stress, taking into account slenderness and buckling ($\leqslant f_y$). f is taken as $\frac{C_r}{\phi A_g}$ with $\phi = 0.9$. For this case the effective area of the section remains the gross area, $A_{eff} = A_g$. However, for the case where $W > W_{\text{lim}}$ a special area reduction on the element is necessary and a new compressive capacity is calculated from the new effective area.

$$b_{new} = 0.95t\sqrt{\frac{kE}{f}}\left(1 - \frac{0.208}{W}\sqrt{\frac{kE}{f}}\right) \tag{8.6.18}$$

The new effective area:

$$A_{eff} = A_g - (b - b_{new})t \tag{8.6.19}$$

The new capacity of the element:

$$C_r = \phi A_{eff} f \tag{8.6.20}$$

The penalty is once again activated if the force from the analysis is greater than the allowable force (T_r or C_r), depending on whether the element is in tension or compression. For this case the penalty for individual i is as follows:

$$g_{3,e}(t) = \left(\frac{F_e}{F_{allowable}} - 1 \right) \tag{8.6.21}$$

F_e is the actual force in member e and $F_{allowable}$ is either T_r or C_r depending on the analyses output. The displacement constraint is typically prescribed or is a calculated assumption based on SANS 10162-1. If the user does not specify a deflection limit, then the algorithm would assume a deflection limit of span divided by 180 based on Annex D, see section 7.5. This is however an assumption where the span has to be interpreted, i.e. as the height of a tower or the span of a dome or even some multiple or variation thereof. Penalty for individual i is activated for the case where the largest deflection in the structure, as determined by the analysis, is greater than the deflection limit.

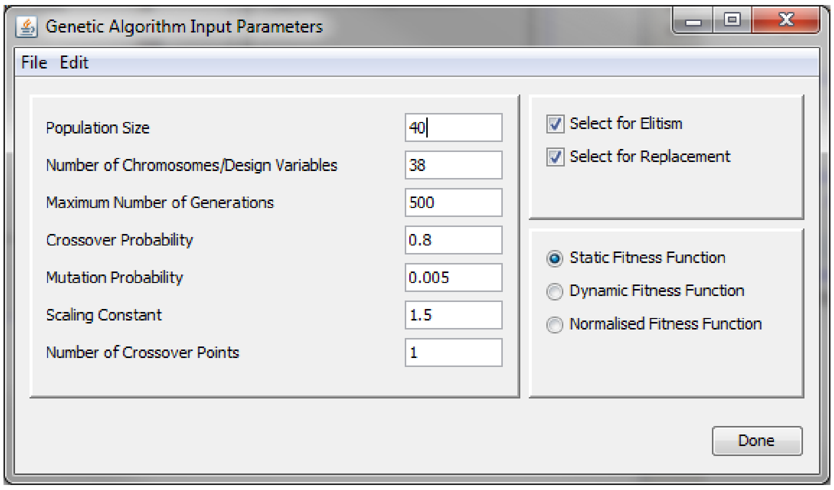
$$g_4(t) = \left(\frac{D_{greatest}}{D_{allowable}} - 1 \right) \tag{8.6.22}$$

For the second part of the function, where constraints are only prescribed, the penalties are calculated in the same way, however without calculating the allowable force, displacement and slenderness limits. The allowable force and displacement are simply taken as prescribed values. The overall penalty approach adopted here is the additive approach, refer to equation 4.6.1, where the penalty terms are simply added to the objective function in order to created the augmented penalised objective function. A higher objective function value will result in a lower fitness function value. For the case where no constraints are violated, $\psi(\mathbf{x}) = 0$. This implementation used an exterior penalty method, refer to section 4.6.2. The overall violation for individual i is measured by ψ_i:

$$\psi_i(t) = \left[\sum_{e=1}^{M} r_e G_e \right] \quad \text{with } \mathrm{r}_e = 1 \tag{8.6.23}$$

$$G_e = \max \left[0, \sum_{s=1}^{4} g_{s,e}(t) \right]^{\beta} \quad \text{with } \beta = 2 \tag{8.6.24}$$

A simple penalty parameter (250) was multiplied to the constraint functions after all the constraints have been checked, instead of multiplying each violation with a small amount r_e. It is easier to check what the effect of the penalty parameter is on the performance of the search by applying the term in this way. After all the penalty calculations have been performed for each element in an individual (i), the function calculates the mass of the structure. The objective function is simply the weight of the structure, $\phi_i = d \cdot \text{mass}_i$, where d is the density. The penalised objective function value ($\phi_{i,p}$) is the



Population Size: 40
Number of Chromosomes/Design Variables: 38
Maximum Number of Generations: 500
Crossover Probability: 0.8
Mutation Probability: 0.005
Scaling Constant: 1.5
Number of Crossover Points: 1

✓ Select for Elitism
✓ Select for Replacement

● Static Fitness Function
○ Dynamic Fitness Function
○ Normalised Fitness Function

Figure 8.5: Genetic parameter and fitness selection options

product of density and mass of the individual, plus the penalty function value. The exterior penalty was slightly modified into a dynamic penalty function by incorporating the generation count (t), see section 4.6.5.

$$\phi_{i,p}(t) = \phi_i + t \cdot 250 \cdot \psi_i \tag{8.6.25}$$

The generation count simply refers to the number of generations already executed, this implies that the severity of a penalty violation increases as the run progresses. The penalty parameter is simply a constant number which amplifies the penalty term. For this study, this parameter was chosen to be 250 as it resulted in the best performance for the algorithm. It was discovered that this number in combination with the maximum number of generations greatly affects the search. The greater the maximum number of generations, the smaller the penalty parameter needs to be.

```
public double findFitnessFunctionValue(Individual individual, ...)
```

Calculating only the objective function would not suffice as a genetic algorithm is a maximisation algorithm, see section 4.5. The fitness function value needs to be calculated in order to convert the problem from a minimisation problem to a maximisation problem. The algorithm allows for three different approaches to fitness, see figure 8.5. Approaches to fitness are discussed in more detail in the subsection 4.5.2.

8.6.2 Static Fitness

The fitness is simply calculated by subtracting the penalised objective function from a very large constant value, refer to section 4.5.2.1.

$$\zeta_i = 1000000000 - \phi_{i,p} \tag{8.6.26}$$

8.6.3 Dynamic Fitness

This approach ensures that the individual with the highest objective function value (lowest fitness) will be assigned a proportional value to that of the lowest objective function value, refer to section 4.5.2.2.

$$\zeta_i = \phi_{\min} + \phi_{\max} - \phi_{i,p} \tag{8.6.27}$$

8.6.4 Normalised Fitness

The normalised fitness approach scales fitness values to fractional sizes, refer to section 4.5.2.2.

$$\zeta_i = \frac{1}{\phi_{i,p} \cdot (1000 \cdot v + 1)} \tag{8.6.28}$$

```
public void statistics()
```

This function simply needs to calculate the sum fitness, the average fitness and the maximum fitness of the population. These values are important for functions such as elitism, refer to section 8.3.

```
public void scaleFitnessFEM(double[] largestDispl,...)
```

The algorithm makes use of linear fitness scaling as discussed in section 4.5.3.

```
public Individual select()
```

The algorithm implements the standard Roulette Wheel Selection, refer to section 4.4.1.

```
public void crossover()
```

This method is not limited to one point crossover, the number of crossover points are defined by the user. It contains a built in check, as a specific crossover location can only be used once in cases where the number of crossover points are more than one. One point crossover produces a whole new structure, which behaves completely different (structurally) compared to the two parent models. It seems that the difference between 1 and 2 point crossover is rather insignificant, due to the rather similiar 'magnitude in difference' of the offspring. Performance decreases once the number of crossover points reaches 3, one could argue that this is the point where the search becomes too random. However, there is an argument that states one point crossover should technically produce the best results, refer to section 5.3.4. The function uses the select function until the population is completely populated with the method of replacement.

```
        ...
        if (Math.random() <= CROSSOVER_PROBABILITY){
        ...
        //Perform crossover
            for(int k = 0; k < CROSSOVER_POINTS + 1; k++){
                begin = crossoverPoints[k];
                end = crossoverPoints[k+1];
                if(k>0)
                    end = crossoverPoints[k+1] - 1;  //check for overlap
                if(k==CROSSOVER_POINTS)              //check for last point
                    end = crossoverPoints[k+1];
                if(counter % 2 == 0){
                    for(int m = begin; m < end; m++){
                        child1.individual[m] = parent1.individual[m];
                        child2.individual[m] = parent2.individual[m];
                    }
                }
                else{
                    for(int m = begin; m < end; m++){
                        child1.individual[m] = parent2.individual[m];
                        child2.individual[m] = parent1.individual[m];
                    }
                }
                counter++;
            }
            counter = 0;
            //Place children in temporary population
            ...
       }
       else{
            //No crossover
            for (int q = 0; q < TOTAL_STRING_LENGTH; q++ ){
                child1.individual[q] = parent1.individual[q];
                child2.individual[q] = parent2.individual[q];
            } ..
       }
       ...
```

Listing 8.1: Crossover

```
public void elitism(double[] largestDispl,...)
```

Elitism for this implementation only allows for one elite individual to pass through to the next generation.

```
public FemModel[] createFemModels(int[] femIndexArray,...)
```

A *femModel* is created for each individual in the population. Therefore, a population of 50 individuals generate 50 *femModels* initially and then 50 new models after crossover and elitism were performed. Refer to Appendix 17.5, section 17.5 for *femModel* code definition. After the nodes, loading, supports and material has been added to the model, the function needs to scale the size of the area array used by the genetic algorithm to the number of elements present in the structure to a size usable by the analysis. At first the array is populated as governed by genetic algorithm, then a new array is created (*femAreas*) using the *femElementArray* and *femIndexArray* as described earlier. For ten elements, the *femElementArray* could typically look as illustrated by figure 8.6.

1	2	3	7	8	9	4	5	6	10

Figure 8.6: A *femElementArray*

It is important to ensure that the correct elements correspond to correct attributes and properties, therefore it remains crucial that the correct order of elements is maintained. For the ten elements, the *femIndexArray* could typically look as illustrated by figure 8.7. This is simply how the algorithm counts the number of elements in a group from the way the user defined it in the input spreadsheet. As the function loops through the array, the value obtained refers to the chromosome area that must be inserted in the analysis area array, see listing 8.2. This is not necessary for the case where grouping is false. The whole process of creating *femModels* is repeated, but in this case for the number of elements and not chromosomes.

0	0	0	0	1	1	2	2	2	2	3	3	4	4	4	4	5

Figure 8.7: A *femIndexArray*

```
areas = new double[gaIndividual.NUMBER_OF_CHROMOSOMES];
  areas = gaIndividual.getAreaIndividual_meters(gaIndividual,
  entries);

        if (isGrouped){
            //Duplicate group elements
            femAreas = new double[numberOfElements];
            for (int i = 0; i < femAreas.length; i++){
                femAreas[femElementArray[i] - 1] = areas[femIndexArray[i]];
            }
        }
```

Listing 8.2: Mapping of area arrays

8.7 Special Notes

It is important to clear the sets created by the *FemModels*, as the program will run out of memory if this is not done. The user must ensure that all the values are provided in the correct units, for the steel sections database all values must be in millimetres or some power thereof. For the model input all units must be in Newton or meter.

The structure should be stable for the algorithm to run:

- Joints and members should be defined as such that loading can be carried effectively through the elements to the supports.
- Supports should be defined as such so that the structure does not become mobile or rotate.

Part IV

Benchmarking Problems

Chapter 9

Introduction

The benchmarking part of this thesis is dedicated towards solving standard problems in literature with the genetic algorithm created in this study. These benchmarking problems have been solved many times before; therefore an algorithm can be benchmarked by comparing its outcome to that of the other studies. The algorithm is acceptable if its performance is comparable to literature to a satisfying degree. The parameters used in these benchmarking problems were chosen based on a mixture of what was used in literature and from running the problem multiple times for different parameters to see which resulted in the best outcome. Each benchmarking problem will commence with an explanation of the objective for that specific benchmarking problem, as different problems in this study serve to illustrate and validate different aspects of the algorithm. The next section will provide the relevant design data, this data is important as it highlights the exact architecture of the problem. The outcome of a problem can only be compared when the design data of the two models at hand is exactly identical. This also applies to the constraints enforced on a problem. Additional information which does not form part of the design data is provided in cases where necessary. Finally, each bench-marking problem will conclude with the results obtained by this study's algorithm and a comparison to other literature.

The order in which these problems are implemented follows a gradual progression from a simple two dimensional problem with fewer variables and prescribed stress and displacement constraints, to a more complex three dimensional problem with more design variables which implements the South African code of design (SANS, 2005).

Chapter 10

10 Bar Truss

10.1 Objective

The 10 Bar Truss depicted in figure 10.1 is a non-convex problem, because it has multiple local minima (Falakian and Mousavi, 2011). See figure 10.2 for illustrative difference between convex and non-convex functions. The objective of this problem is to optimise the cross sectional areas of each element in the truss in order to minimise the weight of the structure. Running this benchmarking problem will establish whether the algorithm works for plane trusses, by comparing the outcome to studies such as Galante (1996) and Nanakorn and Meesomklin (2001). Moreover, this benchmarking problem serves to provide insight into the algorithm's performance.

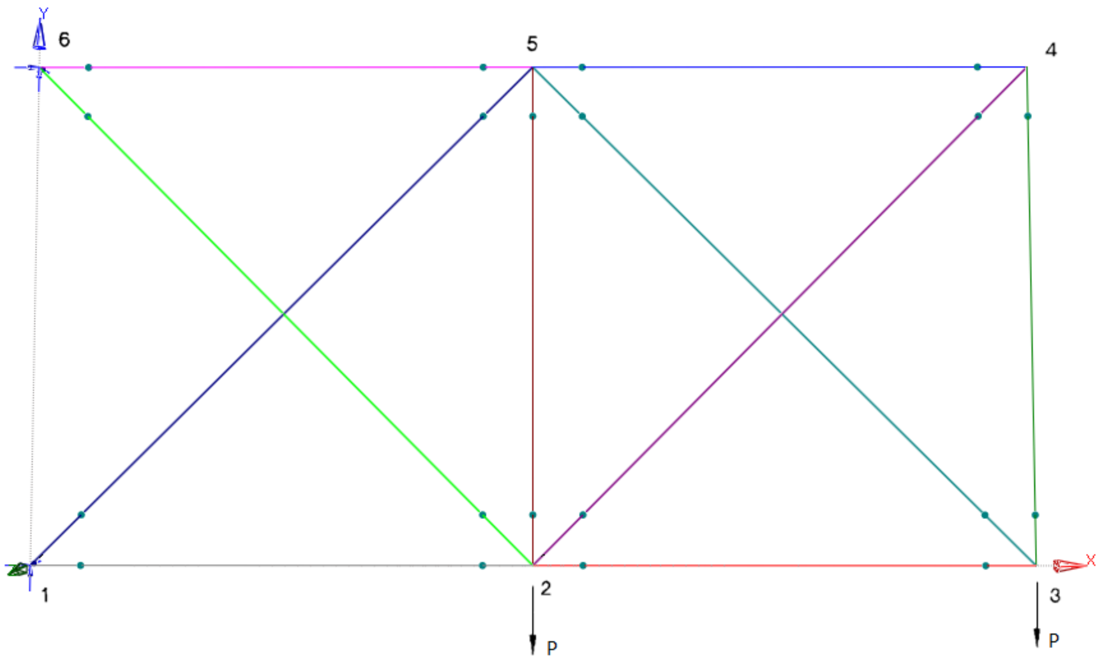


Figure 10.1: 10 Bar Truss

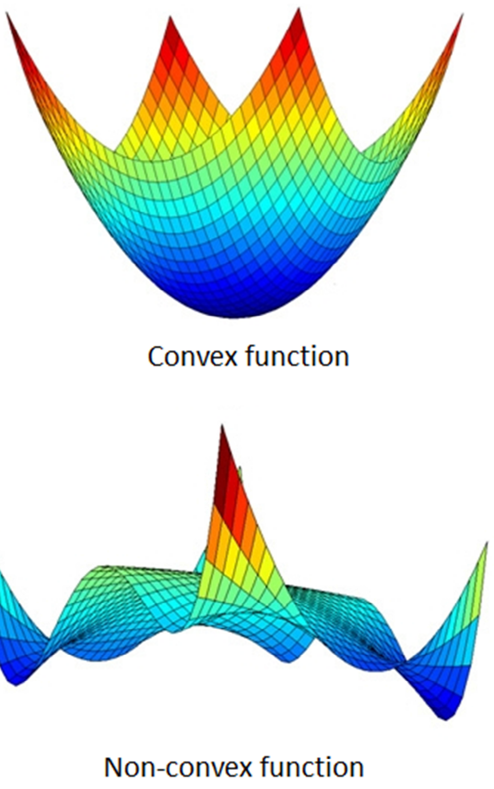


Figure 10.2: A convex function versus a non-convex function

10.2 Design Data

The weight of the truss is optimised by selecting different combinations of cross sectional areas (from a section list provided) for the design variables, refer to table 10.6 and table 10.4. The material properties used in this problem is that of aluminium, refer to table 10.5. The only reason for this specific set of material properties is to create the exact same model as the one used in the literature studies. Table 10.6 is the standard section list used for this problem. It is important to use the same section list, as a different section list will result in a different answer. This is illustrated later in the text.

Table 10.1: 10 Bar Truss Nodal Coordinates

Node	x **(m)**	y **(m)**	z **(m)**
1	0	0	0
2	9.144	0	0
3	18.288	0	0
4	18.288	9.144	0
5	9.144	9.144	0
6	0	9.144	0

Table 10.2: 10 Bar Truss Loading

Node	F_x **(N)**	F_y **(N)**	F_z **(N)**
2	0	-444822	0
3	0	-444822	0

Table 10.3: 10 Bar Truss Fixity

Node	**Fixity**
1	XY_TRANSLATION
6	XY_TRANSLATION

Table 10.4: 10 Bar Truss Element Definition

Design Variable Number	**End Nodes of Members**
1	(6,5)
2	(5,4)
3	(1,2)
4	(2,3)
5	(2,5)
6	(3,4)
7	(2,6)
8	(1,5)
9	(2,4)
10	(5,3)

Table 10.5: 10 Bar Truss Material Properties

Property	**Value**	**Unit**
Modulus of Elasticity	68947.59	MPa
Density	2767.99	kg/m^3

Table 10.6: 10 Bar Truss Section List

Area Section List mm^2						
	1045.159	1161.288	1283.868	1374.191	1535.481	1690.319
	1696.771	1858.061	1890.319	1993.544	2019.351	2180.641
	2238.705	2290.318	2341.931	2477.414	2496.769	2503.221
	2696.769	2722.575	2896.768	2961.284	3096.768	3206.445
	3303.219	3703.218	4658.055	5141.925	7419.34	8709.66
	8967.724	9161.272	9999.98	10322.56	10903.2	12129.01
	12838.68	14193.52	14774.16	17096.74	19354.8	21612.86

10.3 Constraints

The design constraints below are standard to the 10 bar benchmarking problem, refer to Coello *et al.* (1994) or Rajeev and Krishnamoorthy (1992).

- Displacement constraint: $D_{\max} \leqslant 50.8\,\text{mm}$
- Stress constraint: $-172.25\,\text{MPa} \leqslant \sigma_{allow_i} \leqslant 172.25\,\text{MPa}$ with $i = 1, ..., 10$

10.4 Additional Information

Coello *et al.* (1994) used a mutation rate of 0.01 which implies that 1 in every 100 bits can potentially be mutated. This is a very high mutation rate; it was found that the search became too random and produced poor results when this rate was applied in this study. The best result was obtained with a mutation rate of 0.005, where 1 in every 200 bits has a probability to be mutated.

10.5 Results

The 10 bar benchmarking problem was run for all the fitness approaches mentioned in sections 8.6.2, 8.6.3 and 8.6.4. The outcome of the static fitness approach is plotted in figure 10.3 in order to illustrate the performance and inner workings of the algorithm. The mass of the truss decreases as the deflection increases to its limit, this suggests that the limiting constraint for this benchmarking problem is its deflection. The minimum mass obtained for the 10 bar truss was 2494.46kg with a normalised fitness approach (this does not prove the noramalised fitness approach to be superior).

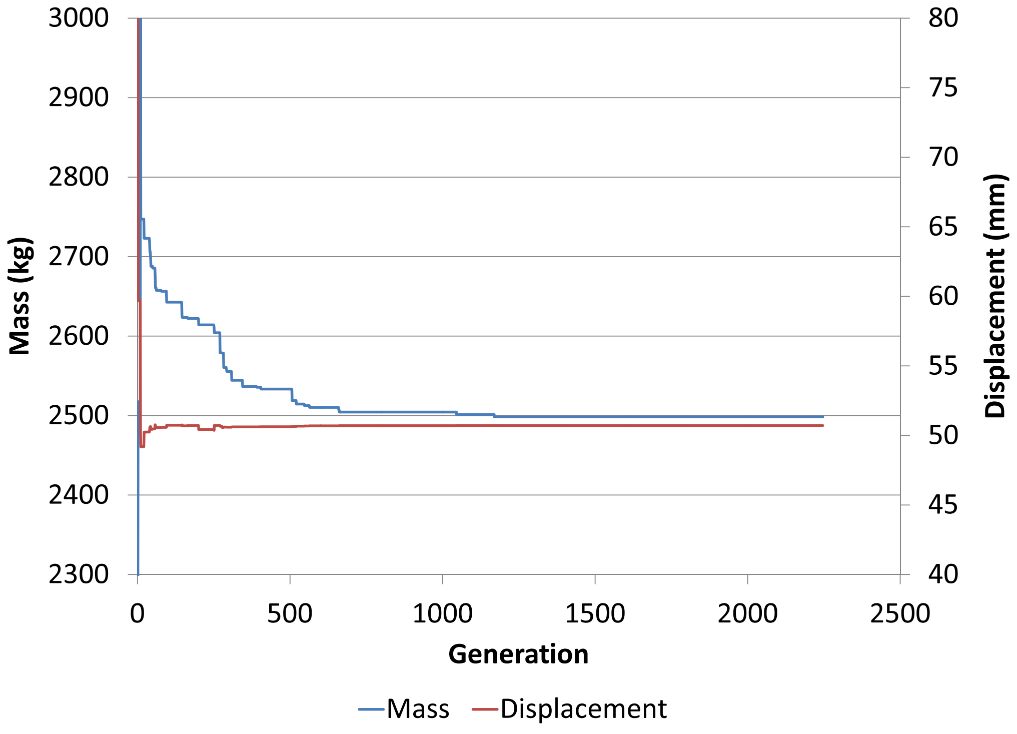


Figure 10.3: 10 bar truss performance for static fitness

Table 10.7: 10 Bar Truss Area Distribution

Design Variable	Area
A1	21612.86
A2	1045.159
A3	14774.16
A4	9999.98
A5	1045.159
A6	1045.159
A7	4658.055
A8	14774.16
A9	14774.16
A10	1045.159

10.6 Comparison

For ease of reference, the following names were used in the table 10.8:

- *Galante* for Galante (1996)
- *Nanakorn* for Nanakorn and Meesomklin (2001)
- *Appelo* for this study
- *Coello* for Coello *et al.* (1994)
- *Sivakumar* for Sivakumar *et al.* (2004)
- *Rajeev* for Rajeev and Krishnamoorthy (1992)

The variables A1 to A 10 in table 10.8 correspond to the design variables given in table 10.7.

Table 10.8: Minimum mass comparison for the 10 bar benchmarking problem

Study	**Mass**	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**A10**
Galante	2475.88	21613	1045	14194	9161	1045	1045	5142	14774	14194	1045
Appelo	2494.46	21613	1045	14774	10000	1045	1045	4658	14774	14774	1045
Nanakorn	2494.48	21613	1045	14774	10000	1045	1045	4658	14774	14194	1045
Coello	2534.08	19355	1045	14774	8710	1045	1045	8968	14194	14194	1045
Sivakumar	2540.06	19355	1045	12839	10903	1045	1045	5142	17097	14774	1284
Rajeev	2546.44	21613	1045	14194	10000	1045	1045	9161	12839	12839	1690

The mass in table 10.8 is given in $[kg]$ and the areas in $[mm^2]$; the areas are represented by design variables A1,...,A10. The performance of the genetic algorithm is highly dependent on selecting the correct parameters, implementing specialised genetic operators and different strategies regarding grouping, fitness and reduced search spaces, to name but a few. For example, consider the 2 genetic parameter sets defined in table 10.9.

Table 10.9: Genetic parameter sets to illustrate algorithm dependence

Genetic Paramater Set	Population size	Crossover rate	Mutation rate
1	50	0.5	0.05
2	150	0.85	0.005

Figure 10.4 illustrates the difference in performance of parameter set 1 and 2. It could be argued that parameter set 1 did not have a large enough population for the algorithm to work with and that its crossover rate was too low. The search is not allowed enough exploration with a crossover rate that is too low; therefore it can be observed that parameter set 1 seems to easily fall onto a plateau,

whereafter it takes a few generations before it finds a fitter solution. Conversely, parameter set 2 shows a gradual decline in mass with far less 'plateau action'. This part is not meant to be viewed as a sensitivity analysis, it merely serves to illustrate the algorithm's dependence on selecting the correct parameters for good performance.

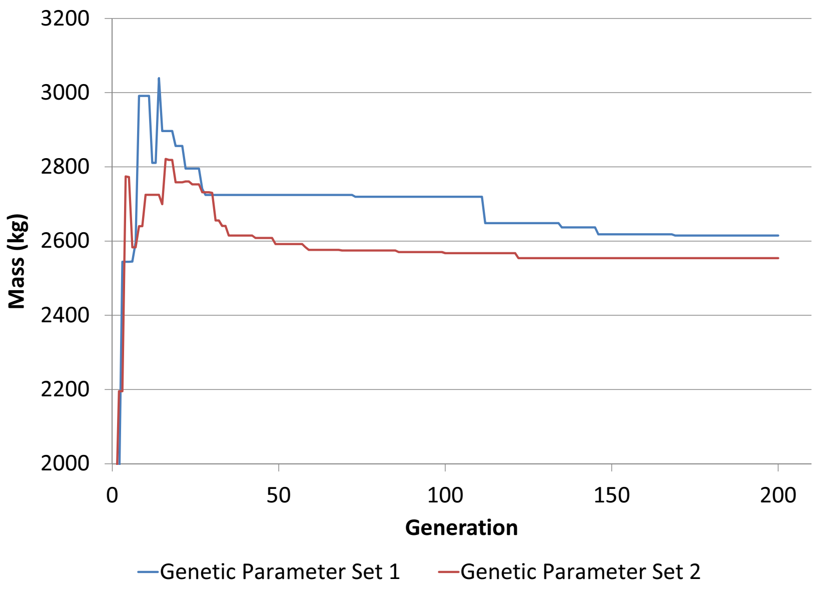


Figure 10.4: Mass comparison for parameter sets 1 and 2

Chapter 11

25 Bar Truss

11.1 Objective

The objective of this benchmarking problem is to optimise a space truss and to make use of grouping. Grouping will allow for the structure to remain symmetrical. The number of elements in this truss is 25, however the number of design variables is only 8. Different colours group different elements together, see figure 11.1.

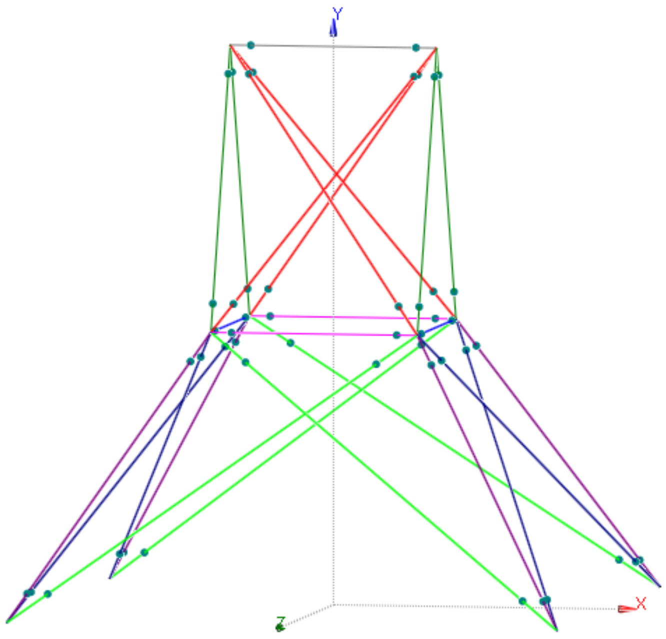


Figure 11.1: 25 Bar Truss

11.2 Design Data

The weight of the truss is optimised by selecting different combinations of cross sectional areas for the design variables, refer to table 11.6 and table 11.4. The material properties used in this problem, as is with the 10 bar truss problem, is that of aluminium, refer to table 11.5.

Table 11.1: 25 Bar Truss Nodal Coordinates

Node	x (m)	y (m)	z (m)
1	-0.9525	0	5.08
2	0.9525	0	5.08
3	-0.9525	0.9525	2.54
4	0.9525	0.9525	2.54
5	0.9525	-0.9525	2.54
6	-0.9525	-0.9525	2.54
7	-2.54	2.54	0
8	2.54	2.54	0
9	2.54	-2.54	0
10	-2.54	-2.54	0

Table 11.2: 25 Bar Truss Loading

Node	F_x **(N)**	F_y **(N)**	F_z **(N)**
1	4449.816	-44498.2	-44498.2
2	0	-44498.2	-44498.2
3	2224.908	0	0
6	2669.89	0	0

Table 11.3: 25 Bar Truss Fixity

Node	**Fixity**
7	ALL_TRANSLATION
8	ALL_TRANSLATION
9	ALL_TRANSLATION
10	ALL_TRANSLATION

Table 11.4: 25 Bar Truss Element Definition and Grouping

Design Variable	**End Nodes of Members**
A1	(1,2)
A2	(1,4),(1,5),(2,3),(2,6)
A3	(1,3),(1,6),(2,4),(2,5)
A4	(3,6),(4,5)
A5	(3,4),(5,6)
A6	(3,10),(4,9),(5,8),(6,7)
A7	(3,8),(4,7),(5,10),(6,9)
A8	(3,7),(4,8),(5,9),(6,10)

Table 11.5: 25 Bar Truss Material Properties

Property	**Value**	**Unit**
Modulus of Elasticity	68947.59	MPa
Density	2767.99	kg/m^3

Table 11.6: 25 Bar Truss Section List

Area List mm^2					
64.516	129.032	193.548	258.064	322.58	387.096
451.612	516.128	580.644	645.16	709.676	774.192
838.708	903.224	967.74	1032.256	1096.772	1161.288
1225.804	1290.32	1354.836	1419.352	1483.868	1548.384
1612.9	1677.416	1806.448	1935.48	2064.512	2193.544

11.3 Constraints

The design constraints below are standard to the 25 bar benchmarking problem, refer to Coello *et al.* (1994).

- Displacement constraint: $D_{\max} \leqslant 8.89\,\text{mm}$
- Stress constraint: $-275.79\,\text{MPa} \leqslant \sigma_{allow_i} \leqslant 275.79\,\text{MPa}$ with $i = 1, ..., 8$

11.4 Additional Information

Groenwold *et al.* (1999)'s information was not included in the comparison since the authors made use of a different area sections list. Erbatur *et al.* (2000) implemented a multilevel optimisation procedure where the search space is reduced for each successive level. This approach starts off with an initial level in the optimisation, where after the solutions from this level are used as the initial population for the next level. 'Sub-profile' lists are compiled for the next level by dividing the initial discrete list into subsets, after which the subsets are enlarged. This results in a smaller search space. This method is only mentioned for completeness sake and will not be further discussed. Only the first level mass, before the search space reduction, is used for comparison below.

11.5 Results

The minimum mass obtained for the 25 bar truss was 222.483kg.

Table 11.7: 25 Bar Truss Area Distribution

Design Variable	**Area**
A1	129.032
A2	258.064
A3	2064.512
A4	129.032
A5	516.128
A6	774.192
A7	580.644
A8	2193.544

The GA found a feasible solution (a solution with no constraint violations), for the case of static fitness, within 5 generations, refer to figure 11.2.

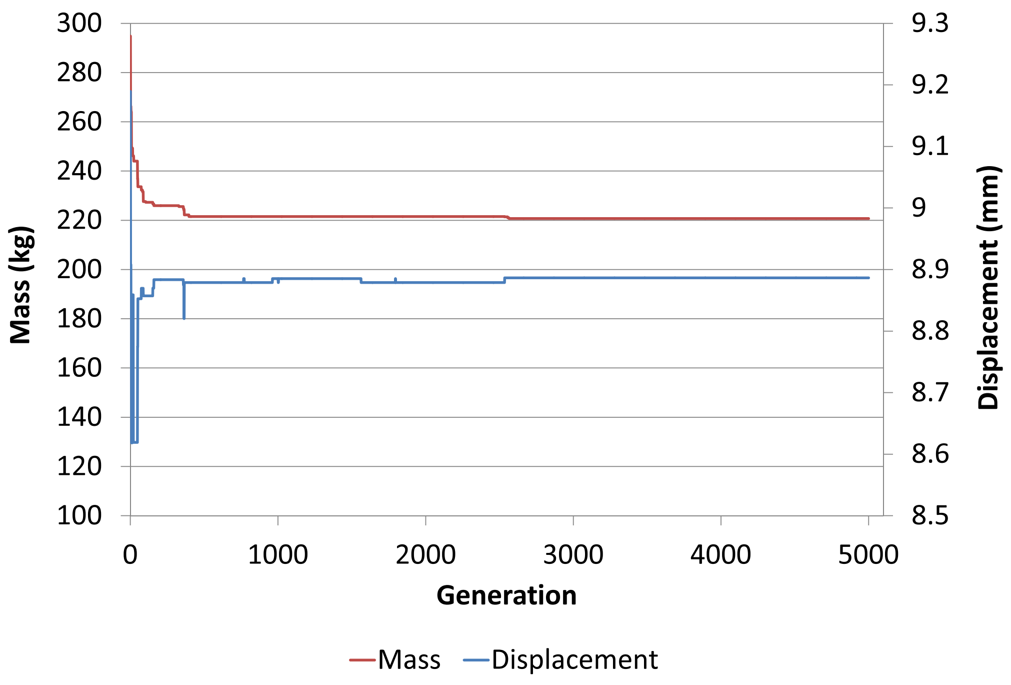


Figure 11.2: 25 Bar truss: Static fitness performance

11.6 Comparison

For ease of reference, the following names were used in table 11.8:

- *Togan* for Togan and Daloglu (2008)
- *Appelo* for this study
- *Coello* for Coello *et al.* (1994)
- *Erbatur* for Erbatur *et al.* (2000)
- *FCD Method* for Flager *et al.* (2011)
- *Rajeev* for Rajeev and Krishnamoorthy (1992)
- *Groenwold* for Groenwold *et al.* (1999)

The 25 bar benchmarking problem shows a significant increase in variation between section sizes obtained by different studies. The design variables A1 to A8 in table 11.8 correspond to the design variables given in table 11.4

Table 11.8: 25 bar benchmarking problem comparison to literature

Study	**Mass (kg)**	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**
Togan	219.25	65	194	2194	65	1290	645	323	2194
Appelo	222.43	65	65	2194	65	1419	774	323	2194
Coello	224.05	65	452	2065	65	903	710	323	2194
Erbatur	233.60	65	645	2194	129	387	710	581	1935
FCD Method	238.96	65	65	2194	65	65	516	1613	1613
Rajeev	247.67	65	1161	1484	129	65	516	1161	1935
Groenwold	248.09	6	1290	2065	6	6	452	1032	1677

Chapter 12

160 Bar Truss

12.1 Objective

The structure in this benchmarking problem is a 3 dimensional 160 bar tower, refer to figure 12.1. The objective of this benchmarking problem is to illustrate the power of the algorithm. In contrast to

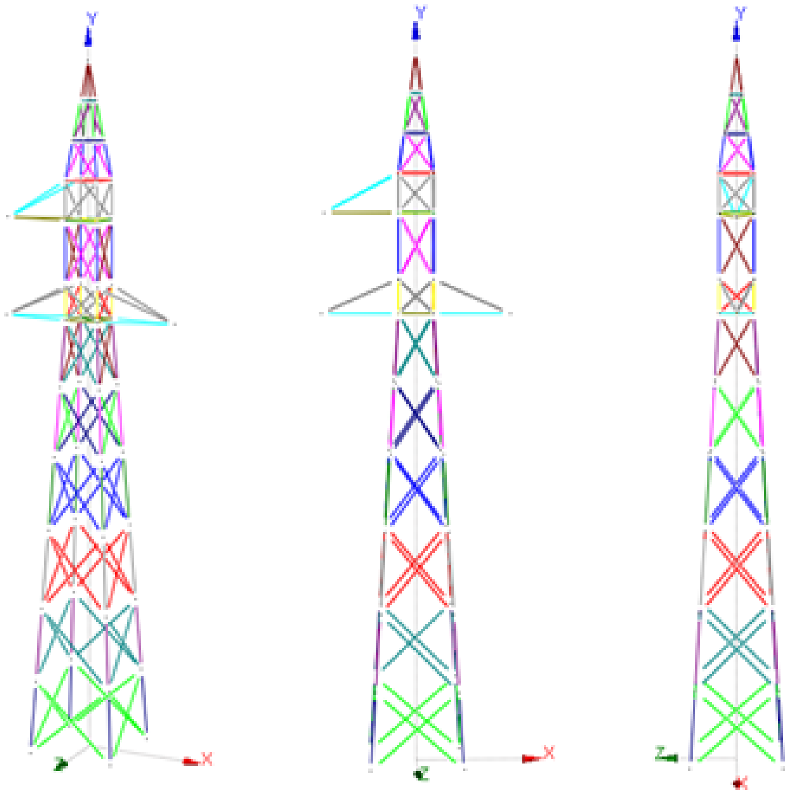


Figure 12.1: 160 Bar Truss Model

previous problems, this problem has a significant increase in the number of design variables (38 design variables) and it sets about steering the study towards solving real life problems.

Krishnamoorthy *et al.* (2002) went as far as creating a 1 792 - and a 2 304 planar space truss and solving for it, refer to figure 12.2. However, for the 1 792 planar space truss only a quarter of the truss was modeled, thereby drastically reducing the number of elements to be processed by the finite element method program. Additionally, the authors only used 24 design variables with 5 sections to choose from. This results in a string encoding length that is relatively short (72 bits). Consequently, the very large scale problem was reduced to a rather small scale problem. The same was done for the 2 304 planar space truss, which only had 10 design variables, with small string lengths of 30. It was therefore decided to model 160 bar benchmarking problem with its 38 design variables.

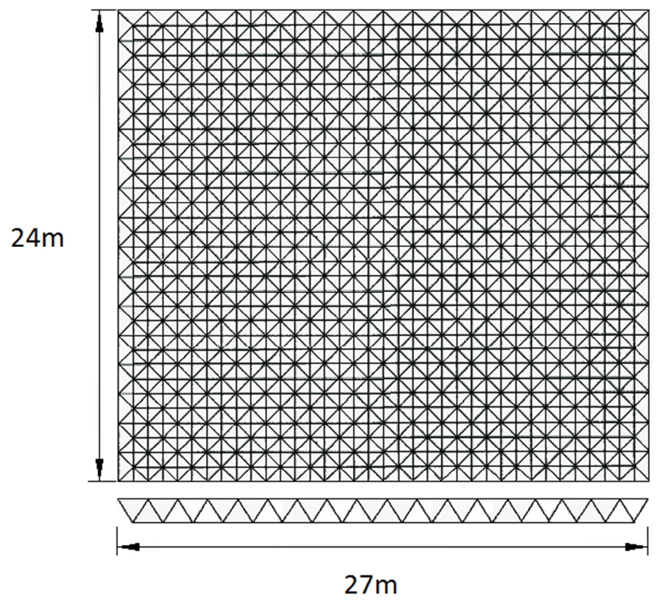


Figure 12.2: 2 304 Planar space truss (Krishnamoorthy *et al.*, 2002)

This problem will be implemented with the SANS (2005) design code and makes use of a section list provided in the Southern African Steel Construction Handbook (SAISC, 2008), see Appendix 17.5. After results were obtained and compared to literature, the truss will be analysed again using frame elements. The concept and motivation behind such elements are thoroughly discussed in chapter 15. The objective here is only to establish whether such elements will produce results similar to that of truss elements. In other words, to validate using such an approach if it be needed.

12.2 Notes on the 160 Bar Truss

Problems found in literature vary greatly, from the way in which grouping is implemented to the forces applied to the structure and constraints taken into account. Rajeev and Krishnamoorthy (1992) and Galante (1996) ran this benchmarking problem, however Rajeev and Krishnamoorthy (1992) had 12 design variables and Galante (1996) had 16 design variables. Groenwold *et al.* (1999) took buckling and slenderness into account, where Rajeev and Krishnamoorthy (1992) did not. The loading applied by Rajeev and Krishnamoorthy (1992) and Groenwold *et al.* (1999) are completely different. The mass of this structure reported by various studies differs significantly, from 666.487kg (Rajeev and Krishnamoorthy, 1992) to 1359.781kg (Groenwold *et al.*, 1999). The studies do not indicate whether own weight was included. It is not always clear what section lists were used by respective authors. It was decided to only benchmark this study with that of Groenwold *et al.* (1999), due to the above described inconsistencies of this benchmarking problem. There are two significant differences between this study and that of Groenwold *et al.* (1999):

- Groenwold *et al.* (1999) used an American design code, whereas this study implemented the South African design code (SANS, 2005)
- Groenwold *et al.* (1999) used American section sizes, whereas this study used section sizes from the Southern African Steel Construction Handbook(SAISC, 2008)
- This study used South African steel (SJ355R)

This benchmarking problem is the preparatory phase to the case study and serve to fulfill the objective of implementing the South African design code in the algorithm. The algorithm itself, as well as the implementation of the design code, needs to be validated before the study can commence with the case study. This is why this study did not implement this problem with the American design code.

12.3 Comments on Comparing Results

Different finite element method programs will produce the same results with great precision, when the same type of analysis is performed. It should be noted that the outcome of the optimisation procedure is therefore not dependent on the analysis program itself or only the optimisation technique itself, but also on the design standards which are implemented (for example SANS 10162-1 or ASCE code). Solving the same problem with different design standards will produce different results. The outcome of an optimised problem, when compared to another, is not necessarily a reflection of the genetic algorithm's performance, but perhaps a reflection on the level of conservatism of a given design code.

12.4 Design Data

The material used in this model is SJ355R steel, refer to table 12.3. This material was chosen as the algorithm implemented the South African design code. Table 12.5 defines the elements in the tower, the bold numbers refer to the element number and the nodes column to the two nodes that create an element. Table 12.6 shows which elements are grouped together. For example, elements 1, 2, 3 and 4 are all in group 1, therefore they are all represented by design variable 1.

Table 12.1: 160 Bar Truss Loading

Node	**Fx (N)**	**Fy (N)**	**Fz (N)**
25	-8272	-4368	0
28	-7514	-4132	2562
37	-6940	-4132	2562
52	-6444	-3001	2705

Table 12.2: 160 Bar Truss Fixity

Node	**Fixity**
1	ALL_TRANSLATION
2	ALL_TRANSLATION
3	ALL_TRANSLATION
4	ALL_TRANSLATION

Table 12.3: 160 Bar Truss Material Properties

Property	**Value**	**Unit**
Modulus of Elasticity	210000	MPa
Density	7850	kg/m^3

Table 12.4: 160 Bar Truss Nodal Coordinates

Node	**x (m)**	**y (m)**	**z (m)**	**Node**	**x (m)**	**y (m)**	**z (m)**
1	-1.05	0	-1.05	**27**	0.4	10.275	-0.4
2	1.05	0	-1.05	**28**	2.14	10.275	0
3	1.05	0	1.05	**29**	0.4	10.275	0.4
4	-1.05	0	1.05	**30**	-0.4	10.275	0.4
5	-0.93929	1.75	-0.93929	**31**	-0.4	11.055	-0.4
6	0.93929	1.75	-0.93929	**32**	0.4	11.055	-0.4
7	0.93929	1.75	0.93929	**33**	0.4	11.055	0.4
8	-0.93929	1.75	0.93929	**34**	-0.4	11.055	0.4
9	-0.82859	3.5	-0.82859	**35**	-0.4	12.565	-0.4
10	0.82859	3.5	-0.82859	**36**	0.4	12.565	-0.4
11	0.82859	3.5	0.82859	**37**	-2.07	12.565	0
12	-0.82859	3.5	0.82859	**38**	0.4	12.565	0.4
13	-0.71156	5.35	-0.71156	**39**	-0.4	12.565	0.4
14	0.71156	5.35	-0.71156	**40**	-0.4	13.465	-0.4
15	0.71156	5.35	0.71156	**41**	0.4	13.465	-0.4
16	-0.71156	5.35	0.71156	**42**	0.4	13.465	0.4
17	-0.60085	7.1	-0.60085	**43**	-0.4	13.465	0.4
18	0.60085	7.1	-0.60085	**44**	-0.26592	14.365	-0.26592
19	0.60085	7.1	0.60085	**45**	0.26592	14.365	-0.26592
20	-0.60085	7.1	0.60085	**46**	0.26592	14.365	0.26592
21	-0.49805	8.72	-0.49805	**47**	-0.26592	14.365	0.26592
22	0.49805	8.72	-0.49805	**48**	-0.12737	15.265	-0.12737
23	0.49805	8.72	0.49805	**49**	0.12737	15.265	-0.12737
24	-0.49805	8.72	0.49805	**50**	0.12737	15.265	0.12737
25	-2.14	10.275	0	**51**	-0.12737	15.265	0.12737
26	-0.4	10.275	-0.4	**52**	0	16.15	0

Table 12.5: 160 Bar Truss Element Definition

	Nodes			Nodes			Nodes			Nodes	
1	1	5	**41**	13	18	**81**	25	31	**121**	36	40
2	2	6	**42**	14	17	**82**	28	32	**122**	38	41
3	3	7	**43**	14	19	**83**	28	33	**123**	39	42
4	4	8	**44**	15	18	**84**	25	34	**124**	35	43
5	1	6	**45**	15	20	**85**	26	31	**125**	40	41
6	2	5	**46**	16	19	**86**	27	32	**126**	41	42
7	2	7	**47**	16	17	**87**	29	33	**127**	42	43
8	3	6	**48**	13	20	**88**	30	34	**128**	43	40
9	3	8	**49**	17	21	**89**	26	32	**129**	35	36
10	4	7	**50**	18	22	**90**	27	31	**130**	36	38
11	4	5	**51**	19	23	**91**	29	34	**131**	38	39
12	1	8	**52**	20	24	**92**	30	33	**132**	39	35
13	5	9	**53**	17	22	**93**	27	33	**133**	40	44
14	6	10	**54**	18	21	**94**	29	32	**134**	41	45
15	7	11	**55**	18	23	**95**	30	31	**135**	42	46
16	8	12	**56**	19	22	**96**	26	34	**136**	43	47
17	5	10	**57**	19	14	**97**	26	29	**137**	40	45
18	6	9	**58**	20	23	**98**	27	30	**138**	41	46
19	6	11	**59**	20	21	**99**	31	35	**139**	42	47
20	7	10	**60**	17	24	**100**	32	36	**140**	43	44
21	7	12	**61**	21	26	**101**	33	38	**141**	44	45
22	8	11	**62**	22	27	**102**	34	39	**142**	45	46
23	8	9	**63**	23	29	**103**	33	39	**143**	46	47
24	5	12	**64**	24	30	**104**	32	35	**144**	44	47
25	9	13	**65**	21	27	**105**	31	36	**145**	44	48
26	10	14	**66**	22	26	**106**	34	38	**146**	45	49
27	11	15	**67**	23	30	**107**	32	38	**147**	46	50
28	12	16	**68**	24	29	**108**	33	36	**148**	47	51
29	9	14	**69**	22	29	**109**	34	35	**149**	45	48
30	10	13	**70**	23	27	**110**	31	39	**150**	46	49
31	10	15	**71**	24	26	**111**	37	35	**151**	47	50
32	11	14	**72**	21	30	**112**	37	39	**152**	44	51
33	11	16	**73**	26	27	**113**	37	40	**153**	48	49
34	12	15	**74**	27	29	**114**	37	43	**154**	49	50
35	12	13	**75**	29	30	**115**	35	40	**155**	50	51
36	9	16	**76**	30	26	**116**	36	41	**156**	48	51
37	13	17	**77**	25	26	**117**	38	42	**157**	48	52
38	14	18	**78**	27	28	**118**	39	43	**158**	49	52
39	15	19	**79**	25	30	**119**	35	38	**159**	50	52
40	16	20	**80**	29	28	**120**	36	39	**160**	51	52

Table 12.6: 160 Bar Truss Grouping

Design Variable	**Elements**							
1	1	2	3	4				
2	5	6	7	8	9	10	11	12
3	13	14	15	16				
4	17	18	19	20	21	22	23	24
5	25	26	27	28				
6	29	30	31	32	33	34	35	36
7	37	38	39	40				
8	41	42	43	44	45	46	47	48
9	49	50	51	52				
10	53	54	57	58				
11	55	56	59	60				
12	61	62	63	64				
13	65	66	67	68				
14	69	70	71	72				
15	73	74	75	76				
16	77	78	79	80				
17	81	82	83	84				
18	85	86	87	88				
19	89	90	91	92				
20	93	94	95	96				
21	97	98						
22	99	100	101	102				
23	103	104	105	106				
24	107	108	109	110				
25	111	112						
26	113	114						
27	115	116	117	118				
28	119	120						
29	121	122	123	124				
30	125	126	127	128				
31	129	130	131	132				
32	133	134	135	136				
33	137	138	139	140				
34	141	142	143	144				
35	145	146	147	148				
36	149	150	151	152				
37	153	154	155	156				
38	157	158	159	160				

12.5 Constraints

The allowable deflection was taken as 16150/180 = 89.7mm. Stress constraints were not explicitly imposed as SANS (2005) were used instead, refer to section 8.6.1.

12.6 Additional Information

Average computation time for the algorithm was 107 minutes for 5000 generations. The best mass (1116.732kg) was obtained with a normalised fitness approach.

12.7 Results

It is clear that the deflection criteria is not the governing constraint for this lattice tower, the average deflection was 57.3mm, which is significantly lower than the permissible deflection of 89.7mm. The first order linear finite element analysis does not consider buckling and second order effects, however the SANS (2005) makes marginal provision for it by classifying elements as class 4 sections and reducing their compression capacity for cases where local buckling can occur. The algorithm takes account of this reduced compression capacity, subsequently such members are subjected to penalty sooner than those who do not buckle which results in a lower probability to survive. It should be noted that the actual area of the element is not reduced, only its compression capacity; the true area of the section is still fed to the finite element analysis.

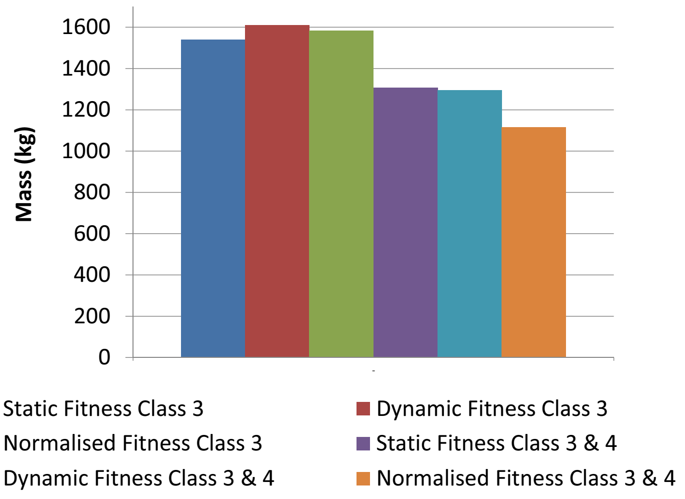


Figure 12.3: Comparison between class 3 section list and combined class 3 and 4 section list

It is interesting to note that the algorithm was initially run with a section list which only contained class 3 sections. This has two major disadvantages. Firstly, the search space is considerably reduced, which implies that the number of possible solutions are less. The fewer solutions there are to choose from, the less likely it is to find a good solution. There is however a fine balance. When the search space becomes too large, the search might get lost and not produce any good result at all. The second disadvantage has to do with the section sizes. Many class 4 elements are smaller in size, which is ideal for bracing members which does not directly carry large loads. The algorithm must simply settle with the next smallest size when these smaller sizes are not available. The difference in weight was rather significant, refer to figure 12.3

12.7.1 Static Fitness Approach

The algorithm takes a few generations to find the first feasible solution. Typical mass behaviour during this stage is a significant increase in mass. The mass will start to decrease once it has found that first feasible solution, a good schema, with which to work with. Observe that there is a rapid decrease in mass right after the first feasible solution was found, where after it takes the algorithm many generations to fine tune. This is a good illustration as to why hybrid algorithms are recommended, refer to section 6.3. The genetic algorithm is very good at searching vast search spaces quickly, but its performance decreases in localised search.

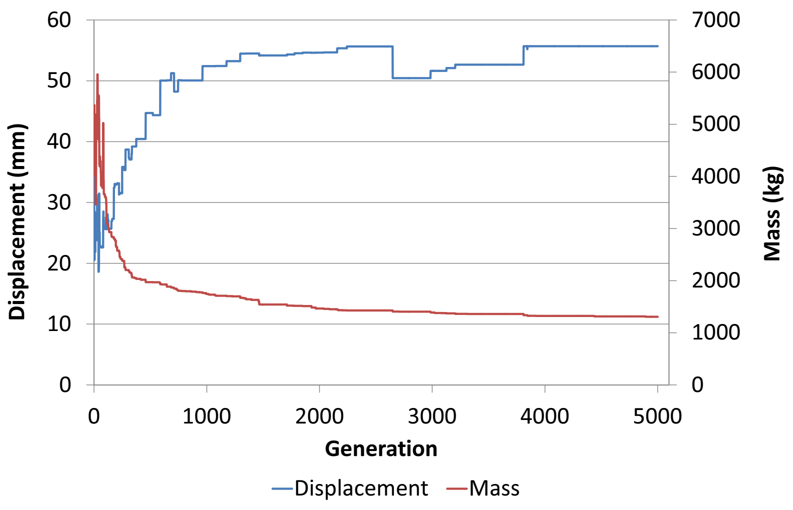


Figure 12.4: 160 Bar Truss Static Fitness - Displacement vs Mass

There is a typical relationship between the fitness of an individual and its deflection. A structure will typically be of lighter mass if it is allowed to deflect more. A lighter mass leads to a lower objective function value which will result in a higher fitness, refer to equation 8.6.26.

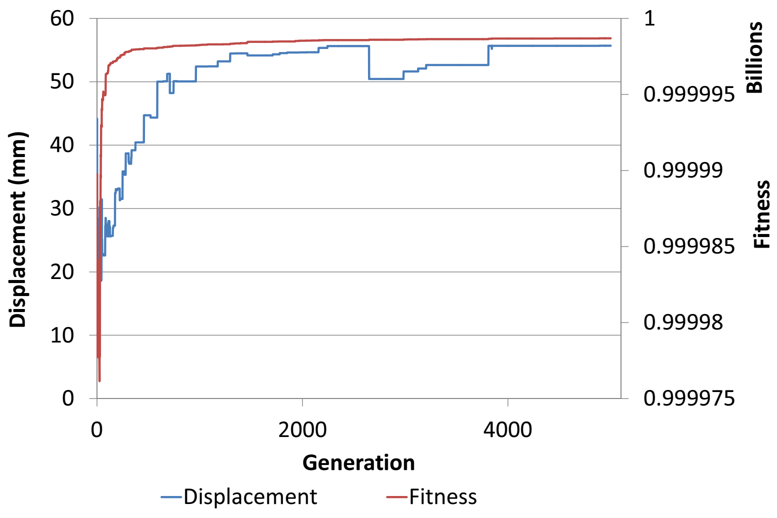


Figure 12.5: 160 Bar Truss Static Fitness - Displacement vs Fitness

12.7.2 Dynamic Fitness Approach

Figure 12.6 illustrates the unpredictable nature of the search for the case where constraint violations are active; the green region indicates generations where the constraints are violated. There is no clear relationship between the fitness and objective values within these green regions. The chaotic nature of the green region in the search is a result of the dynamic penalty function which is generation dependent. There will be a decrease in the fitness function value in this region, even if the best solution from the previous generation is carried through to the next generation. Again, observe the steep initial increase in mass before the gradual optimisation process starts.

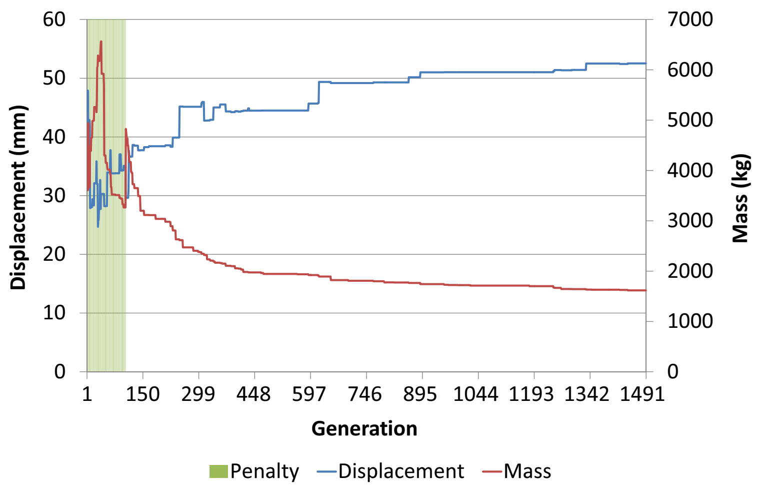


Figure 12.6: 160 Bar Truss Dynamic Fitness - Displacement vs Mass

Figure 12.7 illustrates the dynamic behaviour of the fitness of this approach. The mass decreases as the overall dynamic fitness increases.

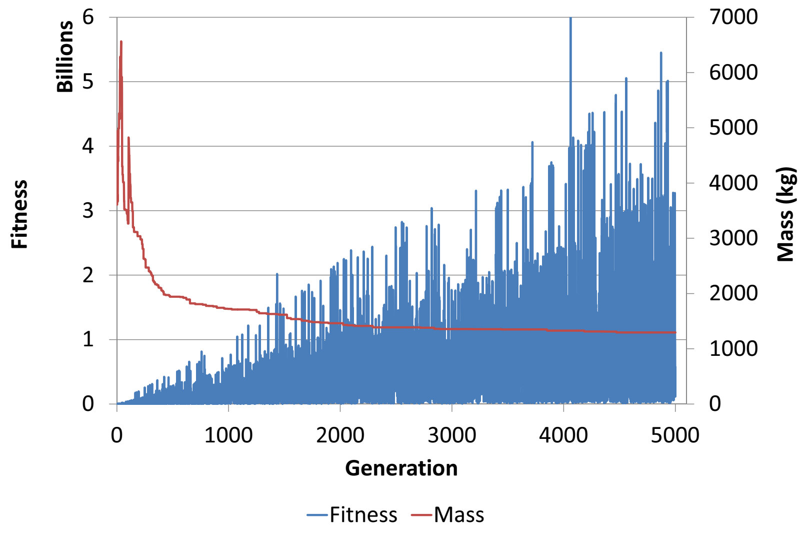


Figure 12.7: 160 Bar Truss Dynamic Fitness - Fitness vs Mass

12.7.3 Normalised Fitness Approach

Normalised fitness is much like the static fitness, except that the fitness lies within the bound (0,1). In contrast to the static fitness, this approach does not need to set a very large constant and to check whether the constant is large enough in order to avoid negative fitness.

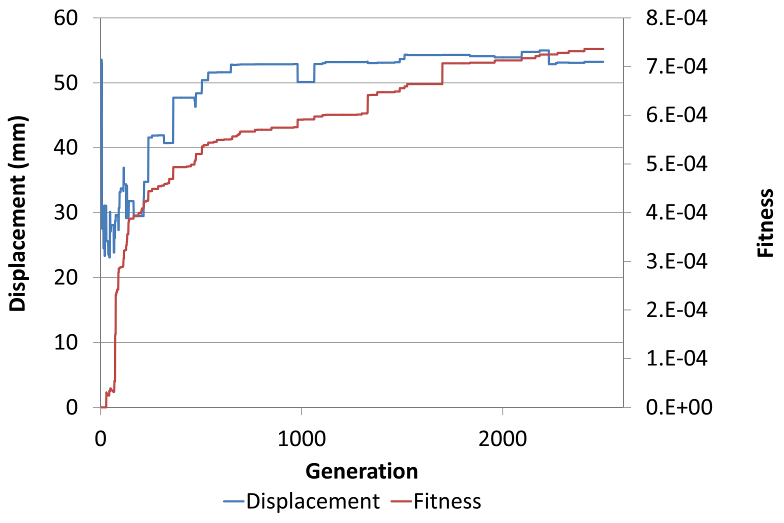


Figure 12.8: 160 Bar Truss Normalised Fitness - Fitness vs Displacement

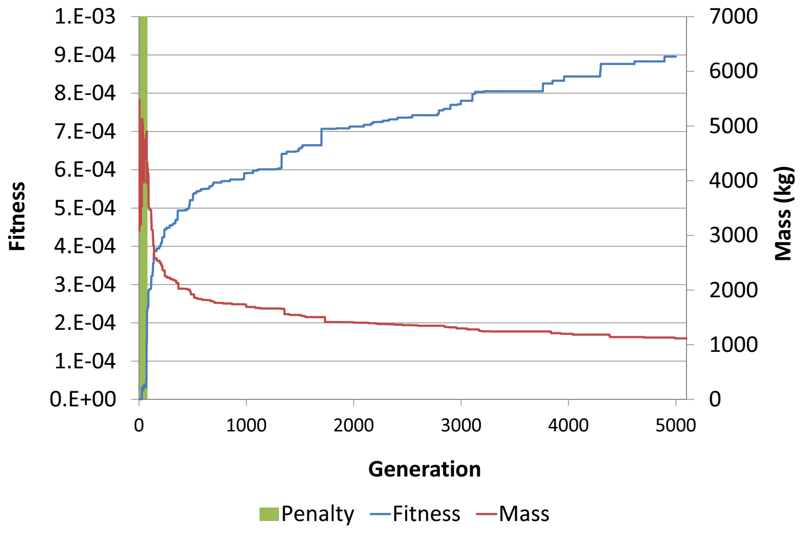


Figure 12.9: 160 Bar Truss Normalised Fitness - Fitness vs Mass

12.8 Comparison

Table 12.7: 160 bar benchmarking problem minimum mass [kg] comparison

Static Mass	Dynamic Mass	Normalised Mass	Groenwold
1308.025	1294.891	1116.732	1359.781

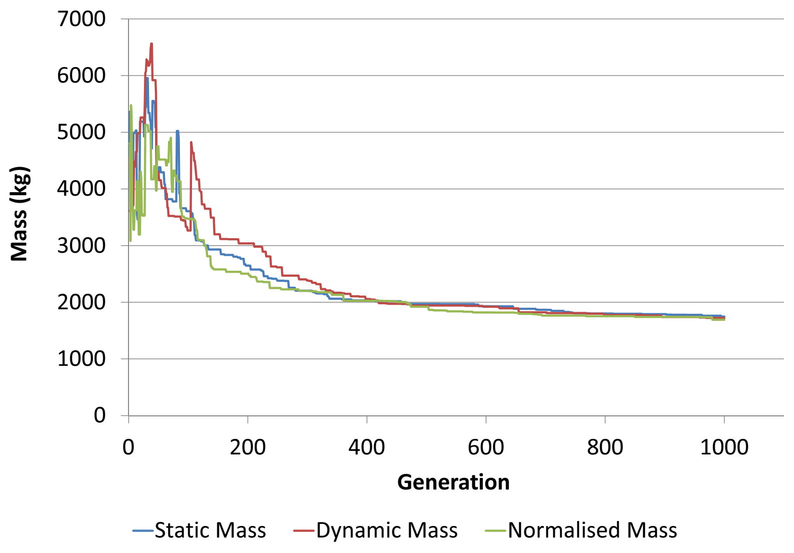


Figure 12.10: 160 Bar Truss Comparison - 1000 Generation Mass Function

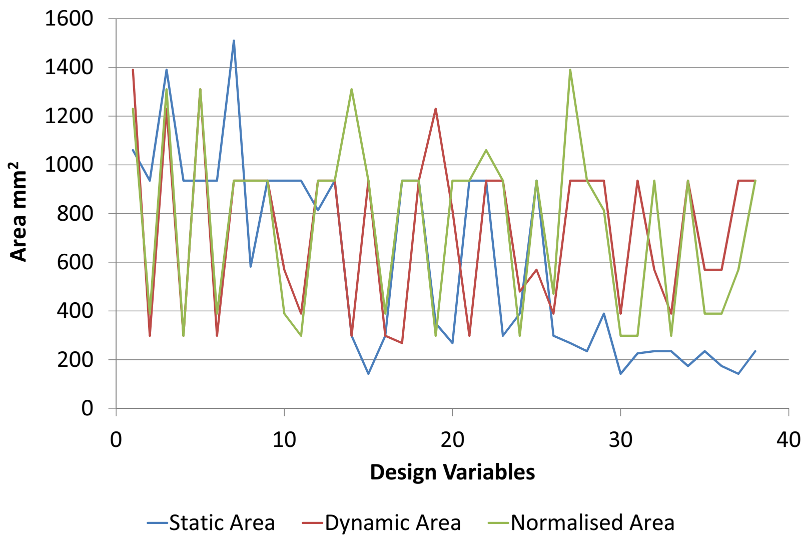


Figure 12.11: 160 Bar Truss Comparison - Design Variables

12.9 Results using Frame Element Results

The frame element's behaviour for this problem is depicted in figure 12.12. Reasons for this approach is thoroughly described in the part V.

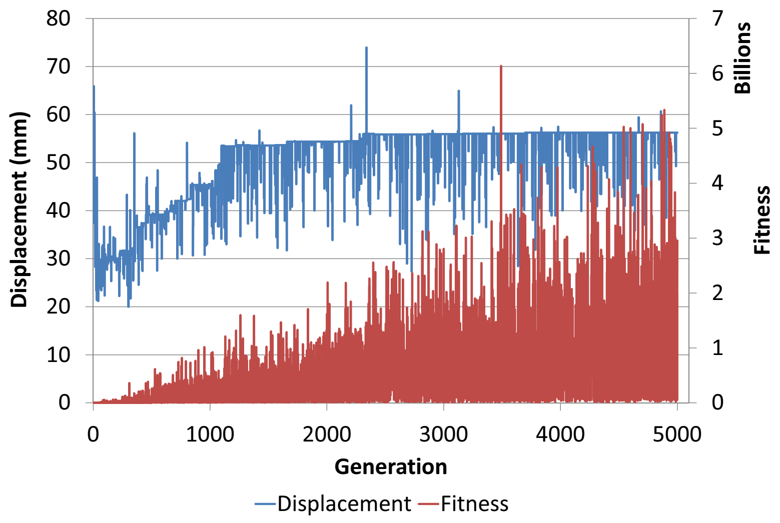


Figure 12.12: 160 Bar Frame Truss - Fitness vs Displacement

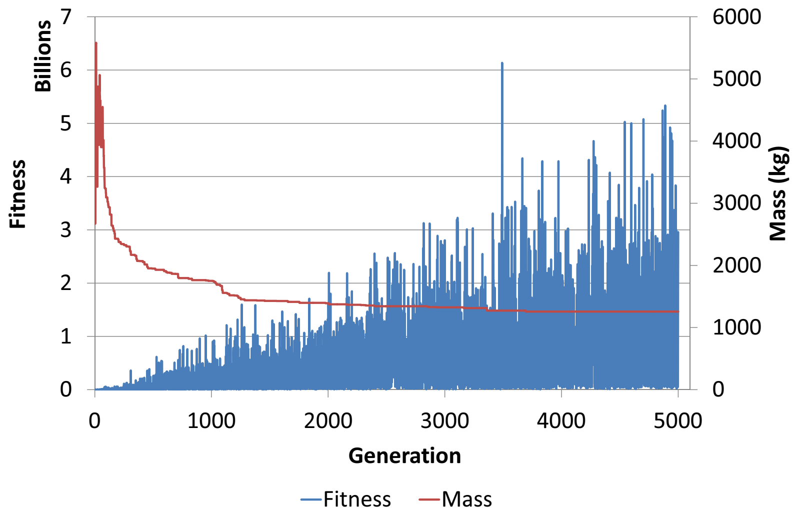


Figure 12.13: 160 Bar Frame Truss - Fitness vs Mass

12.9.1 Comparison Truss and Frame Element

It is only significant to note, at this stage, that the frame element implementation is correct; this approach compares well to the normal truss element analysis and could therefore be used if it be necessary.

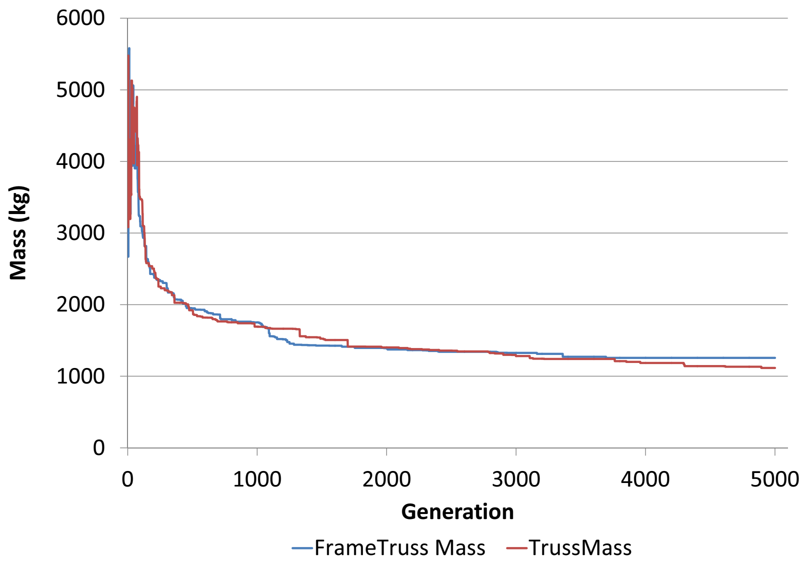


Figure 12.14: 160 Bar Frame Truss Comparison - Mass

Part V

Case Study: Eskom Transmission Tower

Chapter 13

Introduction

The key objectives in overhead power line optimisation are to achieve the lowest maintenance and construction costs, coupled with essential operational reliability (Muftic *et al.*, 2005). According to Diez-Serrano and Marais (2005), the objectives of overhead power line tower design are:

- Produce a safe structure
- Satisfy statutory requirements
- Facilitate maintenance
- Minimise the cost of the structure

The structure considered in this case study is Eskom's *Self-Supporting Suspension 518H Tower*, refer to Appendix 17.3 and Appendix 17.4. The tower consists of 947 elements, see figure 14.2. It is important to take note of the scale of this optimisation problem. The search space size is 947^{49}. According to the BBC (2012) documentary, *To Infinity and Beyond*, one of the largest known numbers to mathematicians is a googol. "A googol is, for example, far larger than the number of atoms in the human body or more than the number of atoms that make up planet earth. This number is even more than all the atoms in the entire observable universe." One cannot even begin to fathom the vastness of this optimisation problem, when it is calculated that this search space is about 7^{45} times larger than a googol. In other words, the number of solutions to the design variables of this problem, is more than 7^{45} times the number of atoms in the universe.

13.1 Objective

This case study only considers one of the objectives mentioned by Diez-Serrano and Marais (2005), which is to minimise the cost of the structure by means of weight minimisation. The objective of this implementation is to create a model that is as close as possible to that of the real-life tower and to investigate key aspects that might make this problem different from standard benchmarking problems.

Finally, once all the issues have been identified, the objective is to optimise the structure or make future recommendations. Establishing and investigating the type of difficulties involved with modelling and optimising real-life structures contribute toward the main objective of this thesis. Therefore, this case study serves as an introductory investigation as to how real life structures are different from benchmarking problems and what kind of steps can be taken to over-come possible issues and errors.

Chapter 14

Design Data

The tower consists of 358 nodes and 947 elements. The section list used is in appendix 17.5. The design data is included in Appendix 17.4.

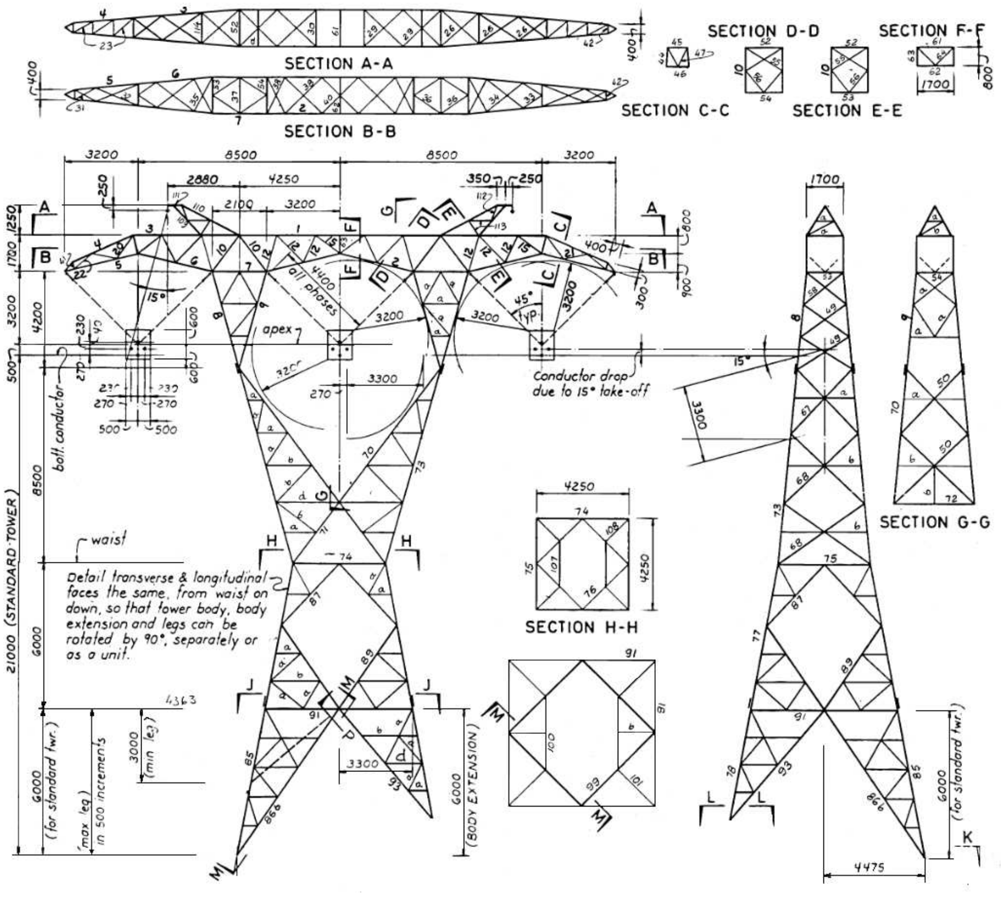

SECTION A-A
SECTION B-B
SECTION C-C
SECTION D-D
SECTION E-E
SECTION F-F
SECTION G-G
SECTION H-H
apex
all phases
Conductor drop due to 15° take-off
bott. conductor
waist
Detail transverse & longitudinal faces the same, from waist on down, so that tower body, body extension and legs can be rotated by 90°, separately or as a unit.
21000 (STANDARD TOWER)
6000 (for standard twr.)
max leg in 500 increments
3000 (min leg)
6000 (BODY EXTENSION)
4475

Figure 14.1: Eskom Transmission Tower Design (*Property of Eskom*)

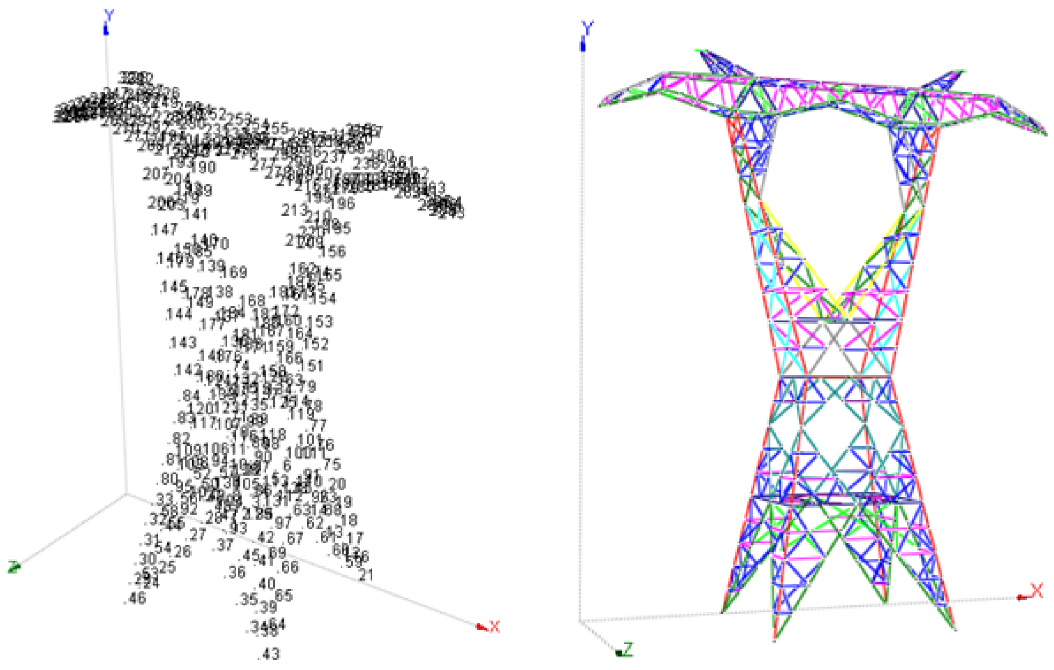


Figure 14.2: Modeling the Eskom Tower

14.1 Load Cases

There are eight defined load cases for this structure: (Data tables available in Appendix 17.6)

- Case 1A
  - High transverse wind
  - $\theta = 90°$
- Case 1AR
  - High transverse wind
  - $\theta = 90°$
  - 38% vertical loads only
- Case 2A
  - All conductors broken
- Case 2BR
  - Broken centre and left conductors

  - 38% vertical loads only

- Case 3
  - Special transverse
- Case 4A
  - Maintenance and construction (left)
- Case 4B
  - Maintenance and construction (centre)
- Case 5
  - Heavy ice

These loads, provided by Eskom, are test loads. However, Eskom used these as design loads, therefore no design factors were introduced in the calculations. These loads were treated as ultimate loads, however serviceability was still taken into account (refer to section 14.3). The motive behind still considering serviceability has to do with the type of load; for example, load 4B is maintenance and construction, here the tower must deflect within limits to ensure for safety of the workers.

Loads that were applied to the conductors, were transferred to places where the conductors are attached to the tower, see figure 14.3. The annotation for figure 14.3 is explained in Appendix 17.6. The reason for this approach is because there are no nodes outside of the structure or type of connection with which to transfer these loads from outside of the structure to the structure itself.

14.2 Grouping

No grouping strategies were programmed for this structure, instead grouping was user defined. Elements of similar sizes, as indicated on the original design, were simply placed in the same group, refer to Appendix 17.4. This could result in a suboptimal structure, as the presumed sizing groups might be incorrect. However, as stated earlier, investigating grouping strategies is not part of the objectives of this research.

14.3 Serviceability

Eskom did not provide any guidance on acceptable deflections for the tower. The serviceability of the structure was therefore roughly based on assumptions regarding Annex D in SANS (2005). The structure was assumed to be an *industrial type building* with a ‘span’ equal to the tower’s height. The maximum deflection was initially limited to the span divided by 180, which was rounded to 155mm. This value was then roughly doubled to 300mm based on engineering judgement regarding the tallness of the structure.

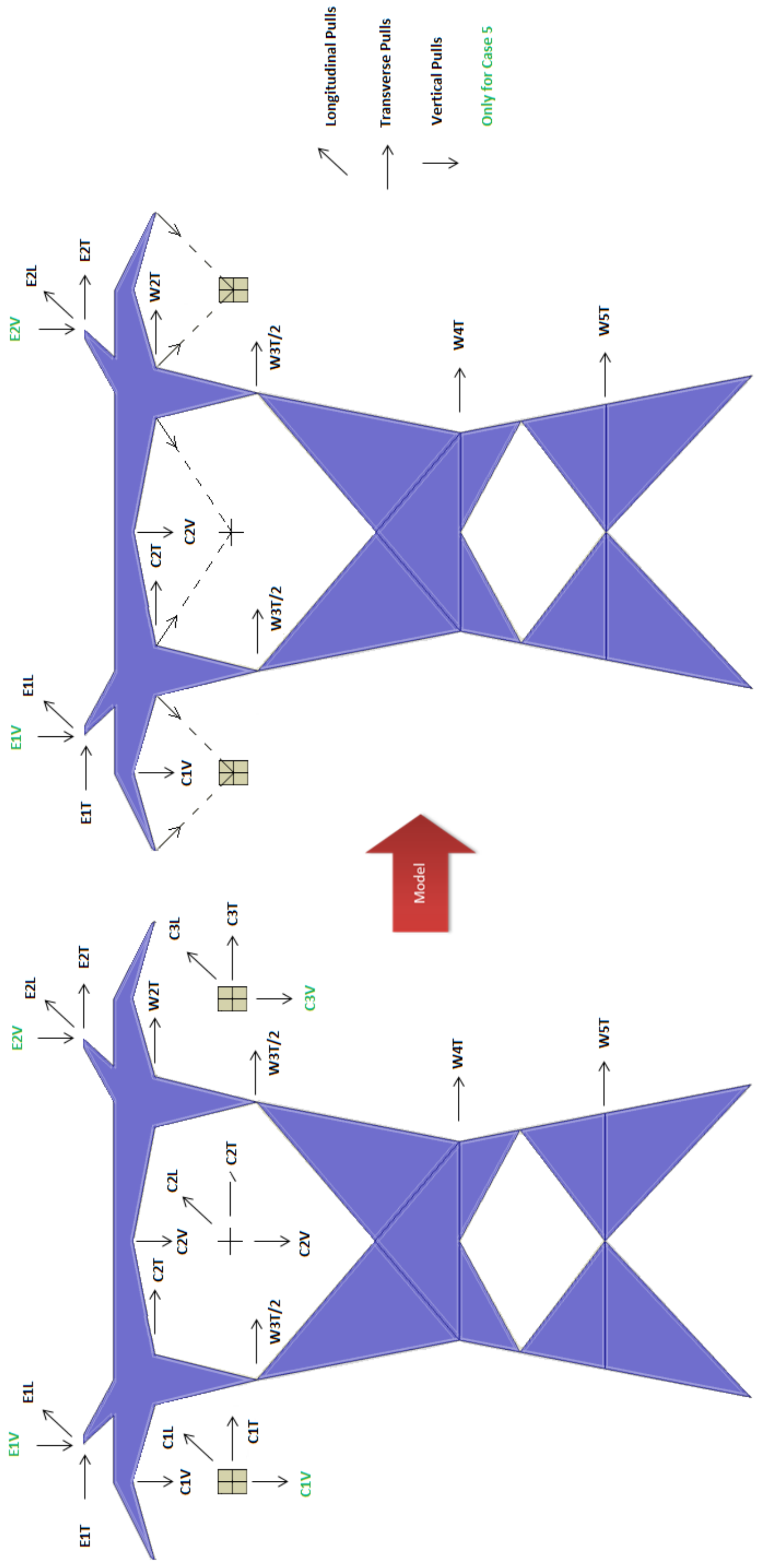


Figure 14.3: Tower Model and Loading

14.4 Genetic Parameters

The following genetic parameters were used in this case study, unless specified otherwise:

Table 14.1: Genetic parameters for the case study

Parameter	**Value**
Crossover probability	0.8
Mutation probability	0.005
Population size	100
Maximum number of generations	5000
Scaling constant	1.5
Number of crossover points	1
Elitism	TRUE
Selection with replacement	TRUE

Chapter 15

Modelling and Implementation

15.1 Modelling Inaccuracies

The program was unable to analyse the structure as a 3D truss model. A possible cause identified for this outcome was unstable planar joints or mechanisms, see section 15.2, due to small modelling inaccuracies. Mechanisms result in an error as there is no stiffness in the perpendicular direction of the plane in which the elements lie. Small modelling inaccuracies arise from calculating nodal coordinates in 3 dimensions for 358 nodes; the 1074 positional values were hand calculated to the 3rd decimal from the engineering drawings. However, the smallest inaccuracy will cause two joined elements to lie in different planes. Figure 15.1 illustrates a mechanism, where elements 1 and 2 are supposed to lie in the plane $x = 1$. This small inaccuracy will cause the structure to loose all its strength, as there is no resistance in the direction perpendicular to the plane (indicated by the red arrow).

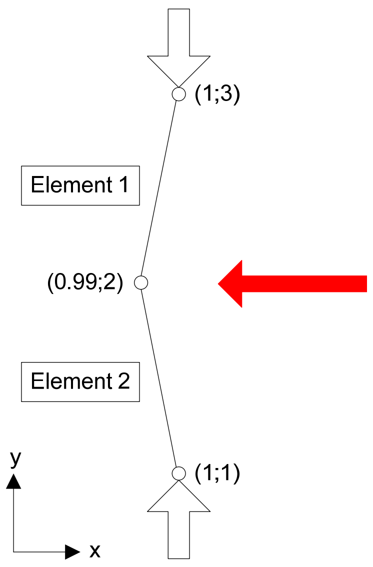


Figure 15.1: Small modelling inaccuracies can result in a mechanism

There are no joints at positions where insulators are positioned (see figure 14.3: the positions of the insulators, indicated by the rectangles, are outside of the structure), therefore there must be joints at places where these insulators are connected to the structure or where concentrated loads are applied. Loading at places outside of the structure (where no nodes are defined) are transferred as illustrated in figure 14.3 to the appropriate nodes.

Members that are enclosed by a system of elements and are redundant, need not be included in the model (PLS, 2011). The inclusion of such members would add unnecessary complications such as extra members and nodes, with no additional information provided by the analysis. These extra members will bear no force, given that it is a linear analysis and all members are truss elements. These members are, however, important in the structure as they would have to carry transverse load in cases where a person might climb the tower and carry 1-3% of the compressive loads. (PLS, 2011). This approach was not implemented, all members where considered in the analysis and in calculating the mass of the tower.

15.2 Dealing with Planar Joints

Planar joints can occur in three dimensional truss element structures. A planar joint is a connection of elements which all lie within the same plane. This joint could start to resemble a mechanism. Figure 15.2 illustrates such a connection. Planar joints are problematic in linear first order analyses as they provide no stiffness in the direction perpendicular to the plane. The program can therefore potentially try to divide by zero which will result in an error. PLS (2011) recommends avoiding planar joints all together. Four methods are recommended in order to avoid the use of planar joints.

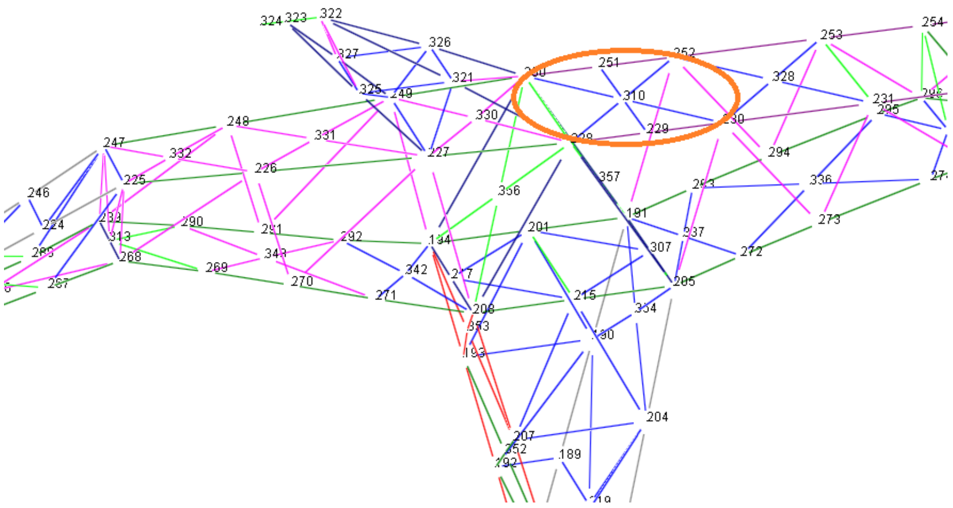


Figure 15.2: A planar joint

15.2.1 Dummy Elements

The adding of a dummy element method recommends adding a fictitious element as a support between the planar joint and any stable point in close proximity (PLS, 2011). This procedure was not selected to solve the modeling problem in order to avoid changes in the force distribution of the structure.

15.2.2 Removing a Degree of Freedom

The removing a degree of freedom method suggests adding a support in the perpendicular direction of the plane, hence the joint will not be able to form a mechanism (PLS, 2011). This approach was not used to solve the modeling problem, as it would require excessive additional work from the user to specify such points and define the supports. The problem needs to be solved without much additional input effort and without making the algorithm problem specific.

15.2.3 Adding Fictitious Springs

The adding fictitious springs method resembles the approach of removing a degree of freedom, however instead of adding supports, springs with a small stiffness of 1 Newton/meter are added to the structure (PLS, 2011). This method was not used in modeling the tower for the same reasons explained above.

15.2.4 Using Frame Elements

The last recommendation by (PLS, 2011) involves replacing the elements which form planar joints with frame elements. These elements have more degrees of freedom with some stiffness in the perpendicular direction of the planar joint. These members are, however, still treated as if they were truss elements. This approach runs the risk of resulting in a tower that is too stiff, especially if all members are modeled as frame elements.

The best solution to this problem was to model the structure with frame elements in order to stabilise planar joints. The structure was still not subjected to design checks involving moments, even though modelling inaccuracies might produce insignificant moments. The structure should not be designed for modelling inaccuracies, in other words, for the limitations of analysing a realistic structure. The engineer should be able to differentiate between modelling issues and actual physical issues. The frame elements, in addition to having an area, are also assigned moments of inertia and St Venant's Torsion constants to provide stiffness. The axial forces from the analysis were used for design checks, this approach was tested in section 12.9.

15.3 Tension-only Members

Members can be defined such that they can only carry tension forces, when a truss structure is designed as if the whole system is made of stiff ropes. Such members are known as tension only members, where

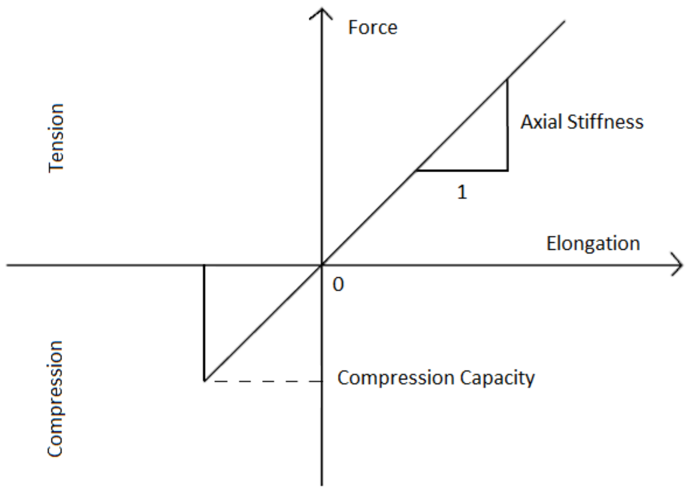


Figure 15.3: Behaviour of Tension-only Element (PLS, 2011)

some programs (such as PROKON) make allowance for such members. Such a member will buckle for the case where the compression force exceeds the member's capacity, thereby loosing its compressive strength, see figure 15.3. Tension members typically have larger slenderness ratios. However, tension only elements add great complexity to linear analyses due to changes in the stiffness matrix after the element buckles (PLS, 2011). No such changes are required for cases without tension members. PLS (2011) recommends avoiding tension-only members, therefore this case study did not make use of tension-only members.

15.4 Effective Length of Members

The effective length of a member affects its compression capacity; a longer effective length results in a smaller compression capacity, refer to section 8.6.1.3. However, the effective length of real life structures of certain members are shortened due to connections. Figure 15.4 illustrates how the effective length of a member is shorter than its actual length due to a connection between two elements. Elements 6047 and 6046 are effectively shortened by the pin connecting them. This approach is not valid if bi-axial bending is considered. Effective length adjustments were made by adding nodes at places where members are connected.

15.5 Length of Members

The member lengths for the pylon model were simply measured from node 1 to node 2 (see figure 7.1) in the algorithm. However, in reality there are at times some member overlap or member length reduction. Member 4011, in figure 15.5, illustrates steel overlap and member 4010 member length reduction. The actual lengths of these elements in the model were taken to the centre of the connection. Connections,

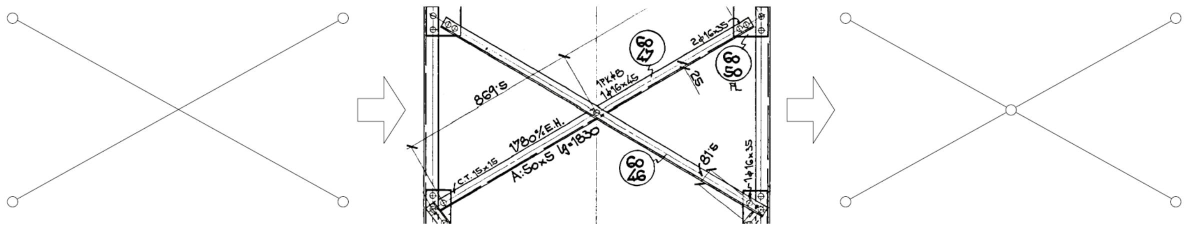


Figure 15.4: Adding nodes for a shorter effective length

however, are not considered in this optimisation procedure; the connections' weight contribution was ignored, such as the gusset plate weight. The weight of the connections of the actual structure is not included in the total weight for the comparison in the results. It is assumed that the total length of elements in the model compared to that of the real structure is the same, as it is assumed that places of overlap and member length reduction balances out.

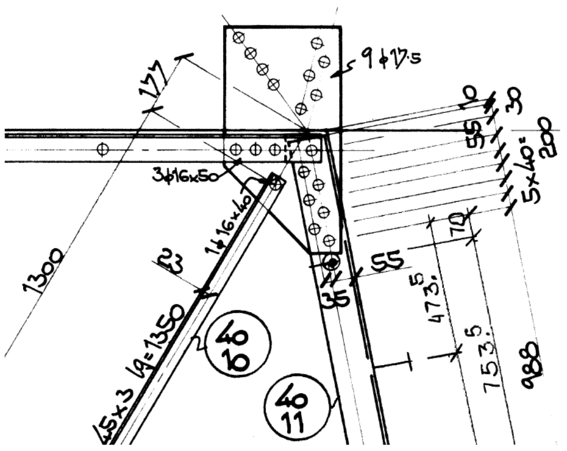


Figure 15.5: Connection illustrating difference in member lengths from model (*Property of Eskom*)

15.6 Notes on Multiple Load Cases

The eight load cases described in section 14.1 cannot simply be used individually to optimize the transmission tower; it would be incorrect to simply run the algorithm for each load case and then to select the heaviest pylon as the solution. The heaviest structure is not necessarily a 'conservative' solution for all the other load cases. Another load case might cause the structure to fail as a result of a given element being under-designed for this load case, even if the structure was designed satisfactory for the initial 'conversative' load case. This can be the case even if the loading for this load case is smaller in magnitude and the pylon (overall) a lighter structure. This is due to possible changes in the direction and position of the loading, or perhaps a given load where there was none before. All load cases must be considered to ensure structural feasibility. This can, however, not be achieved by simply adding all the load cases together. Such a load condition will be unrealistic and will result in a severely over-designed structure, which defeats all optimisation purposes. Eskom did not provide any load combinations, nonetheless, load combinations can easily be dealt with by simply creating a new combined load case. The challenge lies in solving for multiple load cases, and not in solving for a load combination. The solution must somehow be valid for all load cases, separately but simultaneously. This can be done by subjecting each model to all 8 loading conditions one by one. Some violation 'bookkeeping' must be kept for the case where a solution violates a constraint for a given load combination. The performance of the structure is evaluated and summarised for all loading conditions. This suggests that the structure with the best overall performance will have the highest fitness and the highest probability to be selected for the next generation, refer to section 4.4.1. Stress, displacement and slenderness violations must be checked by performing an analysis for each load condition.

This problem becomes difficult in the sense that it takes a great amount of computing power. Each model must be tested for all 8 cases, one after another; that is each individual for the whole population in a generation. The number of analyses increases 16 fold for every run. A population of 100, for the pylon model, takes one and a half days to produce 5000 generations without considering multiple load cases. It should be noted that it is not in this case the GA that makes the run so computationally expensive.

The finite element analysis computations, or more accurately, inverting a stiffness matrix, for 947 elements (358 nodes) with 6 degrees of freedom is computationally expensive. Equation 15.6.1 expresses the basic stiffness equation which forms the basis for solving a finite element problem, refer to (Cook *et al.*, 2002) for more information.

$$\mathbf{Kd} = \mathbf{F} \tag{15.6.1}$$

The stiffness matrix, equation 15.6.2, becomes larger and larger for each element added to a structure; smaller elemental matrices can be added to form one larger system stiffness matrix for the complete structure.

$$\mathbf{k} = \begin{bmatrix} \frac{EA}{l} & 0 & 0 & -\frac{EA}{l} & 0 & 0 \\ 0 & \frac{12EI}{l^3} & \frac{6EI}{l^2} & 0 & -\frac{12EI}{l^3} & \frac{6EI}{l^2} \\ 0 & \frac{6EI}{l^2} & \frac{4EI}{l} & 0 & -\frac{6EI}{l^2} & \frac{2EI}{l} \\ -\frac{EA}{l} & 0 & 0 & \frac{EA}{l} & 0 & 0 \\ 0 & -\frac{12EI}{l^3} & -\frac{6EI}{l^2} & 0 & \frac{12EI}{l^3} & -\frac{6EI}{l^2} \\ 0 & \frac{6EI}{l^2} & \frac{2EI}{l} & 0 & -\frac{6EI}{l^2} & \frac{4EI}{l} \end{bmatrix} \tag{15.6.2}$$

However, the analysis of each new load case does not need a new inverted stiffness matrix as the model itself did not change. The program can be coded as such that the inversion of the stiffness matrix is performed only once, where after simple matrix multiplication can be performed to solve for the unknown displacements and rotations, and back substitution to solve for all the unknown forces. This approach should, for all practical purposes, take about the same amount of time as solving for one load case. Moreover, there is some playoff between the number of individuals in a population and the number of generations. The number of individuals can be increased to, for example, 200 and therefore the number of generations can be decreased. There is however, a fine balance; increasing the number of individuals increases the number of analyses that must be performed, however decreasing the number of generations decreases the number of analyses. In other words, more individuals require more analyses but fewer generations. Furthermore, it is not necessary to run the algorithm for 5000 generations, as minimal changes are made to the solution for the last four thousand generations.

The first approach to solving the multiple load case problem was to sum all the penalised objective functions and to divide the sum by the number of load combinations. This approach provides some average performance of the tower under the various load conditions.

```
for each model $i$ in the population {
    for each load case $l$ {
        create a femModel
        analyse the femModel
        for each element $e$ in the femModel {
            store internal axial forces $F_{i_e,l}$
            store model displacement $= \max\{D_l\}$
        }
    find the penalised objective $\phi_{i_p,l}$ for load case $l$
    }
    find the penalised objective for the model $\phi_{p,i} = \frac{\sum_{l=1}^{8} \phi_{i_p,l}}{8}$
    find the fitness $\zeta_i$ with $\phi_{p,i}$
}
```

This approach did not provide good results, refer to figure 15.6. One argument is that the average of all the penalised objective functions is not necessarily a resemblance of any individual penalised objective value for a given load case. For example, consider the hypothetical objective function values in table 15.1 for two load cases:

Table 15.1: Hypothetical objective function values of 2 load cases for explanation purposes

Load Case	Objective Function Value
1	1
2	10

The average of these two values is 5.5. This number does not tell the algorithm anything about any of the two load cases. Furthermore, consider for argument's sake a thousand load cases with a large variance, providing it with average information is equivalent to expecting the algorithm to solve for anything that can happen to the structure without giving it any particular information on the behaviour of the structure subjected to the given load cases.

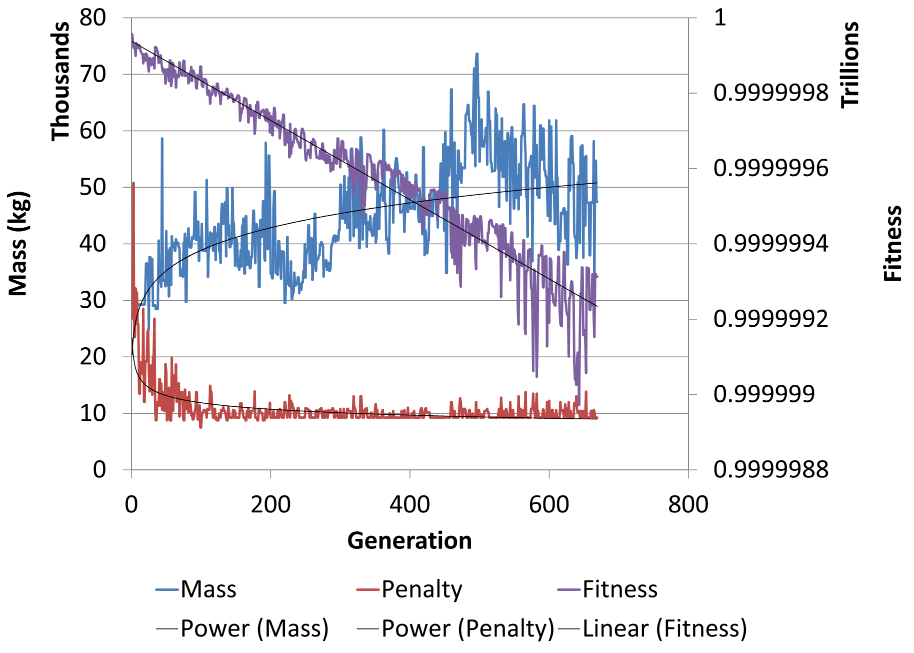


Figure 15.6: Performance for static fitness function with an average penalised objective function approach

Observe that the fitness function values decrease linearly as the generations increase. This is due to a built-in generation parameter in the penalty function. The penalty is increased for every new generation that constraint violations are present, even if the solution did not change. The mass basically jumps around at random, which indicates that the search has become random.

The second approach was to change the penalised objective function value from the average of all load cases, to the most severe for all load cases.

```
for each model $i$ in the population {
    for each load case $l$ {
            create a femModel
            analyse the femModel
            for each element $e$ in the femModel {
                store internal axial forces $F_{i_e,l}$
                store model displacement $= \max\{D_l\}$
            }
        find the penalised objective $\phi_{i_p,l}$ for load case $l$
        }
        find the penalised objective for the model $\phi_{p,i} = \max\{\phi_{i_p,l}\}$ for $l = 1, 2, ..., 8$
        find the fitness $\zeta_i$ with $\phi_{p,i}$
    }
```

Figure 15.7 illustrates the second approach for solving the multiple load case problem. This approach does result in better algorithmic behaviour, however, the algorithm still struggles to find that first feasible solution. One reason might be that the algorithm is provided too little information about the load cases, because only the load case which results in the lowest fitness for that particular generation is communicated to the algorithm. It might appear from the algorithm's 'point of view' that the objective function is continuously changing and can therefore not find direction in the search.

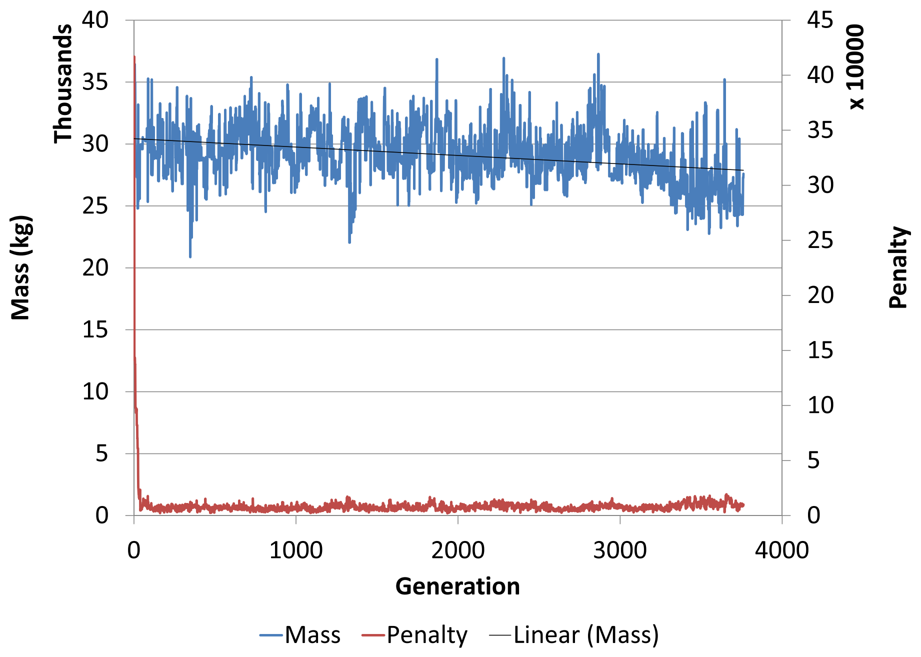


Figure 15.7: Performance for static fitness function with a maximum penalised objective function approach

The outcomes above suggest that the solution of the multiple load case problem must somehow include information of all the load cases. The study suggests gathering more information on the behaviour of the model for all the load cases applied to it by performing a form of pre-evaluation for each and then somehow including the outcome in the objective, be it in a weighted form or simply setting some standard. There is a definite need to perform a sensitivity study; setting the genetic parameters to unfitting values could possibly disturb the search and restrict it from finding an outcome. The optimisation procedure must be of such that the direction of the search can become clear to the genetic algorithm, in other words, the objective function must be able to provide proper guidance towards finding fit solutions.

15.7 Provisional Solution for the Case Study

The provisional solution provided by this study is simply an optimised solution for the critical load case 2A. The allowable deflection was taken to be 300mm. The element sizes are given in table 15.2.

The total mass of the actual Eskom tower is 30,392 tons. This weight excludes the weight of connections, for example gusset plates and bolting. The weight found by the algorithm for the critical load case is 30,644 tons.

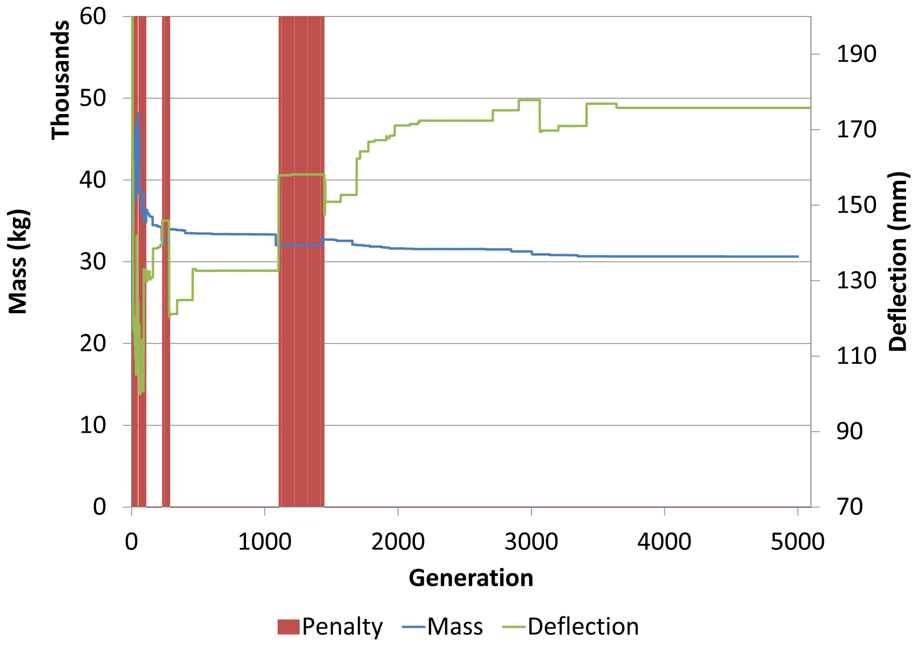


Figure 15.8: Critical load case mass and deflection behaviour

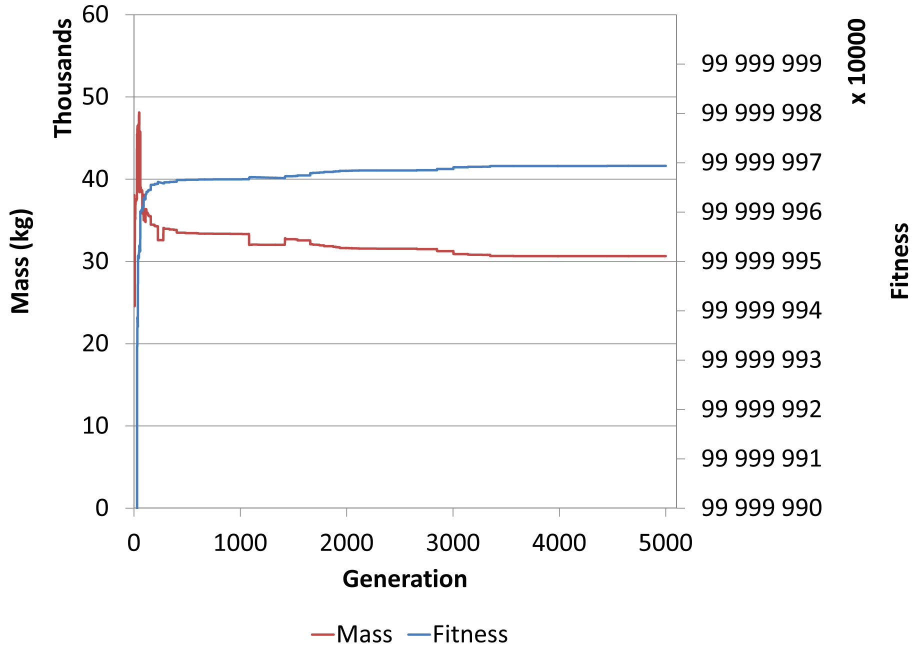


Figure 15.9: Critical load case mass and fitness behaviour

Table 15.2: Element sizes for the given design variables

Design Variable	Area mm^2
1	4300
2	430
3	4300
4	4300
5	2750
6	1510
7	1710
8	3480
9	935
10	935
11	1060
12	430
13	935
14	1310
15	935
16	1870
17	582
18	935
19	935
20	430
21	268
22	582
23	1550
24	1390
25	235

15.8 Alternative Proposal for the Multiple Load Cases

Spillers and MacBain (2009) argue that a problem with two load cases can be decomposed into a single load case. The authors do this by defining a primal and dual problem for the two load cases. The primal problem is a minimisation problem for the maximum axial force in a member as a result of the 2 load cases. The dual problem is obtained from standard linear progamming. The authors use two identities to decompose the problem. Refer to Spillers and MacBain (2009) for more information. This approach, however, does not work for three or more load cases and is therefore rather limiting.

Konak *et al.* (2006) mentions two traditional methods with which to approach such a problem. The first is to combine all the objectives by some means, for example, the weighted sum method. This method is however, not highly recommended due to difficulties with weights and balancing of the objectives. The second approach is to add all the objectives, except for one, to the constraints and then to optimise for the remaining objective. This method also does not come highly recommended due to difficulties in finding suitable constraining values. A new approach is needed with which to solve the problem. The following are important to note:

- The objective function changes with each new load case.
- It is perhaps not possible to create one master objective function, due to the conflicting nature of some of the objectives.

Perhaps the aim of the problem should move towards finding a set of solutions, where each solution is acceptable to a satisfying degree without dominating other solutions. Konak *et al.* (2006) state that the GA is ideal to handle such problems.

15.8.1 Multi-Criteria Optimisation

Thus far the fitness for a specific solution was expressed as a single number, even though many parameters were involved. This cannot be done for the multiple load case problem. In this case, solutions can be obtained in a number of ways, where the solutions themselves cannot necessarily be combined which makes it impossible to find a single expression for the fitness. Coley (1999) explained multi-criteria optimisation with the following example: An engineer might wish to minimise the weight of a steel structure so that the cost of the structure will be at a minimum. However, the structure must also be sufficiently safe and failure will lead to great cost implications. It is clear that the minimum cost might not necessarily be a feasible solution due to the risks involved, therefore for this problem both the weight of the structure and its safety must be considered. Ideally, the solution should provide the lowest possible cost of structure for the highest possible safety. One way of dealing with such problems is by applying the concept of Pareto Optimality. Castro and Barbosa (2000) defines a Pareto set as “the set of solutions which are such that no improvement can be made in one objective without

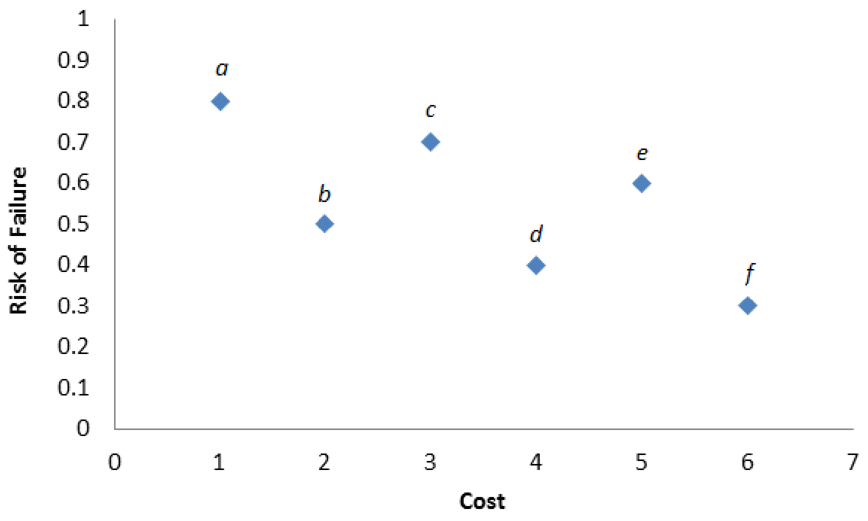


Figure 15.10: Pareto Optimality (Coley, 1999)

deteriorating at least one of the other objectives is called the Pareto set of non-dominated solutions and an approximation of it would be very useful in order to get insight into the problem and assist the decision making process." In figure 15.10, the plotted points a to f represent possible solutions to Coley (1999)'s example. a is a solution which holds the lowest cost, but has the highest risk of failure (the x axis is some arbitrary normalised cost unit). On the other hand, f has the lowest risk of failure, but with the highest cost. e and c are said to be dominated, this is due to the fact that there are other solutions that offer both reduced risk and cost. These superior solutions are termed non-dominated.

Figure 15.11 enables an engineer to make more informed decisions. There are different ways of implementing Pareto optimality with a genetic algorithm during the selection procedure. One way is to divide individuals into nondominate and dominate groups, where the nondominated are assigned a rank value of 1. Individuals are removed from the selection pool once they are assigned to the nondominated set. The whole process is repeated, but this time the rank value is 2. The process is terminated as soon as all members are ranked.

A standard formulation for the multi-objective problem with K objectives can be defined as: (Konak *et al.*, 2006)

For a decision variable vector with n dimensions, $\mathbf{x} = \{x_1, ..., x_n\}$, in a solution space $\mathbf{X}$, find $\mathbf{x}^*$ such that $\phi(x^*) = \{\phi_1(\mathbf{x}^*), ..., \phi_k(\mathbf{x}^*)\}$ is a minimum, where the solution is subjected to constraints $g_j(\mathbf{x}^*) \geqslant 0$ for $j = 1, ..., J$ and bounds $h_m(\mathbf{x}*) = 0$ for $m = 1, ..., M$. A feasible solution $\mathbf{x}$ will dominate a feasible solution $\mathbf{y}$ if $\phi_i(\mathbf{x}) \leqslant \phi_i(\mathbf{y})$ for $i = 1, ..., K$ and $\phi_j(\mathbf{x}) < \phi_j(\mathbf{y})$ for at least one objective function j.

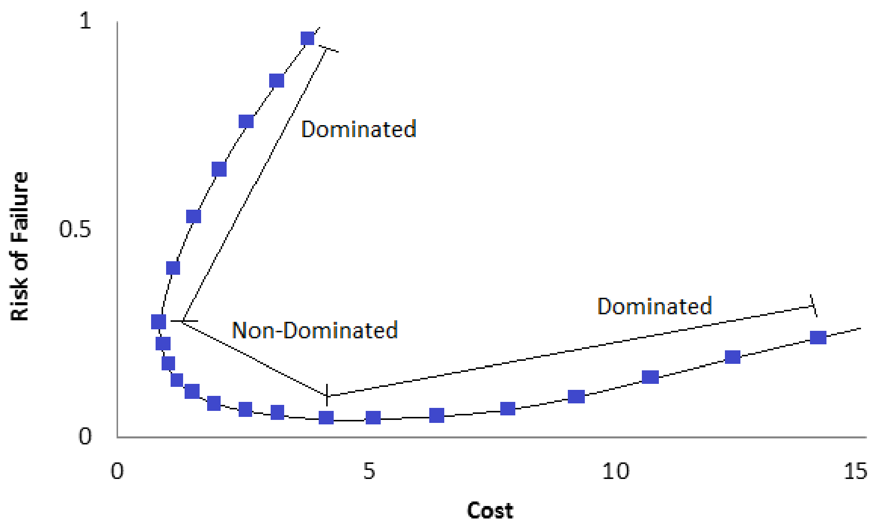


Figure 15.11: Dominated and Non-Dominated Solutions (Coley, 1999)

In a similar way that safety and minimum cost were two objectives in the example problem above, so too can different load cases be different objectives. Castro and Barbosa (2000) suggest an algorithm which modifies an evenly distributed set of solutions by ranking the set based on its non-domination properties, after which a filter is created in order to preserve the Pareto set solutions. This algorithm will need special operators such as exclusion. Castro and Barbosa (2000) state that the following features make the genetic algorithm favourable for multi-criteria optimisation through means of a Pareto set:

- The algorithm is population based
- It only needs objective function values
- The use of probabilistic transition rules makes it less susceptible to local minima

15.8.2 Implementing a Pareto Set in the Genetic Algorithm

Osyczka and Kundu (1995) explains that the basic concept of incorporating a Pareto set hinges on ascribing fitness as such that greater fitness is awarded to solutions further away from the current Pareto set. Award every Pareto solution a *distance* value denoted as d_l for $l = 1, ..., l_p$ where l_p indicates the number of existing Pareto solutions. Let $\mathbf{f}_l = [f_{1l}, ..., f_{Il}]^T$ be the objective functions vector for the l^{th} Pareto solution. The exterior penalised objective function suggested by Osyczka and Kundu (1995) is:

$$\phi_{i,p}(\mathbf{x}) = \phi_i(\mathbf{x}) + r\sum_{m=1}^{M}[h_m(\mathbf{x})]^2 + r\sum_{k=1}^{K}G_k[g_k(\mathbf{x})]^2 \text{ for } i = 1, ..., I \tag{15.8.1}$$

Where $G_k = 0$ for $g_k(\mathbf{x}) \geqslant 0$ and $G_k = 1$ for $g_k(\mathbf{x}) < 0$ and r simply scales the penalty. The relative distance for each new solution $\mathbf{x}$ is:

$$z_l(\mathbf{x}) = \sqrt{\sum_{i=1}^{I} \left(\frac{f_{il} - \phi_{p,i}(\mathbf{x})}{f_{il}} \right)^2} \text{ for } l = 1, ..., l_p \tag{15.8.2}$$

15.8.2.1 Pseudo Code (Osyczka and Kundu, 1995)

```
begin at generation $g = 1$ and individual $i = 1$
    create initial population and set the first random solution as
    the Pareto optimal solution
        set Pareto optimal solution fitness $F$ equal to $d_1$
            value $d_1$ is a random starting point
    if g = 1
        generate random solution x
        else proceed from $*$
    for solution x
        calculate the relative distances with equation 15.8.2
    find $z_{l^*}(\mathbf{x}) = \min\{z_l(\mathbf{x})\}$ for $l = 1, ..., l_p$
        where index $l^*$ indicates which Pareto solution is closest to the newly
        generated solution x
    $*$ if the last solution x is a new Pareto solution {
        calculate the fitness for the new Pareto solution
            $F = d_{l^*} + z_{l^*}(\mathbf{x})$
        update existing Pareto set
            remove all entries in old Pareto set that is dominated by the new set
            add the rest of the new set to old set
            set $z$ of new Pareto solution equal to $F$}
    $*$ else {
        calculate the fitness for this solution
            $F = d_{l^*} - z_{l^*}(\mathbf{x})$
            check for negative fitness
                if $F < 0$ then $F = 0$ }
    find the maximum distance from all existing Pareto solutions
        $d_{\max} = \max\{d_l\}$ for $l = 1, ..., l_p$
            where $l_p$ is the number of Pareto solutions
    substitute $d_l = d_{\max}$ for $l = 1, ..., l_p$
    g = g + 1
```

Algorithm termination criteria were left out of the pseudo code, this can simply be a predefined number of maximum generations or some other criteria.

This approach must first be tested with test functions to ensure that the new code is adequate. It would have to follow the same procedure as was done in this study, by applying the algorithm to test problems with known outcomes before it can be applied to a real life problems with unknown outcomes. In this way, the adequacy of the new algorithm is extrapolated from the known to the unknown.

Part VI

Closure

Chapter 16

Conclusion

The first aim of the research was to thoroughly investigate the mechanics behind the genetic algorithm and to introduce this research field to the Structural Department of Civil Engineering at Stellenbosch University. There after the study implemented a genetic algorithm to serve as an optimisation tool to optimise steel plane and space trusses, along with a Finite Element Method Program, whilst taking into account various constraints. These constraints were typically stress and displacement constraints, or constraints provided by SANS 10162, which would also account for slenderness and buckling effects. Trusses were optimised for their weight, hence the design variables were the profiles' cross sectional areas. However, for future research, it could be extended to a multi-objective optimisation process.

With the use of benchmarking problems, it was proven that the algorithm produces competitive results. The algorithm was then adapted and modified, first for a theoretical 160 bar tower, implemented with South African design standards, and then for a complete realistic South African practical application, the Standard Eskom Transmission Tower.

The algorithm provided solutions to discrete structural optimisation problems within acceptable times for research purposes. Keeping in mind the No Free Lunch theorems, the purpose of this study was not to claim that the GA is the ultimate solution to all optimisation problems, however merely to illustrate that it is a good choice for structural optimisation. Solutions found by the algorithm were feasible, both mathematically and practically and no gradient computations were necessary.

Genetic algorithms are slower than traditional methods, however with present day computing power this is not necessarily a disadvantage anymore (Rajeev and Krishnamoorthy, 1992). The GA uses a statistical approach to navigate the search, in contrast to deterministic methods, this method is probabilistic and stochastic. The algorithm's behaviour can be predicted, but not determined exactly. The computations for each solution in a generation are independent, this allows for parallel computing. The Schema theorem behind the genetic algorithm gives it a mathematical foundation upon which the gain and loss of schema in succeeding generations operate. This theorem establishes that the overall fitness of a population improves as the run progresses through the generations, which is a fundamental prerequisite for any optimisation method. However, it cannot be mathematically deduced that the algorithm will converge at the true optimum. This is due to the fact that the algorithm is not calculus based. As genetic algorithms produce better results than traditional methods for engineering application, it can be accepted as a suitable optimisation tool for structural engineering design (Rajeev and Krishnamoorthy, 1992).

Chapter 17

Future Research

This study only focused on sizing optimisation of the composing elements of a structure for the case of one dimensional bar/truss elements. However, genetic algorithms can be applied to various forms of structural optimisation and structural optimisation types. These include topology and shape optimisation, frame elements with more degrees of freedom and additional moment calculations. The algorithm could be extended to include all the various forms of optimisation, as well as using different types of elements. Because the code is now readily available, further research into areas such as improved fitness functions, penalties or perhaps even a completely new innovative way of dealing with constraints are now possible. The basic genetic processes is independent and unattached to details of the problem at hand, therefore to establish a primary genetic algorithm library that contains all the genetic mechanisms, operators and approaches with an interface to an objective function (which will be problem specific) would be a convenient optimisation tool. The program could be extended to other disciplines; it need not only serve as structural optimisation. However, the code structure needs refinement and improvement. Work is needed into solving computer memory problems and to reduce the algorithm's run time. This research recommends coding a finite element analysis uniquely for the use of the GA, that is memory efficient for a large number of generations. A means of applying the GA to problems with a large number of design variables or dimensions must be investigated. The program was implemented for a single objective; to minimise the mass of the structure. However, it can easily be extended to a multi-objective optimisation program. The individual can be divided into parts, the chromosomes of different parts could refer to different objectives. Krishnamoorthy *et al.* (2002) implemented shape optimisation for tubes, the first part indicates the number of groups necessary to achieve a final solution and the second part is used for the thickness of the section. These two parts were real-coded (not in binary format). The third part had a binary encoding scheme which related to group cross sections.

The whole optimisation process can be combined with reliability. The optimisation of structures has a direct effect on the cost, where cost needs to be kept at a minimum. This minimum cost is however not only governed by structural principles, but also reliability theory. Eventually the aim of the research

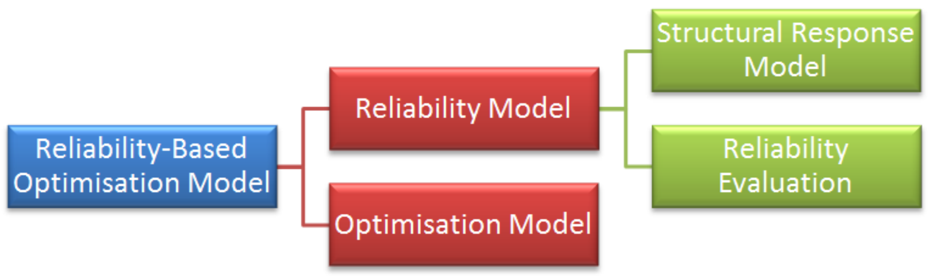

Reliability-Based Optimisation Model
Reliability Model
Optimisation Model
Structural Response Model
Reliability Evaluation

Figure 17.1: Reliability-based optimisation approach by Enevoldsen and Sorensen (1994)

initiative is to perform reliability-based optimisation of a structure, by taking into consideration certain reliability principles whilst keeping the structure at an optimum, refer to figure 17.1. Research needs to investigate how reliability based optimisation models for large realistic structures can be formulated and simplified without a substantial loss of information. Also, to establish whether there is an overlap between optimisation and reliability. In essence, future research is needed to formulate a reliability based optimisation model which consists of a reliability model and an optimisation model which are linked together.

Penalty techniques have been greatly criticised, even though they remain the most common way with which to convert a constrained problem into an unconstrained problem for the case of the genetic algorithm. Perhaps a complete new approach is needed that involves innovative thinking by going back to genetic and evolutionary principles. These principles have proved to be of much value in this study and it cannot be assumed that the concept is already fully developed. The same applies for the encoding scheme in order to increase the number of design variables exponentially; it should be kept in mind that chromosomes store unimaginable amounts of information.

With regard to the analysis, this implementation was only a first order linear analysis. It would be more accurate to implement a second order analysis, also taking into account second order effects and stability.

17.1 Different Approach to Optimisation

In this study, the structure was optimised with respect to stress and displacement constraints, or through implementing the South African design code. Another approach to optimise the structure, other than having stress and displacement constraints, is through constrained vibration frequencies and modes with an eigenvalue method. This will not only be a useful alternative means with which to approach a solution from a different engineering angle, but will also be needed in cases of earthquake design where ground movement becomes an additional load case and structural dynamic movement a constraint.

17.2 A structure with Frame and Truss Elements

Rather than defining a whole structure as frame elements or truss elements, it might be better to model some elements as frame elements and others as truss elements. This will reduce the risk of modelling a structure that is too stiff, refer to section 15.2.4. The aim would be to use as few as possible frame elements, just enough to make the structure stable.

17.3 Hybrid Algorithm

Some classical optimisation technique could be implemented after the genetic algorithm has finished, refer to section 6.3. This combined algorithm would result in an answer closer to the optimum. The real extent to which the other advanced operators benefit a search needs investigation, refer to section 6.

17.4 Upgrading the Genetic Algorithm

Geometry optimisation can be implemented by the GA. In other words, the GA can be coded in such a fashion where the user only needs to define certein key nodes in order to give the structure some shape and to apply loading. The algorithm will then completely design the structure, from defining the geometry to choosing the appropriate element sizes and even which element shapes are best suited. Elements and nodes are removed from a grid of points which are all connected via elements. This grid can vary in density. Hultman (2010) explains that special constraint criteria are needed in order to ensure that the structure remains stable, e.g. the lattice structure must not become a mechanism. Elements must be chosen as such that the structure still deflects within limits. Another constraint will be the structure's constructability. For this case two or more elements cannot share the same end nodes. Also, an element cannot begin and end at the same node. This is just one approach with which the genetic algorithm can be converted into a very powerful tool. Better ways need to be investigated into defining and implementing constraints in such a fashion as to produce feasible structures.

17.5 Different Types of Structures

This study only considered stable lattice structures, however, perhaps it is no longer vital to optimise a specific type of structure, but rather finding the best type which the structure must be. For example, a transmission tower is perhaps only at its optimum when built as a guyed stayed structure, refer to figure 17.2.

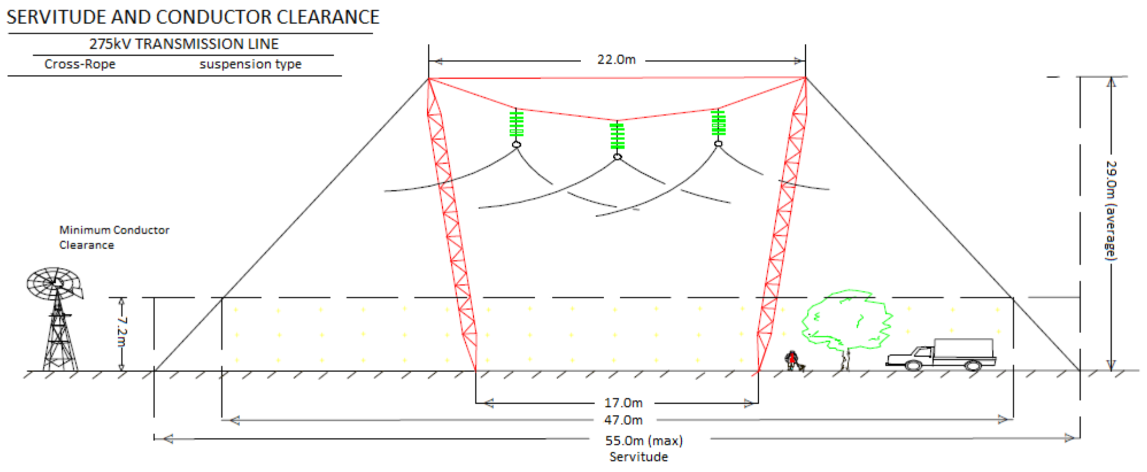

SERVITUDE AND CONDUCTOR CLEARANCE
275kV TRANSMISSION LINE
Cross-Rope suspension type
22.0m
29.0m (average)
Minimum Conductor Clearance
7.2m
17.0m
47.0m
55.0m (max)
Servitude

Figure 17.2: Eskom Cross Rope Suspension Tower (Makhura, 2010)

List of References

Auer, B.J. (2005 April). *Size and Shape Optimization of Frame and Truss Structures Through Evolutionary Methods*. Master's thesis.

BBC (2012 March). To infinity and beyond. Part of the Brittish Broadcasting Commision Horizon series. Date accessed 26 September 2013.
Available at: `http://www.bbc.co.uk/programmes/b00qszch`

Castro, R. and Barbosa, H. (2000). A genetic algorithm for multi-objective structural optimization. Tech. Rep., Universidade Federal do Rio de Janeiro and Laboratorio Nacional de Computacao Ceintifica. Year of release is unknown.

Coello, C.A., Rudnick, M. and Christiansen, A.D. (1994). Using genetic algorithms for optimal design of trusses. *IEEE*, pp. 88–94.

Coley, D.A. (1999). *An Introduction to Genetic Algorithms for Scientists and Engineers*. World Scientific Publishing Co. Pte. Ltd.

Cook, R., Malkus, D., Plesha, E. and Witt, R. (2002). *Concepts and Applications of Finite Element Analysis*. John Wiley and Sons. Inc.

Diez-Serrano, J. and Marais, P. (2005). Supporting structures. In: *The Plannin, Design and Construction of Overhead Power Lines*, chap. 20. Crown Publications cc. Book authors are Bisnath, S., Britten, A.C., Cretchley, D.H., Muftic, D., Pillay, T. and Vajeth, R.

Enevoldsen, I. and Sorensen, J.D. (1994). Reliability-based optimisation in structural engineering. *Structural Safety*, pp. 169–196.

Erbatur, F., Hasancebi, O., Tütüncü, I. and Kilic, H. (2000). Optimal design of planar and space structures with genetic algorithms. *Computers and Structures*, pp. 209–224.

Falakian, A. and Mousavi, S. (2011). Hybrid genetic algorithm for structural optimization. *Journal of Basic and Applied Scientific Research*, pp. 256–261.

Flager, F., Soremekun, G., Shea, K., Fischer, M. and Haymaker, J. (2011 October). Fully condtrained design: A scalable method for discrete member sizing optimization of steel frames structures. Tech. Rep., Stanford University. Center for Integrated Facility Engineering.

Gahsemi, M., Hinton, E. and Wood, R. (1999). Optimization of trusses using genetic algorithms for discrete and continuous variables. *Engineering Computations*, pp. 272–303.

Galante, M. (1996). Genetic algorithms as an approach to optimize real world trusses. *Internaltion Journal for Numerical Methods in Engineering, Vol 39*, pp. 361–382.

Gen, M. and Cheng, R. (1996). A survey of penalty techniques in genetic algorithms. *Evolutionary Computation*, pp. 804–809.

Goldberg, D.E. (1989). *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison Wesley Longman, Inc.

Goldberg, D.E. and Samtani, M.P. (1986). Engineering optimization via genetic algorithm. *Ninth Conference on Electronic Computation*, pp. 471–482.

Groenwold, A.A., Stander, N. and Snyman, J.A. (1999). A regional genetic algorithm for the discrete optimal design of truss structures. *International Journal for Numerical Methods in Engineering*, pp. 749–766.

Haftka, R.T. and Gürdal, Z. (1992). *Elements of Structural Optimization*. Kluwer Academic Publishers.

Hultman, M. (2010). Weight optimization of steel trusses by a genetic algorithm. Tech. Rep., Lund Institute of Technology. Sweden.

Jaber, A.Q., Hidehiko, Y. and Ramli, R. (2006). Machine learning in production sstems design using genetic algorithms. *International Journal of Computational Intelligence, Vol. 4, Number 1*, pp. 72–79.

Konak, A., Coit, D. and Smith, A. (2006). Mutli-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering and System Safety 91*, pp. 992–1007.

Krishnamoorthy, C.S., Venkatesh, P.P. and Sudarshan, R. (2002). Object oriented framework for genetic algorithms with application to space truss optimization. *Journal of Computing in Civil Engineering*, pp. 66–75.

Makhura, N. (2010 July). Final environmental impact and draft emp for proposed construction of 1x275kv from glockner-kookfontein substations and upgrade of kookfontein. Tech. Rep., Baagi Environmental Consultants.

Montgomary, D.C. and Runger, G.C. (2007). *Applied Statistics and Probability for Engineers*. John Wiley and Sons, Inc.

Muftic, D., Marais, P. and Mtolo, D. (2005). Introduction to design optimisation. In: *The Plannin, Design and Construction of Overhead Power Lines*, chap. 10. Crown Publications cc. Book authors are Bisnath, S., Britten, A.C., Cretchley, D.H., Muftic, D., Pillay, T. and Vajeth, R.

Nanakorn, P. and Meesomklin, K. (2001). An adaptive penalty funciton in genetic algorithms for structural design optimization. *Computers and Structures*, pp. 2527–2539.

Osyczka, A. and Kundu, S. (1995). A new method to solce generalized multicriteria optimization problems using the simple genetic algorithm. *Structural Optimization, Vol. 10*, pp. 94–99.

PLS (2011). *Tower Version 11.1 User's Manual.* Power Line Systems Inc.

Raj, R.P. and Kalyanaraman, V. (2005). Ga based optimal design of steel truss bridge. In: *6th World Congress of Structural and Multidisciplinary Optimization.*

Rajeev, S. and Krishnamoorthy, C.S. (1992). Discrete optimization of structures using genetic algorithms. *Journal of Structural Engineering, Vol. 118, No. 5*, pp. 1233–1250.

Rao, S.S. (2009). *Engineering Optimization - Theory and Practice, Fourth Addition.* John Wiley and Sons, Inc.

Rothlauf, F. (2011). *Design of Modern Heuristics, Principles and Application.* Springer.

SAISC (2008). *South African Steel Construction Handbook - Red Book.* Southern African Institute of Steel Construction.

SANS (2005). South african national standards 10162. Tech. Rep. 1, Standards South Africa.

Sivakumar, P., Rajaraman, A., Knight, G.M.S. and Ramachandramurthy, D. (2004). Object-oriented optimization approach using genetic algorithms for lattice towers. *Journal of Computing in Civil Engineering*, pp. 162–171.

Smith, A.E. and Coit, D.W. (1995). Penalty functions. In: *Handbook of Evolutionary Computation*, chap. Section C 5.2. Oxford University Press and Institute of Physics Publishing. Book authors are Beack, T., Fogel, D. and Michalewicz, Z.

Spillers, W. and MacBain, K.M. (2009). *Structural Optimization.* Springer Science and Business Media.

Togan, V. and Daloglu, A.T. (2008). An improved genetic algorithm with initial population strategy and self-adaptive member grouping. *Computers and Structures*, pp. 1204–1218.

Toropov, V.V. and Mahfouz, S.Y. (2001). Design optimization of structural steelwork using a genetic algorithm, fem and a system of design rules. *Engineering Computations, Vol. 18 Iss:3*, pp. 437–459.

Venter, J.C. (2012). Web page name: What types of genome maps are there? *Name of website: Genome News Network.*

Wagner (2000 June). Number of stars in the milky way. The Physics Factbook An Encyclopedia of Scientific Essays. Date accessed 13 June 2013.
Available at: `http://www.iol.co.za/news/south-africa/universities-running-on-empty-1.411745`

Yeniay, Ö. (2005). Penalty function methods for constrained optimization with genetic algorithms. *Mathematical and Computational Applications, Vol. 10*, pp. 45–56.

Appendices

17.1 Test Functions/Aritificial Landscapes

De Jong's Function 1

$$f_1(x_i) = \sum_{i=1}^{3} x_i^2 \tag{.0.1}$$

with $-5.12 \leqslant x_i \leqslant 5.12$.

De Jong's Function 2

$$f_2(x_i) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2 \tag{.0.2}$$

with $-2.048 \leqslant x_i \leqslant 2.048$.

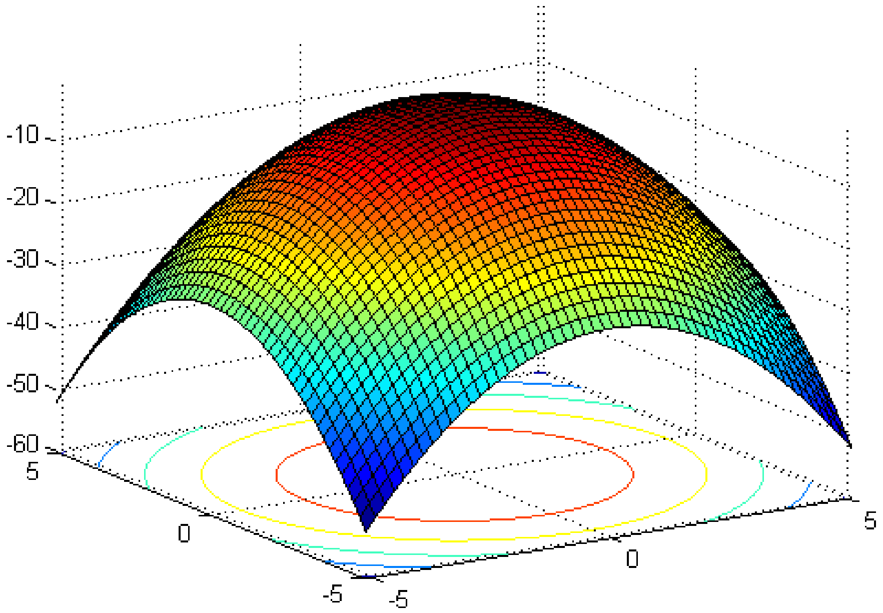


Figure 1: De Jong's Function 1 in 2D

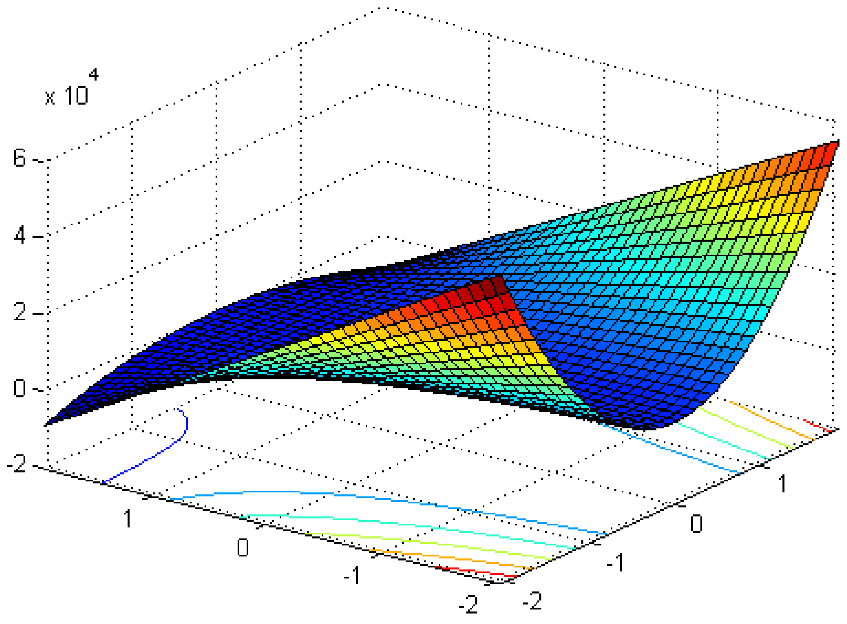


Figure 2: De Jong's Function 2 in 2D

De Jong's Function 3

$$f_3(x_i) = \sum_{i=1}^{5} \text{integer}(x_i) \tag{.0.3}$$

with $-5.12 \leqslant x_i \leqslant 5.12$.

De Jong's Function 4

$$f_4(x_i) = \sum_{i=1}^{30} i x_i^4 + \text{Gauss}(0,1) \tag{.0.4}$$

with $-1.28 \leqslant x_i \leqslant 1.28$.

De Jong's Function 5

$$f_5(x_i) = 0.002 + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2} (x_i - a_{ij})^6} \tag{.0.5}$$

with $-65.536 \leqslant x_i \leqslant 65.536$.

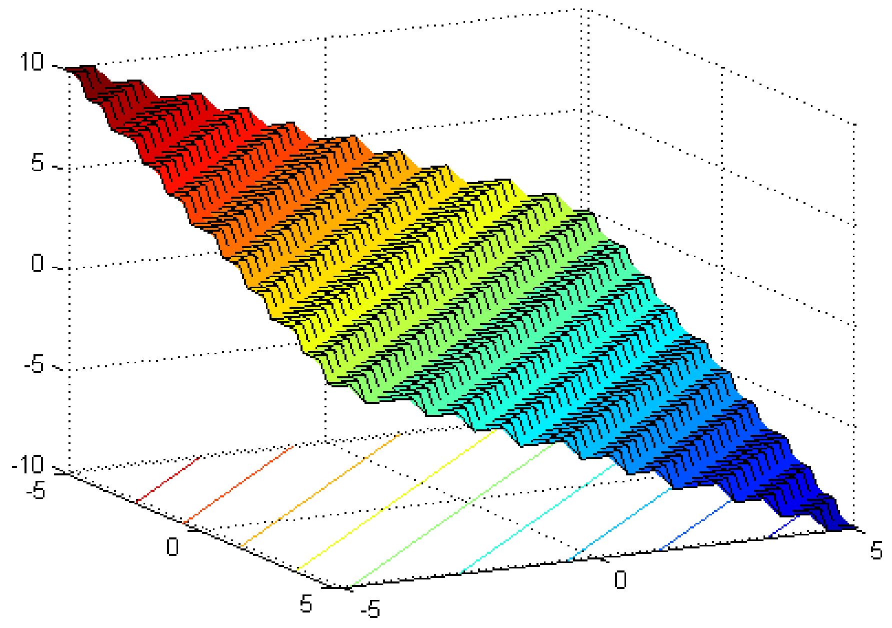


Figure 3: De Jong's Function 3 in 2D

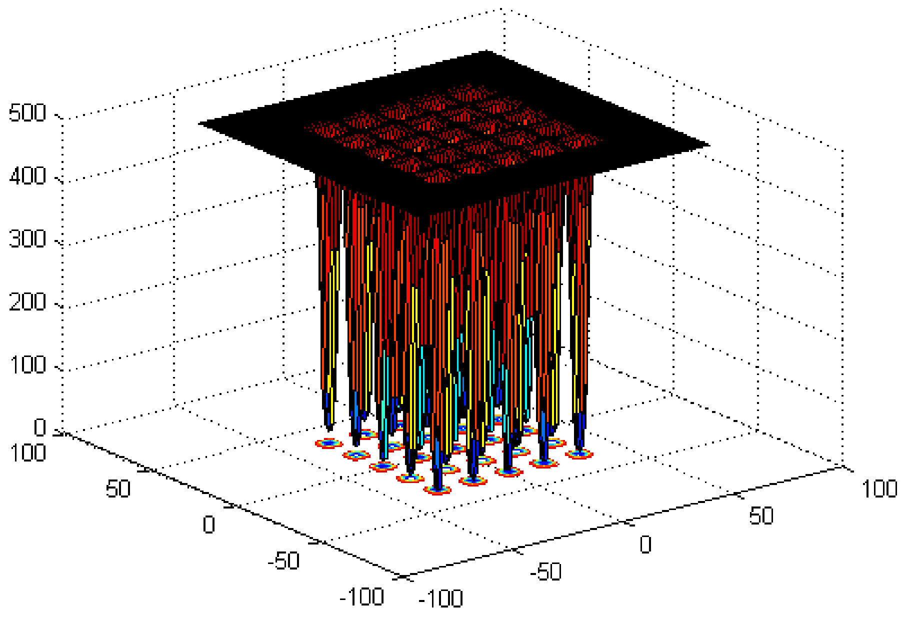


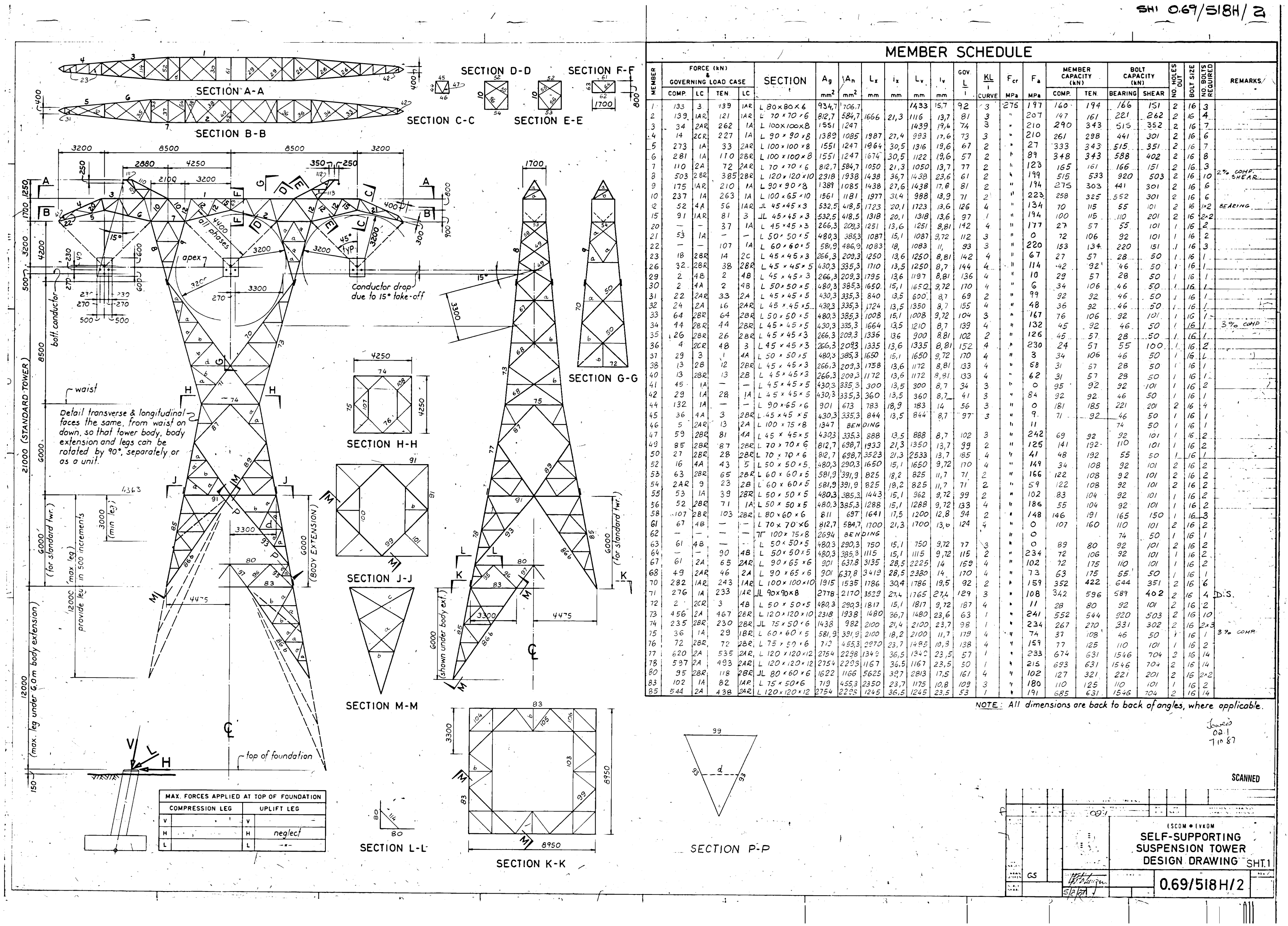
Figure 4: De Jong's Function 5 in 2D

17.2 List of Possible Errors

The list below are common errors to check for:

- Incorrect file paths
- Empty text fields that should have values
- The input section list has incorrect units
  - Area is given in x10^3 mm
- 'Error of empty string' means that an essential value in the input sheet is empty, i.e. element number missing
- The structure is unstable, refer to section 8.7
- Check the loading directions and structure orientation
- Structure might contain planar nodes, refer to section 15.2

17.3 Eskom Tower Drawings



MEMBER SCHEDULE

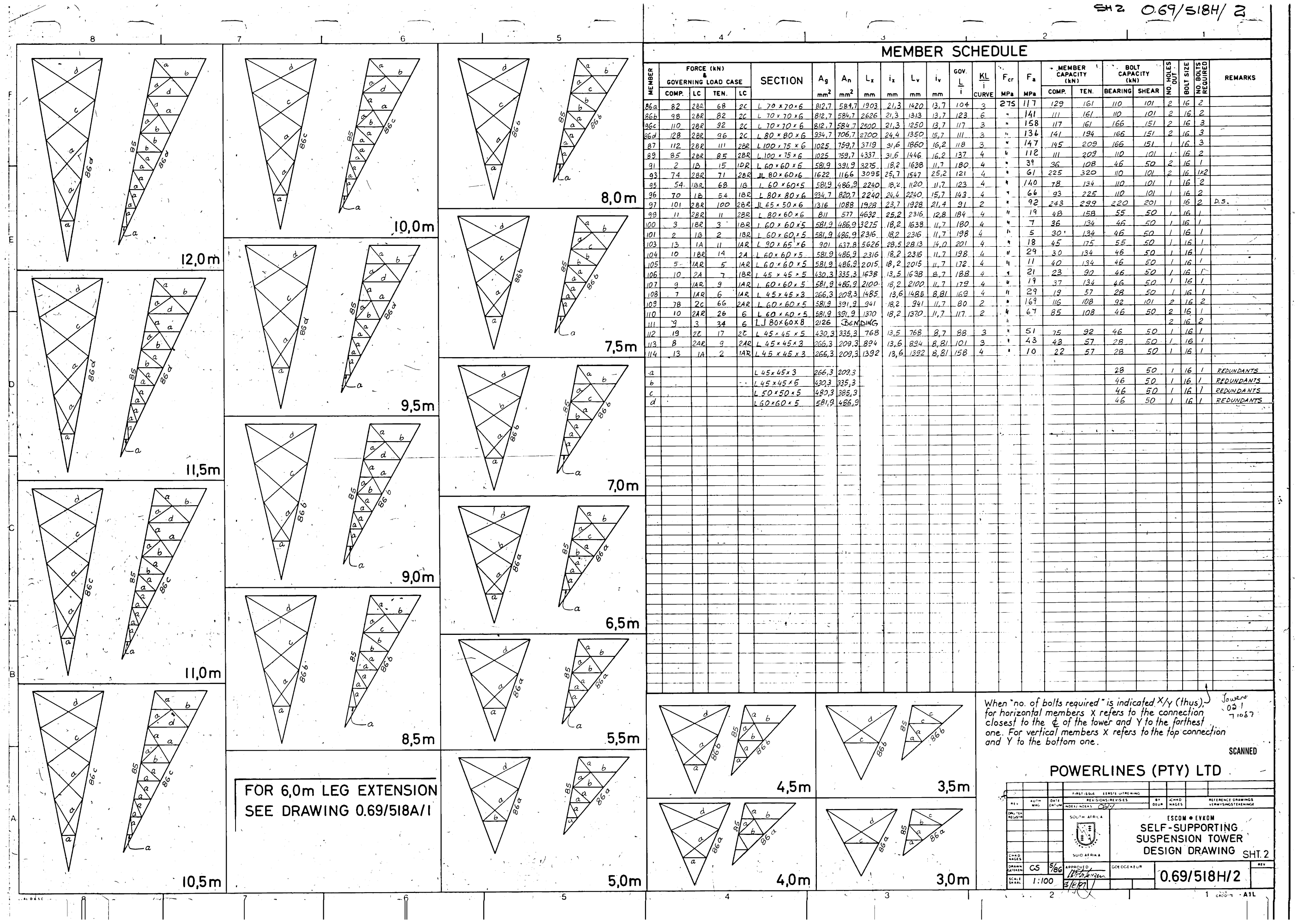
MEMBER	FORCE (kN) & GOVERNING LOAD CASE				SECTION	A_g	A_n	L_x	i_x	L_v	i_v	GOV. $\frac{L}{i}$	$\frac{KL}{i}$	F_{cr}	F_a	MEMBER CAPACITY (kN)		BOLT CAPACITY (kN)		NO. HOLES OUT	BOLT SIZE	NO. BOLTS REQUIRED	REMARKS
	COMP.	LC	TEN.	LC		mm^2	mm^2	mm	mm	mm	mm		CURVE	MPa	MPa	COMP.	TEN.	BEARING	SHEAR				
1	133	3	139	1AR	L 80×80×6	934,7	706,7			1433	15,7	92	3	275	197	160	194	166	151	2	16	3	
2	139	1AR	121	1AR	L 70×70×6	812,7	584,7	1666	21,3	1116	13,7	81	3	"	207	147	161	221	262	2	16	4	
3	34	2AR	262	1A	L 100×100×8	1551	1247			1439	19,6	74	3	"	210	290	343	515	352	2	16	7	
4	14	2CR	227	1A	L 90×90×8	1389	1085	1987	27,4	993	17,6	73	3	"	210	261	298	441	301	2	16	6	
5	273	1A	33	2AR	L 100×100×8	1551	1247	1964	30,5	1316	19,6	67	2	"	27	333	343	515	351	2	16	7	
6	281	1A	110	2BR	L 100×100×8	1551	1247	1674	30,5	1122	19,6	57	2	"	89	348	343	588	402	2	16	8	
7	110	2A	72	2AR	L 70×70×6	812,7	584,7	1050	21,3	1050	13,7	77	2	"	123	165	161	166	151	2	16	3	
8	503	2BR	385	2BR	L 120×120×10	2318	1938	1438	36,7	1438	23,6	61	2	"	199	515	533	920	503	2	16	10	2% COMP. SHEAR
9	175	1AR	210	1A	L 90×90×8	1389	1085	1438	27,6	1438	17,8	81	2	"	194	275	303	441	301	2	16	6	
10	237	1A	263	1A	L 100×65×10	1561	1181	1977	31,4	988	13,9	71	2	"	223	258	325	552	301	2	16	6	
12	52	4A	56	1AR	JL 45×45×3	532,5	418,5	1723	20,1	1723	13,6	126	4	"	134	70	115	55	101	2	16	1×2	BEARING
15	91	1AR	81	3	JL 45×45×3	532,5	418,5	1318	20,1	1318	13,6	97	1	"	194	100	115	110	201	2	16	2×2	
20	—	—	37	1A	L 45×45×3	266,3	209,3	1251	13,6	1251	8,81	142	4	"	177	27	57	55	101	1	16	2	
21	53	1A	—	—	L 50×50×5	480,3	385,3	1087	15,1	1087	9,72	112	3	"	0	72	106	92	101	1	16	2	
22	—	—	107	1A	L 60×60×5	581,9	486,9	1083	18,	1083	11,	93	3	"	220	153	134	220	151	1	16	3	
23	18	2BR	14	2C	L 45×45×3	266,3	209,3	1250	13,6	1250	8,81	142	4	"	67	27	57	28	50	1	16	1	
26	32	2BR	38	2BR	L 45×45×5	430,3	335,3	1710	13,5	1250	8,7	144	4	"	114	42	92	46	50	1	16	1	
29	2	4B	2	4B	L 45×45×3	266,3	209,3	1795	13,6	1197	8,81	136	4	"	10	29	57	28	50	1	16	1	
30	2	4A	2	4B	L 50×50×5	480,3	385,3	1650	15,1	1650	9,72	170	4	"	6	34	106	46	50	1	16	1	
31	22	2AR	33	2A	L 45×45×5	430,3	335,3	840	13,5	600	8,7	69	2	"	99	92	92	46	50	1	16	1	
32	24	2A	16	2AR	L 45×45×5	430,3	335,3	1724	13,5	1350	8,7	155	4	"	48	36	92	46	50	1	16	1	
33	64	2BR	64	2BR	L 50×50×5	480,3	385,3	1008	15,1	1008	9,72	104	3	"	167	76	106	92	101	1	16	1	
34	44	2BR	44	2BR	L 45×45×5	430,3	335,3	1664	13,5	1210	8,7	139	4	"	132	45	92	46	50	1	16	1	3% COMP
35	26	2BR	26	2BR	L 45×45×3	266,3	209,3	1336	13,6	900	8,81	102	2	"	126	45	57	28	50	1	16	1	
36	4	2CR	48	3	L 45×45×3	266,3	209,3	1335	13,6	1335	8,81	152	4	"	230	24	57	55	100	1	16	2	
37	29	3	1	4A	L 50×50×5	480,3	385,3	1650	15,1	1650	9,72	170	4	"	3	34	106	46	50	1	16	1	
38	13	2B	12	2BR	L 45×45×3	266,3	209,3	1758	13,6	1172	8,81	133	4	"	58	31	57	28	50	1	16	1	
40	13	2BR	13	2B	L 45×45×3	266,3	209,3	1172	13,6	1172	8,81	133	4	"	62	31	57	28	50	1	16	1	
41	45	1A	—	—	L 45×45×5	430,3	335,3	300	13,5	300	8,7	34	3	"	0	95	92	92	101	1	16	2	
42	29	1A	28	1A	L 45×45×5	430,3	335,3	360	13,5	360	8,7	41	3	"	84	92	92	46	50	1	16	1	
44	132	1A	—	—	L 90×65×6	901	613	783	18,9	783	14	56	3	"	0	181	185	221	201	2	16	4	
45	36	4A	3	2BR	L 45×45×5	430,3	335,3	844	13,5	844	8,7	97	3	"	9	71	92	46	50	1	16	1	
46	5	2AR	13	2A	L 100×75×8	1347	BENDING							"	11			74	50	1	16	1	
47	59	2BR	81	4A	L 45×45×5	430,3	335,3	888	13,5	888	8,7	102	3	"	242	69	92	92	101	1	16	2	
49	85	2BR	87	2BR	L 70×70×6	812,7	698,7	1933	21,3	1350	13,7	99	2	"	125	141	192	110	101	1	16	2	
50	27	2BR	28	2BR	L 70×70×6	812,7	698,7	3523	21,3	2533	13,7	185	4	"	41	48	192	55	50	1	16	1	
52	16	4A	43	5	L 50×50×5	480,3	290,3	1650	15,1	1650	9,72	170	4	"	149	34	108	92	101	2	16	2	
53	63	2BR	65	2BR	L 60×60×5	581,9	391,9	825	18,2	825	11,7	71	2	"	166	122	108	92	101	2	16	2	
54	2AR	9	23	2B	L 60×60×5	581,9	391,9	825	18,2	825	11,7	71	2	"	59	122	108	92	101	2	16	2	
55	53	1A	39	2BR	L 50×50×5	480,3	385,3	1443	15,1	962	9,72	99	2	"	102	83	104	92	101	1	16	2	
56	52	2BR	71	1A	L 50×50×5	480,3	385,3	1288	15,1	1288	9,72	133	4	"	185	55	104	92	101	1	16	2	
58	107	2BR	103	2BR	L 80×60×6	811	697	1641	17,5	1200	12,8	94	2	"	148	146	191	165	150	1	16	3	
61	67	4B	—	—	L 70×70×6	812,7	584,7	1700	21,3	1700	13,6	124	4	"	0	107	160	110	101	2	16	2	
62	—	—	—	—	ℸΓ 100×75×8	2694	BENDING							"	0			74	50	1	16	1	
63	61	4B	—	—	L 50×50×5	480,3	290,3	750	15,1	750	9,72	77	3	"	0	89	80	92	101	2	16	2	
64	—	—	90	4B	L 50×50×5	480,3	385,3	1115	15,1	1115	9,72	115	2	"	234	72	106	92	101	1	16	2	
67	61	2A	65	2AR	L 90×65×6	901	637,8	3135	28,5	2225	14	159	4	"	102	72	175	110	101	1	16	2	
68	49	2AR	46	2A	L 90×65×6	901	637,8	3419	28,5	2380	14	170	4	"	73	63	175	55	50	1	16	1	
70	282	1AR	243	1AR	L 100×100×10	1915	1535	1786	30,4	1786	19,5	92	2	"	159	352	422	644	351	2	16	6	
71	276	1A	233	1AR	JL 90×90×8	2778	2170	3529	27,4	1765	27,4	129	3	"	108	342	596	589	402	2	16	4	D.S.
72	2	2CR	3	4B	L 50×50×5	480,3	290,3	1817	15,1	1817	9,72	187	4	"	11	28	80	92	101	2	16	2	
73	456	2A	467	2BR	L 120×120×10	2318	1938	1480	36,7	1480	23,6	63	1	"	241	552	544	920	503	2	16	10	
74	235	2BR	230	2BR	JL 75×50×6	1438	982	2100	21,4	2100	23,7	98	1	"	234	267	270	331	302	2	16	2×3	
75	36	1A	29	1BR	L 60×60×5	581,9	391,9	2100	18,2	2100	11,7	179	4	"	74	37	108	46	50	1	16	1	3% COMP.
76	72	2BR	72	2BR	L 75×50×6	713	455,3	2970	23,7	1485	10,3	138	4	"	159	77	125	110	101	1	16	2	
77	620	2A	535	2AR	L 120×120×12	2754	2298	1349	36,5	1349	23,5	57	1	"	233	674	631	1546	704	2	16	14	
78	597	2A	493	2AR	L 120×120×12	2754	2298	1167	36,5	1167	23,5	50	1	"	215	693	631	1546	704	2	16	14	
80	95	2BR	118	2BR	JL 80×60×6	1622	1166	5625	39,7	2813	17,5	161	4	"	102	127	321	221	201	2	16	2×2	
83	102	1A	82	1AR	L 75×50×6	719	455,3	2350	23,7	1175	10,8	109	3	"	180	110	125	110	101	1	16	2	
85	544	2A	438	2AR	L 120×120×12	2754	2298	1245	36,5	1245	23,5	53	1	"	191	685	631	1546	704	2	16	14	

NOTE: All dimensions are back to back of angles, where applicable.

MAX. FORCES APPLIED AT TOP OF FOUNDATION			
COMPRESSION LEG		UPLIFT LEG	
V		V	
H		H	neglect
L		L	—"—

ESCOM • EVKOM
SELF-SUPPORTING SUSPENSION TOWER DESIGN DRAWING SHT.1
0.69/518H/2
GS
5/2/87

Figure 5: Tower Drawing Main



FOR 6,0m LEG EXTENSION
SEE DRAWING 0.69/518A/1

MEMBER SCHEDULE

MEMBER	FORCE (kN) & GOVERNING LOAD CASE				SECTION	A_g	A_n	L_x	i_x	L_v	i_v	GOV. $\frac{L}{i}$	$\frac{KL}{i}$	F_{cr}	F_a	MEMBER CAPACITY (kN)		BOLT CAPACITY (kN)		NO. HOLES OUT	BOLT SIZE	NO. BOLTS REQUIRED	REMARKS
	COMP.	LC	TEN.	LC		mm²	mm²	mm	mm	mm	mm		CURVE	MPa	MPa	COMP.	TEN.	BEARING	SHEAR				
86a	82	2BR	68	2C	L 70×70×6	812,7	584,7	1903	21,3	1420	13,7	104	3	275	117	129	161	110	101	2	16	2	
86b	98	2BR	82	2C	L 70×70×6	812,7	584,7	2626	21,3	1313	13,7	123	6	"	141	111	161	110	101	2	16	2	
86c	110	2BR	92	2C	L 70×70×6	812,7	584,7	2500	21,3	1250	13,7	117	3	"	158	117	161	166	151	2	16	3	
86d	128	2BR	96	2C	L 80×80×6	934,7	706,7	2700	24,4	1350	15,7	111	3	"	136	141	194	166	151	2	16	3	
87	112	2BR	111	2BR	L 100×75×6	1025	759,7	3719	31,6	1860	16,2	118	3	"	147	145	209	166	151	1	16	3	
89	85	2BR	85	2BR	L 100×75×6	1025	759,7	4337	31,6	1446	16,2	137	4	"	112	111	209	110	101	1	16	2	
91	2	1B	15	1DR	L 60×60×5	581,9	391,9	3275	18,2	1638	11,7	180	4	"	39	36	108	46	50	2	16	1	
93	74	2BR	71	2BR	JL 80×60×6	1622	1166	3095	25,7	1547	25,2	121	4	"	61	225	320	110	101	2	16	1x2	
95	54	1BR	68	1B	L 60×60×5	581,9	486,9	2240	18,2	1120	11,7	123	4	"	140	78	134	110	101	1	16	2	
96	70	1B	54	1BR	L 80×80×6	934,7	820,7	2240	24,4	2240	15,7	143	4	"	66	93	225	110	101	1	16	2	
97	101	2BR	100	2BR	JL 65×50×6	1316	1088	1928	23,7	1928	21,4	91	2	"	92	243	299	220	201	1	16	2	D.S.
99	11	2BR	11	2BR	L 80×60×6	811	577	4632	25,2	2316	12,8	184	4	"	19	48	158	55	50	1	16	1	
100	3	1BR	3	1BR	L 60×60×5	581,9	486,9	3275	18,2	1638	11,7	180	4	"	7	36	134	46	50	1	16	1	
101	2	1B	2	1BR	L 60×60×5	581,9	486,9	2316	18,2	2316	11,7	198	4	"	5	30	134	46	50	1	16	1	
103	13	1A	11	1AR	L 90×65×6	901	637,8	5626	28,5	2813	14,0	201	4	"	18	45	175	55	50	1	16	1	
104	10	1BR	14	2A	L 60×60×5	581,9	486,9	2316	18,2	2316	11,7	198	4	"	29	30	134	46	50	1	16	1	
105	5	1AR	5	1AR	L 60×60×5	581,9	486,9	2015	18,2	2015	11,7	172	4	"	11	40	134	46	50	1	16	1	
106	10	2A	7	1BR	L 45×45×5	430,3	335,3	1638	13,5	1638	8,7	188	4	"	21	23	90	46	50	1	16	1	
107	9	1AR	9	1AR	L 60×60×5	581,9	486,9	2100	18,2	2100	11,7	179	4	"	19	37	134	46	50	1	16	1	
108	7	1AR	6	1AR	L 45×45×3	266,3	209,3	1485	13,6	1485	8,81	169	4	"	29	19	57	28	50	1	16	1	
109	78	2C	66	2AR	L 60×60×5	581,9	391,9	941	18,2	941	11,7	80	2	"	169	116	108	92	101	2	16	2	
110	10	2AR	26	6	L 60×60×5	581,9	391,9	1370	18,2	1370	11,7	117	2	"	67	85	108	46	50	2	16	1	
111	9	3	34	6	LJ 80×60×8	2126	BENDING							"						2	16	2	
112	19	2C	17	2C	L 45×45×5	430,3	335,3	768	13,5	768	8,7	88	3	"	51	75	92	46	50	1	16	1	
113	8	2AR	9	2AR	L 45×45×3	266,3	209,3	894	13,6	894	8,81	101	3	"	43	43	57	28	50	1	16	1	
114	13	1A	2	1AR	L 45×45×3	266,3	209,3	1392	13,6	1392	8,81	158	4	"	10	22	57	28	50	1	16	1	
a					L 45×45×3	266,3	209,3											28	50	1	16	1	REDUNDANTS
b					L 45×45×5	430,3	335,3											46	50	1	16	1	REDUNDANTS
c					L 50×50×5	480,3	385,3											46	50	1	16	1	REDUNDANTS
d					L 60×60×5	581,9	486,9											46	50	1	16	1	REDUNDANTS

When "no. of bolts required" is indicated X/Y (thus), for horizontal members X refers to the connection closest to the ℄ of the tower and Y to the farthest one. For vertical members X refers to the top connection and Y to the bottom one.

Towers 021 71087

SCANNED

POWERLINES (PTY) LTD

REV	AUTH MAG	DATE DATUM	FIRST ISSUE EERSTE UITREIKING / REVISIONS/REVISIES / INDEX/INDEKS 021	BY DEUR	CHKD NAGES	REFERENCE DRAWINGS VERWYSINGSTEKENINGE
DRG TEK REGISTR						
CHKD NAGES						
DRAWN GETEKEN	GS	5/86	APPROVED [illegible] 3/8/87	GOEDGEKEUR		
SCALE SKAAL	1:100					

SOUTH AFRICA
ESCOM • EVKOM
SUID AFRIKA

ESCOM • EVKOM
SELF-SUPPORTING
SUSPENSION TOWER
DESIGN DRAWING SHT. 2

0.69/518H/2

REV

SIZE GROOTTE A1L

Figure 6: Tower Drawing Legs

17.4 Eskom Design Data

Table 1: Tower Nodal Coordinates

Node	x **(m)**	y **(m)**	z **(m)**	**Node**	x **(m)**	y **(m)**	z **(m)**
1	7.225	0	0	46	7.225	0	8.95
2	8.12	1.2	0.235	47	7.46	1.2	0.895
3	9.015	2.4	0.47	48	7.695	2.4	1.79
4	9.91	3.6	0.705	49	7.93	3.6	2.685
5	10.805	4.8	0.94	50	8.165	4.8	3.58
6	11.7	6	1.175	51	15.425	25.55	4.475
7	7.46	1.2	0.235	52	8.4	6	4.475
8	7.695	2.4	0.47	53	7.46	1.2	8.055
9	7.93	3.6	0.705	54	7.695	2.4	7.16
10	8.165	4.8	0.94	55	7.93	3.6	6.265
11	8.4	6	1.175	56	8.165	4.8	5.37
12	15.28	1.2	0.235	57	8.165	4.8	1.93
13	14.385	2.4	0.47	58	8.165	4.8	7.02
14	13.49	3.6	0.705	59	15.94	1.2	0.895
15	12.595	4.8	0.94	60	15.705	2.4	1.79
16	15.94	1.2	0.235	61	15.47	3.6	2.685
17	15.705	2.4	0.47	62	15.235	4.8	3.58
18	15.47	3.6	0.705	63	15	6	4.475
19	15.235	4.8	0.94	64	15.94	1.2	8.055
20	15	6	1.175	65	15.705	2.4	7.16
21	16.175	0	0	66	15.47	3.6	6.265
22	9.155	4.8	0.94	67	15.235	4.8	5.37
23	14.245	4.8	0.94	68	15.235	4.8	1.93
24	8.12	1.2	8.715	69	15.235	4.8	7.02
25	9.015	2.4	8.48	70	8.635	7.2	1.41
26	9.91	3.6	8.245	71	8.87	8.4	1.645
27	10.805	4.8	8.01	72	9.105	9.6	1.88
28	11.7	6	7.775	73	9.34	10.8	2.115
29	7.46	1.2	8.715	74	9.575	12	2.35
30	7.695	2.4	8.48	75	14.765	7.2	1.41
31	7.93	3.6	8.245	76	14.53	8.4	1.645
32	8.165	4.8	8.01	77	14.295	9.6	1.88
33	8.4	6	7.775	78	14.06	10.8	2.115
34	15.28	1.2	8.715	79	13.825	12	2.35
35	14.385	2.4	8.48	80	8.635	7.2	7.54
36	13.49	3.6	8.245	81	8.87	8.4	7.305
37	12.595	4.8	8.01	82	9.105	9.6	7.07
38	15.94	1.2	8.715	83	9.34	10.8	6.835
39	15.705	2.4	8.48	84	9.575	12	6.6
40	15.47	3.6	8.245	85	14.765	7.2	7.54
41	15.235	4.8	8.01	86	14.53	8.4	7.305
42	15	6	7.775	87	14.295	9.6	7.07
43	16.175	0	8.95	88	14.06	10.8	6.835
44	9.155	4.8	7.99	89	13.825	12	6.6
45	14.245	4.8	7.99	90	10.0625	6	1.175

Table 2: Tower Nodal Coordinates

Node	x **(m)**	y **(m)**	z **(m)**	**Node**	x **(m)**	y **(m)**	z **(m)**
91	13.3375	6	1.175	136	9.22525	13.399	2.490451
92	10.0625	6	7.775	137	8.8755	14.798	2.630902
93	13.3375	6	7.775	138	8.51875	16.225	2.774163
94	8.4	6	2.8375	139	8.1625	17.65	2.917224
95	8.4	6	6.1125	140	7.80625	19.075	3.060285
96	15	6	2.8375	141	7.45	20.5	3.203346
97	15	6	6.1125	142	9.22525	13.399	6.459549
98	10.805	7.2	1.41	143	8.8755	14.798	6.319098
99	9.91	8.4	1.645	144	8.51875	16.225	6.175837
100	12.595	7.2	1.41	145	8.1625	17.65	6.032776
101	13.49	8.4	1.645	146	7.80625	19.075	5.889715
102	10.805	7.2	7.54	147	7.45	20.5	5.746654
103	9.91	8.4	7.305	148	9.22525	13.399	4.475
104	12.595	7.2	7.54	149	8.51875	16.225	4.475
105	13.49	8.4	7.305	150	7.80625	19.075	4.475
106	8.635	7.2	3.58	151	14.17475	13.399	2.490451
107	8.87	8.4	2.685	152	14.5245	14.798	2.630902
108	8.635	7.2	5.37	153	14.88125	16.225	2.774163
109	8.87	8.4	6.265	154	15.2375	17.65	2.917224
110	14.765	7.2	3.58	155	15.59375	19.075	3.060285
111	14.53	8.4	2.685	156	15.95	20.5	3.203346
112	14.765	7.2	5.37	157	14.17475	13.399	6.459549
113	14.53	8.4	6.265	158	14.5245	14.798	6.319098
114	12.9775	10.8	2.16	159	14.88125	16.225	6.175837
115	10.4225	10.8	2.16	160	15.2375	17.65	6.032776
116	12.9775	10.8	6.79	161	15.59375	19.075	5.889715
117	10.4225	10.8	6.79	162	15.95	20.5	5.746654
118	14.06	10.8	5.7525	163	14.17475	13.399	4.475
119	14.06	10.8	3.1975	164	14.88125	16.225	4.475
120	9.34	10.8	5.7525	165	15.59375	19.075	4.475
121	9.34	10.8	3.1975	166	12.77575	13.399	2.490451
122	11.7	12	2.35	167	11.7265	14.798	2.630902
123	11.7	12	6.6	168	10.65625	16.225	2.774163
124	9.575	12	4.475	169	9.5875	17.65	2.917224
125	13.825	12	4.475	170	8.51875	19.075	3.060285
126	10.0625	6	2.8375	171	10.62425	13.399	2.490451
127	10.0625	6	6.1125	172	12.74375	16.225	2.774163
128	13.3375	6	2.8375	173	13.8125	17.65	2.917224
129	13.3375	6	6.1125	174	14.88125	19.075	3.060285
130	10.0625	6	4.475	175	12.77575	13.399	6.459549
131	13.3375	6	4.475	176	11.7265	14.798	6.319098
132	10.6375	12	3.4125	177	10.65625	16.225	6.175837
133	10.6375	12	5.5375	178	9.5875	17.65	6.032776
134	12.7625	12	3.4125	179	8.51875	19.075	5.889715
135	12.7625	12	5.5375	180	10.62425	13.399	6.459549

Table 3: Tower Nodal Coordinates

Node	x **(m)**	y **(m)**	z **(m)**	**Node**	x **(m)**	y **(m)**	z **(m)**
181	12.74375	16.225	6.175837	226	4.316	26.4	5.061519
182	13.8125	17.65	6.032776	227	6.025	26.4	5.219765
183	14.88125	19.075	5.889715	228	7.45	26.4	5.324
184	10.65625	16.225	4.475	229	8.2375	26.4	5.324
185	8.51875	19.075	4.475	230	9.025	26.4	5.324
186	12.74375	16.225	4.475	231	10.6	26.4	5.324
187	14.88125	19.075	4.475	232	11.7	26.4	5.324
188	11.7265	14.798	4.475	233	12.8	26.4	5.324
189	7.8	21.9	3.343898	234	14.375	26.4	5.324
190	8.15	23.3	3.484449	235	15.1625	26.4	5.324
191	8.5	24.7	3.625	236	15.95	26.4	5.324
192	7.1	21.9	3.343898	237	17.375	26.4	5.219765
193	6.75	23.3	3.484449	238	19.084	26.4	5.061519
194	6.4	24.7	3.625	239	20.3665	26.4	4.93897
195	16.3	21.9	3.343898	240	21.17616	26.00334	4.850647
196	16.65	23.3	3.484449	241	22.13616	25.54376	4.762323
197	17	24.7	3.625	242	23.09616	25.09709	4.674
198	15.6	21.9	3.343898	243	23.4	24.7	4.474
199	15.25	23.3	3.484449	244	0.303837	25.09709	4.274
200	14.9	24.7	3.625	245	1.263837	25.54376	4.185677
201	7.45	24.7	3.625	246	2.223837	26.00334	4.097353
202	15.95	24.7	3.625	247	3.0335	26.4	4.00903
203	7.8	21.9	5.606102	248	4.316	26.4	3.886481
204	8.15	23.3	5.465551	249	6.025	26.4	3.728235
205	8.5	24.7	5.325	250	7.45	26.4	3.624
206	7.1	21.9	5.606102	251	8.2375	26.4	3.624
207	6.75	23.3	5.465551	252	9.025	26.4	3.624
208	6.4	24.7	5.325	253	10.6	26.4	3.624
209	16.3	21.9	5.606102	254	11.7	26.4	3.624
210	16.65	23.3	5.465551	255	12.8	26.4	3.624
211	17	24.7	5.325	256	14.375	26.4	3.624
212	15.6	21.9	5.606102	257	15.1625	26.4	3.624
213	15.25	23.3	5.465551	258	15.95	26.4	3.624
214	14.9	24.7	5.325	259	17.375	26.4	3.728235
215	7.45	24.7	5.325	260	19.084	26.4	3.886481
216	15.95	24.7	5.325	261	20.3665	26.4	4.00903
217	6.4	24.7	4.475	262	21.17616	26.00334	4.097353
218	17	24.7	4.475	263	22.13616	25.54376	4.185677
219	7.8	21.9	4.475	264	23.09616	25.09709	4.274
220	15.6	21.9	4.475	265	0.38506	24.8125	4.674
221	0	24.7	4.474	266	0.987	25	4.768008
222	0.303837	25.09709	4.674	267	2.223837	25.365	4.850647
223	1.263837	25.54376	4.762323	268	2.9385	25.6	4.953882
224	2.223837	26.00334	4.850647	269	3.7805	25.39806	5.03408
225	3.0335	26.4	4.93897	270	4.6105	25.16891	5.113135

Table 4: Tower Nodal Coordinates

Node	x (m)	y (m)	z (m)	Node	x (m)	y (m)	z (m)
271	5.4405	24.93386	5.19219	316	18.58	27.67	4.475
272	9.212	24.92809	5.324	317	18.83	27.67	4.475
273	10.038	25.16891	5.324	318	17.975	27.025	4.927
274	11.225	25.49	5.324	319	17.025	27.025	4.023
275	11.7	25.6	5.324	320	17.975	27.025	4.023
276	12.175	25.49	5.324	321	6.375	27.025	4.927
277	13.362	25.16891	5.324	322	5.17	27.67	4.475
278	14.188	24.92809	5.324	323	4.82	27.67	4.475
279	17.9595	24.93386	5.19219	324	4.57	27.67	4.475
280	18.7895	25.16891	5.113135	325	5.425	27.025	4.927
281	19.6195	25.39806	5.03408	326	6.375	27.025	4.023
282	20.4615	25.6	4.953882	327	5.425	27.025	4.023
283	21.17616	25.365	4.850647	328	9.8125	26.4	4.475
284	22.413	25	4.768008	329	13.5875	26.4	4.475
285	23.01494	24.8125	4.674	330	6.710068	26.4	4.475
286	0.38506	24.8125	4.276	331	5.060176	26.4	4.475
287	0.987	25	4.181992	332	3.591226	26.4	4.475
288	2.223837	25.365	4.099353	333	16.68993	26.4	4.475
289	2.9385	25.6	3.996118	334	18.33982	26.4	4.475
290	3.7805	25.39806	3.91592	335	19.80877	26.4	4.475
291	4.6105	25.16891	3.836865	336	10.2185	25.209	4.475
292	5.4405	24.93386	3.75781	337	8.856	24.814	4.475
293	9.212	24.92809	3.626	338	13.1815	25.209	4.475
294	10.038	25.16891	3.626	339	14.544	24.814	4.475
295	11.225	25.49	3.626	340	17.47033	24.81476	4.475
296	11.7	25.6	3.626	341	18.90451	25.19829	4.475
297	12.175	25.49	3.626	342	5.92967	24.81498	4.475
298	13.362	25.16891	3.626	343	4.495491	25.19829	4.475
299	14.188	24.92809	3.626	344	21.69538	25.22688	4.475
300	17.9595	24.93386	3.75781	345	22.78736	24.88371	4.475
301	18.7895	25.16891	3.836865	346	1.704621	25.22688	4.475
302	19.6195	25.39806	3.91592	347	0.612637	24.88371	4.475
303	20.4615	25.6	3.996118	348	16.08273	21.2433	4.475
304	21.17616	25.365	4.099353	349	16.4871	22.64842	4.475
305	22.413	25	4.181992	350	16.84929	24.09714	4.475
306	23.01494	24.8125	4.276	351	7.317269	21.2433	4.475
307	8.5	24.7	4.475	352	6.912895	22.64842	4.475
308	14.9	24.7	4.475	353	6.550715	24.09714	4.475
309	11.7	25.6	4.475	354	8.334084	24.03634	4.475
310	8.2375	26.4	4.475	355	15.06592	24.03634	4.475
311	15.1625	26.4	4.475	356	6.925	25.55	4.475
312	20.4615	25.6	4.475	357	7.975	25.55	4.475
313	2.9385	25.6	4.475	358	16.475	25.55	4.475
314	17.025	27.025	4.927				
315	18.23	27.67	4.475				

Table 5: Tower Fixity

Node	**Fixity**
1	ALL_TRANSLATION
21	ALL_TRANSLATION
43	ALL_TRANSLATION
46	ALL_TRANSLATION

Table 6: Tower Element Definition

	Node 1	**Node 2**		**Node 1**	**Node 2**		**Node 1**	**Node 2**
1	1	7	46	29	30	91	48	49
2	7	8	47	30	31	92	49	50
3	8	9	48	31	32	93	50	52
4	9	10	49	32	33	94	46	53
5	10	11	50	46	24	95	53	54
6	1	2	51	24	25	96	54	55
7	2	3	52	25	26	97	55	56
8	3	4	53	26	27	98	56	52
9	4	5	54	27	28	99	7	47
10	5	6	55	43	34	100	8	48
11	7	2	56	34	35	101	9	49
12	8	3	57	35	36	102	10	57
13	9	4	58	36	37	103	57	50
14	10	22	59	37	28	104	94	52
15	22	5	60	43	38	105	11	94
16	90	6	61	38	39	106	53	29
17	11	90	62	39	40	107	54	30
18	21	12	63	40	41	108	55	31
19	12	13	64	41	42	109	56	58
20	13	14	65	29	24	110	58	32
21	14	15	66	30	25	111	95	33
22	15	6	67	31	26	112	52	95
23	21	16	68	44	27	113	47	8
24	16	17	69	32	44	114	48	9
25	17	18	70	92	28	115	49	57
26	18	19	71	33	92	116	57	11
27	19	20	72	34	38	117	53	30
28	12	16	73	35	39	118	54	31
29	13	17	74	36	40	119	55	58
30	14	18	75	37	45	120	58	33
31	23	15	76	45	41	121	43	64
32	19	23	77	93	42	122	64	65
33	91	20	78	28	93	123	65	66
34	6	91	79	24	30	124	66	67
35	2	8	80	25	31	125	67	63
36	3	9	81	26	44	126	21	59
37	9	22	82	31	44	127	59	60
38	4	22	83	44	33	128	60	61
39	22	11	84	34	39	129	61	62
40	12	17	85	35	40	130	62	63
41	13	18	86	36	45	131	38	64
42	14	23	87	40	45	132	39	65
43	18	23	88	45	42	133	40	66
44	23	20	89	1	47	134	41	69
45	46	29	90	47	48	135	69	67

Table 7: Tower Element Definition

	Node 1	Node 2		Node 1	Node 2		Node 1	Node 2
136	97	63	181	63	128	226	109	82
137	42	97	182	128	6	227	28	102
138	59	16	183	6	126	228	102	103
139	60	17	184	126	52	229	103	82
140	61	18	185	126	11	230	28	104
141	62	68	186	127	33	231	104	105
142	68	19	187	129	42	232	105	87
143	96	20	188	128	20	233	63	112
144	63	96	189	129	131	234	112	113
145	64	39	190	131	63	235	113	87
146	65	40	191	131	128	236	63	110
147	66	69	192	127	130	237	110	111
148	69	42	193	130	52	238	111	77
149	40	69	194	130	126	239	85	112
150	9	57	195	11	70	240	86	113
151	31	58	196	70	71	241	110	75
152	59	17	197	71	72	242	111	76
153	60	18	198	72	73	243	75	100
154	61	68	199	73	74	244	76	101
155	18	68	200	33	80	245	98	70
156	68	20	201	80	81	246	99	71
157	63	36	202	81	82	247	70	106
158	28	66	203	82	83	248	71	107
159	66	34	204	83	84	249	108	80
160	36	64	205	42	85	250	109	81
161	64	34	206	85	86	251	80	102
162	6	61	207	86	87	252	81	103
163	63	14	208	87	88	253	104	85
164	14	59	209	88	89	254	105	86
165	61	12	210	20	75	255	104	93
166	12	59	211	75	76	256	92	102
167	52	4	212	76	77	257	94	106
168	6	49	213	77	78	258	95	108
169	4	47	214	78	79	259	91	100
170	49	2	215	6	100	260	90	98
171	2	47	216	100	101	261	96	110
172	28	55	217	101	77	262	97	112
173	52	26	218	6	98	263	93	85
174	26	53	219	98	99	264	92	80
175	55	24	220	99	72	265	95	80
176	24	53	221	52	106	266	94	70
177	28	127	222	106	107	267	90	70
178	127	52	223	107	72	268	91	75
179	28	129	224	52	108	269	96	75
180	129	63	225	108	109	270	97	85

Table 8: Tower Element Definition

	Node 1	**Node 2**		**Node 1**	**Node 2**		**Node 1**	**Node 2**
271	70	107	316	120	84	361	150	147
272	70	99	317	117	84	362	150	141
273	75	101	318	116	89	363	79	151
274	75	111	319	124	132	364	151	152
275	85	113	320	132	122	365	152	153
276	85	105	321	122	134	366	153	154
277	80	103	322	134	125	367	154	155
278	80	109	323	125	135	368	89	157
279	77	114	324	135	123	369	157	158
280	114	122	325	123	133	370	158	159
281	72	115	326	133	124	371	159	160
282	115	122	327	132	74	372	160	161
283	72	121	328	135	89	373	155	156
284	121	124	329	134	79	374	161	162
285	82	120	330	133	84	375	79	163
286	120	124	331	132	133	376	163	158
287	82	117	332	134	135	377	89	163
288	117	123	333	84	142	378	163	152
289	87	116	334	142	143	379	152	164
290	116	123	335	143	144	380	164	160
291	87	118	336	144	145	381	158	164
292	118	125	337	145	146	382	164	154
293	77	119	338	146	147	383	154	165
294	119	125	339	74	136	384	165	162
295	88	118	340	136	137	385	160	165
296	119	78	341	137	138	386	165	156
297	78	114	342	138	139	387	151	163
298	115	73	343	139	140	388	163	157
299	73	121	344	140	141	389	153	164
300	120	83	345	142	148	390	164	159
301	83	117	346	148	136	391	155	165
302	116	88	347	144	149	392	165	161
303	89	125	348	149	138	393	79	166
304	125	79	349	146	150	394	166	167
305	79	122	350	150	140	395	167	168
306	122	74	351	84	148	396	168	169
307	74	124	352	148	137	397	169	170
308	124	84	353	74	148	398	170	141
309	84	123	354	148	143	399	74	171
310	123	89	355	143	149	400	171	167
311	118	89	356	149	139	401	167	172
312	119	79	357	137	149	402	172	173
313	114	79	358	149	145	403	173	174
314	115	74	359	145	150	404	174	156
315	121	74	360	139	150	405	151	166

Table 9: Tower Element Definition

	Node 1	**Node 2**		**Node 1**	**Node 2**		**Node 1**	**Node 2**
406	171	136	451	145	179	496	201	191
407	152	167	452	160	183	497	200	202
408	167	137	453	176	188	498	202	197
409	153	172	454	188	167	499	198	195
410	168	138	455	188	184	500	199	196
411	154	173	456	188	186	501	192	190
412	169	139	457	167	184	502	195	199
413	155	174	458	176	184	503	193	201
414	170	140	459	176	186	504	190	201
415	166	152	460	167	186	505	199	202
416	171	137	461	177	184	506	196	202
417	152	172	462	184	168	507	147	206
418	137	168	463	184	178	508	206	207
419	153	173	464	184	169	509	207	208
420	138	169	465	178	185	510	147	203
421	154	174	466	169	185	511	203	204
422	139	170	467	179	185	512	204	205
423	84	180	468	185	170	513	162	212
424	180	176	469	185	147	514	212	213
425	176	181	470	185	141	515	213	214
426	181	182	471	172	186	516	162	209
427	182	183	472	186	181	517	209	210
428	183	162	473	186	173	518	210	211
429	89	175	474	186	182	519	206	203
430	175	176	475	173	187	520	212	209
431	176	177	476	182	187	521	207	204
432	177	178	477	174	187	522	213	210
433	178	179	478	187	183	523	208	215
434	179	147	479	187	156	524	215	205
435	142	180	480	187	162	525	214	216
436	143	176	481	141	192	526	216	211
437	144	177	482	192	193	527	206	204
438	145	178	483	193	194	528	209	213
439	146	179	484	141	189	529	207	215
440	157	175	485	189	190	530	204	215
441	158	176	486	190	191	531	213	216
442	159	181	487	156	198	532	210	216
443	160	182	488	198	199	533	195	348
444	161	183	489	199	200	534	209	348
445	180	143	490	156	195	535	162	348
446	175	158	491	195	196	536	156	348
447	143	177	492	196	197	537	206	351
448	158	181	493	192	189	538	147	351
449	144	178	494	193	190	539	141	351
450	159	182	495	194	201	540	192	351

Table 10: Tower Element Definition

	Node 1	**Node 2**		**Node 1**	**Node 2**		**Node 1**	**Node 2**
541	195	349	586	243	285	631	236	237
542	196	349	587	243	264	632	237	238
543	210	349	588	243	306	633	238	239
544	209	349	589	286	287	634	239	240
545	206	352	590	287	288	635	240	241
546	192	352	591	288	289	636	241	242
547	193	352	592	289	290	637	244	245
548	207	352	593	290	291	638	245	246
549	197	350	594	291	292	639	246	247
550	211	350	595	292	194	640	247	248
551	210	350	596	265	266	641	248	249
552	196	350	597	266	267	642	249	250
553	194	353	598	267	268	643	250	251
554	208	353	599	268	269	644	251	252
555	207	353	600	269	270	645	252	253
556	193	353	601	270	271	646	253	254
557	208	217	602	271	208	647	254	255
558	217	194	603	285	284	648	255	256
559	211	218	604	284	283	649	256	257
560	218	197	605	283	282	650	257	258
561	203	219	606	306	305	651	258	259
562	219	189	607	305	304	652	259	260
563	198	220	608	304	303	653	260	261
564	220	212	609	282	281	654	261	262
565	219	204	610	281	280	655	262	263
566	219	190	611	280	279	656	263	264
567	220	199	612	279	211	657	191	293
568	220	213	613	303	302	658	293	294
569	190	354	614	302	301	659	294	295
570	205	354	615	301	300	660	295	296
571	191	354	616	300	197	661	296	297
572	204	354	617	222	223	662	297	298
573	214	355	618	223	224	663	298	299
574	200	355	619	224	225	664	299	200
575	213	355	620	225	226	665	205	272
576	199	355	621	226	227	666	272	273
577	205	307	622	227	228	667	273	274
578	191	307	623	228	229	668	274	275
579	308	214	624	229	230	669	275	276
580	200	308	625	230	231	670	276	277
581	221	222	626	231	232	671	277	278
582	221	265	627	232	233	672	278	214
583	221	244	628	233	234	673	286	265
584	221	286	629	234	235	674	286	347
585	243	242	630	235	236	675	287	347

Table 11: Tower Element Definition

	Node 1	Node 2		Node 1	Node 2		Node 1	Node 2
676	266	347	721	299	339	766	249	227
677	265	347	722	278	339	767	249	330
678	268	346	723	308	202	768	330	228
679	289	346	724	308	216	769	330	250
680	287	346	725	202	218	770	227	330
681	266	346	726	216	218	771	250	228
682	225	313	727	202	216	772	250	310
683	313	247	728	197	340	773	310	230
684	268	313	729	279	340	774	228	310
685	289	313	730	300	340	775	310	252
686	269	313	731	211	340	776	229	310
687	290	313	732	281	341	777	310	251
688	269	343	733	302	341	778	252	328
689	271	343	734	300	341	779	328	253
690	292	343	735	279	341	780	328	231
691	290	343	736	281	312	781	230	328
692	194	342	737	302	312	782	253	231
693	208	342	738	239	312	783	254	232
694	271	342	739	312	261	784	255	233
695	292	342	740	312	303	785	255	329
696	217	201	741	282	312	786	329	256
697	217	215	742	284	344	787	329	234
698	201	215	743	305	344	788	233	329
699	201	307	744	285	345	789	256	311
700	215	307	745	306	345	790	311	236
701	272	337	746	303	344	791	234	311
702	293	337	747	282	344	792	311	258
703	191	337	748	305	345	793	235	311
704	205	337	749	284	345	794	311	257
705	295	336	750	285	306	795	236	258
706	274	336	751	244	222	796	237	333
707	272	336	752	222	245	797	259	333
708	293	336	753	245	223	798	258	333
709	295	309	754	223	246	799	236	333
710	274	309	755	246	224	800	259	237
711	275	309	756	224	247	801	237	334
712	309	296	757	247	225	802	238	334
713	309	297	758	247	332	803	260	334
714	309	276	759	226	332	804	259	334
715	297	338	760	248	332	805	238	335
716	299	338	761	225	332	806	261	335
717	278	338	762	227	331	807	239	335
718	276	338	763	249	331	808	260	335
719	200	339	764	248	331	809	261	239
720	214	339	765	226	331	810	239	262

Table 12: Tower Element Definition

	Node 1	Node 2		Node 1	Node 2		Node 1	Node 2
811	262	240	856	200	258	901	250	326
812	240	263	857	214	236	902	326	322
813	263	241	858	258	197	903	321	322
814	241	264	859	236	211	904	227	325
815	264	242	860	197	259	905	249	327
816	244	286	861	211	237	906	327	323
817	265	222	862	259	301	907	325	323
818	286	245	863	237	280	908	322	323
819	265	223	864	301	260	909	323	324
820	245	288	865	280	238	910	250	321
821	223	267	866	260	303	911	321	326
822	288	247	867	238	282	912	249	325
823	267	225	868	303	261	913	325	327
824	289	247	869	282	239	914	249	326
825	268	225	870	261	304	915	227	321
826	289	248	871	239	283	916	321	325
827	268	226	872	304	263	917	326	327
828	226	270	873	283	241	918	325	322
829	248	291	874	263	306	919	327	322
830	291	249	875	241	285	920	214	51
831	270	227	876	306	264	921	258	51
832	227	208	877	285	242	922	236	51
833	249	194	878	232	309	923	200	51
834	208	228	879	309	254	924	211	358
835	194	250	880	258	319	925	236	358
836	250	191	881	236	314	926	258	358
837	228	205	882	314	315	927	197	358
838	191	252	883	319	315	928	228	356
839	205	230	884	315	316	929	250	356
840	252	294	885	316	317	930	228	357
841	230	273	886	316	320	931	205	357
842	294	253	887	320	259	932	194	356
843	273	231	888	316	318	933	208	356
844	253	296	889	318	237	934	191	357
845	231	275	890	259	319	935	250	357
846	254	296	891	237	314	936	121	115
847	232	275	892	319	320	937	114	119
848	296	255	893	314	318	938	116	118
849	275	233	894	320	315	939	120	117
850	255	298	895	318	315	940	101	111
851	233	277	896	258	314	941	100	110
852	298	256	897	314	319	942	107	99
853	277	234	898	237	320	943	106	98
854	256	200	899	318	320	944	103	109
855	234	214	900	228	321	945	102	108

Table 13: Tower Element Definition

	Node 1	**Node 2**
946	113	105
947	112	104

Table 14: Tower Grouping

Design Variable 1										
1	2	3	4	5	23	24	25	26	27	45
46	47	48	49	60	61	62	63	64	195	196
197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	333	334	335	336
337	338	339	340	341	342	343	344	363	364	365
366	367	368	369	370	371	372	373	374	481	482
483	490	491	492	507	508	509	516	517	518	
Design Variable 2										
6	7	8	9	10	18	19	20	21	22	50
51	52	53	54	55	56	57	58	59	89	90
91	92	93	94	95	96	97	98	121	122	123
124	125	126	127	128	129	130	457	458	459	460
463	464	465	466	469	470	473	474	475	476	479
480	495	496	497	498	523	524	525	526	533	534
535	536	537	538	539	540	541	542	543	544	545
546	547	548	657	658	659	660	661	662	663	664
665	666	667	668	669	670	671	672	783		
Design Variable 3										
157	158	162	163	167	168	172	173	453	454	686
687	698	727	736	737	771	782	784	795	820	821
846	847	872	873	878	879	920	921	922	923	924
925	926	927	928	929	930	931	932	933	934	935
Design Variable 4										
11	12	28	29	35	36	37	38	39	40	41
42	43	44	65	66	72	73	79	80	81	82
83	84	85	86	87	88	99	100	106	107	113
114	115	116	117	118	119	120	131	132	138	139
145	146	147	148	149	150	151	152	153	154	155
156	159	160	164	165	169	170	174	175	240	242
244	246	248	250	252	254	255	256	257	258	259
260	261	262	263	264	265	266	267	268	269	270
271	272	273	274	275	276	277	278	295	296	297
298	299	300	301	302	311	312	313	314	315	316
317	318	327	328	329	330	349	350	391	392	405
406	411	412	413	414	419	420	421	422	435	438
439	440	443	444	449	450	451	452	467	468	477
478	493	494	499	500	501	502	503	504	505	506
519	520	521	522	527	528	529	530	531	532	561
562	563	564	565	566	567	568	569	570	571	572
573	574	575	576	692	693	694	695	696	697	699
700	701	702	703	704	705	706	707	708	709	710
713	714	715	716	717	718	719	720	721	722	723

Table 15: Tower Grouping

Design Variable 4										
713	714	715	716	717	718	719	720	721	722	723
724	725	726	728	729	730	731	752	753	754	755
756	757	766	772	773	774	775	776	777	778	779
780	781	785	786	787	788	789	790	791	792	793
794	800	809	810	811	812	813	814	822	823	870
871	890	891	892	893	897	898	899	911	912	913
914	915	916	917							
Design Variable 5										
13	14	15	30	31	32	67	68	69	74	75
76	101	102	103	108	109	110	133	134	135	140
141	142	161	166	171	176	190	193	239	241	243
245	247	249	251	253	345	346	347	348	387	388
389	390	409	410	415	416	417	418	437	442	445
446	447	448	455	456	461	462	471	472	673	674
675	676	677	678	679	680	681	682	683	688	689
690	691	732	733	734	738	739	744	745	746	747
748	749	750	751	758	759	760	761	762	763	764
765	767	768	769	770	735	742	743	796	797	798
799	801	802	803	804	805	806	807	808	815	816
817	876	877	894	895	896	910	918	919		
Design Variable 6										
16	17	33	34	70	71	77	78	104	105	111
112	136	137	143	144	185	186	187	188	189	191
192	194	303	304	307	308	331	332	407	408	436
441	557	558	559	560	577	578	579	580	684	685
740	741	818	819	874	875	880	881	882	883	886
887	888	889	900	901	902	903	904	905	906	907
Design Variable 7										
177	178	179	180	181	182	183	184	623	624	625
626	627	628	629	630	643	644	645	646	647	648
649	650									
Design Variable 8										
215	216	217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234	235	236
237	238	279	280	281	282	283	284	285	286	287
288	289	290	291	292	293	294				
Design Variable 9										
305	306	309	310							
Design Variable 10										
319	320	321	322	323	324	325	326			

Table 16: Tower Grouping

Design Variable 11										
393	394	399	400	423	424	429	430	581	583	585
587	617	618	619	634	635	636	637	638	639	654
655	656									
Design Variable 12										
395	396	397	398	401	402	403	404	425	426	427
428	431	432	433	434						
Design Variable 13										
484	485	486	487	488	489	510	511	512	513	514
515										
Design Variable 14										
834	835	836	837	856	857	858	859			
Design Variable 15										
824	825	826	827	828	829	830	831	832	833	838
839	840	841	842	843	844	845	848	849	850	851
852	853	854	855	860	861	862	863	864	865	866
867	868	869								
Design Variable 16										
582	584	586	588	589	590	591	592	593	594	595
596	597	598	599	600	601	602	603	604	605	606
607	608	609	610	611	612	613	614	615	616	620
621	622	631	632	633	640	641	642	651	652	653
Design Variable 17										
884	885	908	909							
Design Variable 18										
711	712									
Design Variable 19										
549	550	551	552	553	554	555	556			
Design Variable 20										
351	352	353	354	355	356	357	358	359	360	361
362	375	376	377	378	379	380	381	382	383	384
385	386									
Design Variable 21										
936	937	938	939							
Design Variable 22										
940	941	942	943	944	945	946	947			

17.5 Equal Leg Angle Section List

Table 17: Equal Angle Section List

Designation hxbxt mm	**m** kg/m	**A** 10^3 mm^2	$\mathbf{r_u}$ mm	$\mathbf{r_v}$ mm	**J** 10^3mm^4	$\mathbf{a_y}$ mm
25x25x3	1.11	0.142	9.43	4.83	0.476	7.21
25x25x5	1.77	0.226	9.14	4.8	1.98	7.98
30x30x3	1.36	0.174	11.3	5.81	0.635	8.35
30x30x5	2.18	0.278	11.1	5.75	2.58	9.18
40x40x3	1.85	0.235	15.3	7.84	0.882	10.7
40x40x4	2.42	0.308	15.2	7.77	1.92	11.2
40x40x5	2.97	0.379	15.1	7.73	3.56	11.6
40x40x6	3.52	0.448	14.9	7.7	5.92	12
45x45x3	2.1	0.268	17.2	8.88	1.06	11.9
45x45x4	2.74	0.349	17.1	8.76	2.27	12.3
45x45x5	3.38	0.43	17	8.71	4.17	12.8
45x45x6	4	0.509	16.9	8.67	6.9	13.2
50x50x3	2.34	0.298	19.2	9.92	1.15	13.1
50x50x4	3.06	0.389	19.1	9.79	2.48	13.6
50x50x5	3.77	0.48	19	9.73	4.58	14
50x50x6	4.47	0.569	18.9	9.68	7.62	14.5
50x50x8	5.82	0.741	18.6	9.63	17	15.2

Table 18: Equal Angle Section List

Designation	**m**	**A**	$\mathbf{r_u}$	$\mathbf{r_v}$	**J**	$\mathbf{a_y}$
hxbxt mm	**kg/m**	10^3 **mm**^{2}	**mm**	**mm**	10^3**mm**^{4}	**mm**
60x60x4	3.7	0.471	23	11.8	3.07	16
60x60x5	4.57	0.582	23	11.7	5.64	16.4
60x60x6	5.42	0.691	22.9	11.7	9.36	16.9
60x60x8	7.09	0.903	22.6	11.6	21	17.7
60x60x10	8.69	1.11	22.3	11.6	39.2	18.5
70x70x6	6.38	0.813	26.8	13.7	11.2	19.3
70x70x8	8.36	1.06	26.6	13.6	25	20.1
70x70x10	10.3	1.31	26.3	13.5	46.8	20.9
80x80x6	7.34	0.935	30.8	15.7	13	21.7
80x80x8	9.63	1.23	30.6	15.6	29.1	22.6
80x80x10	11.9	1.51	30.3	15.5	54.5	23.4
80x80x12	14	1.79	30	15.5	91.2	24.1
90x90x6	8.3	1.06	34.7	17.8	15	24.1
90x90x8	10.9	1.39	34.5	17.6	33.3	25
90x90x10	13.4	1.71	34.3	17.5	62.4	25.8
90x90x12	15.9	2.03	34	17.4	104	26.6
100x100x8	12.2	1.55	38.5	19.6	37.6	27.4
100x100x10	15	1.92	38.3	19.5	70.3	28.2
100x100x12	17.8	2.27	38	19.4	118	29
100x100x15	21.9	2.79	37.5	19.3	221	30.2
120x120x8	14.7	1.87	46.5	23.7	45.4	32.3
120x120x10	18.2	2.32	46.3	23.6	85.1	33.1
120x120x12	21.6	2.75	46	23.5	143	34
120x120x15	26.6	3.39	45.6	23.3	269	35.1
150x150x10	23	2.93	58.2	29.7	110	40.3
150x150x12	27.3	3.48	58	29.5	184	41.2
150x150x15	33.8	4.3	57.6	29.3	347	42.5
150x150x18	40.1	5.1	57.1	29.2	584	43.7
200x200x16	48.5	6.18	77.6	39.4	564	55.2
200x200x18	54.2	6.91	77.3	39.3	790	56
200x200x20	59.9	7.63	77	39.2	1070	56.8
200x200x24	71.1	9.06	76.4	39	1800	58.4

17.6 Eskom Transmission Tower: Load Cases

All the loading is in kN. The load cases are described in section 14.1.

Table 19: Case 1A

Loading Attachment Point	**Required Vertical Load V**	**Required Transverse Load H**	**Required Longitudinal Load**	**Rigging Weight**	**Vertical Resultant**
E1T	0	15.18	0	0.06	-0.06
E1TA	11.3	-4.11	0	0.06	11.24
E2T	11.3	11.07	0	0.06	11.24
C1T	93.5	91	0	0.38	93.12
C2T	93.5	91	0	0.38	93.12
C3T	93.5	91	0	0.38	93.12
W2T	0	17.2	0	0.21	-0.21
W3T	0	26.8	0	0.29	-0.29
W4T	0	28.8	0	0.2	-0.2
W5T	0	30.6	0	0.24	-0.24
W6T	0	41.3	0	0.29	-0.29

Table 20: Case 1AR

Loading Attachment Point	**Required Vertical Load V**	**Required Transverse Load H**	**Required Longitudinal Load**	**Rigging Weight**	**Vertical Resultant**
E1T	0	12.63	0	0.06	-0.06
E1TA	4.29	-1.56	0	0.06	4.23
E2T	4.3	11.07	0	0.06	4.24
C1T	35.53	91	0	0.38	35.15
C2T	35.53	91	0	0.38	35.15
C3T	35.53	91	0	0.38	35.15
W2T	0	17.2	0	0.21	-0.21
W3T	0	26.8	0	0.29	-0.29
W4T	0	28.8	0	0.2	-0.2
W5T	0	30.6	0	0.24	-0.24
W6T	0	41.3	0	0.29	-0.29

Table 21: Case 2A

Loading Attachment Point	**Required Vertical Load V**	**Required Transverse Load H**	**Required Longitudinal Load**	**Rigging Weight**	**Vertical Resultant**
E1T	0	4.18	0	0.06	-0.06
E1TA	6.56	-2.39	0	0.06	6.5
E2T	6.56	1.79	0	0.06	6.5
C1T	55	14.3	0	0.1	54.9
C2T	55	14.3	0	0.1	54.9
C3T	55	14.3	0	0.1	54.9
W2T	0	4.3	0	0.21	-0.21
W3T	0	6.8	0	0.29	-0.29
W4T	0	7.2	0	0.2	-0.2
W5T	0	7.7	0	0.24	-0.24
W6T	0	10.3	0	0.29	-0.29
E1L	0	0	30	0.06	-0.06
E2L	0	0	30	0.06	-0.06
C1L	0	0	74.31	0.2	-0.2
C2L	0	0	74.31	0.2	-0.2
C3L	0	0	74.31	0.2	-0.2

Table 22: Case 2BR

Loading Attachment Point	Required Vertical Load V	Required Transverse Load H	Required Longitudinal Load	Rigging Weight	Vertical Resultant
E1T	0	2.7	0	0.06	-0.06
E1TA	2.49	-0.91	0	0.06	2.43
E2T	-0.23	2.33	0	0.06	-0.29
C1T	20.9	14.3	0	0.1	20.8
C2T	20.9	14.3	0	0.1	20.8
C3T	27.21	18.9	0	0.1	27.11
W2T	0	4.3	0	0.21	-0.21
W3T	0	6.8	0	0.29	-0.29
W4T	0	7.2	0	0.2	-0.2
W5T	0	7.7	0	0.24	-0.24
W6T	0	10.3	0	0.29	-0.29
E1L	0	0	30	0.06	-0.06
C1L	0	0	74.85	0.2	-0.2
C2L	0	0	74.85	0.2	-0.2
E2TA	3.32	2.94	0	0.06	3.26

Table 23: Case 3

Loading Attachment Point	Required Vertical Load V	Required Transverse Load H	Required Longitudinal Load	Rigging Weight	Vertical Resultant
E1T	0	9.31	0	0.06	-0.06
E2T	0	9.31	0	0.06	-0.06
C1T	0	75.4	0	0.38	-0.38
C2T	0	75.4	0	0.38	-0.38
C3T	0	75.4	0	0.38	-0.38
W2T	0	17.2	0	0.21	-0.21
W3T	0	26.8	0	0.29	-0.29
W4T	0	28.8	0	0.2	-0.2
W5T	0	30.6	0	0.24	-0.24
W6T	0	41.3	0	0.29	-0.29

Table 24: Case 4A

Loading Attachment Point	**Required Vertical Load V**	**Required Transverse Load H**	**Required Longitudinal Load**	**Rigging Weight**	**Vertical Resultant**
C1V	125	0	0	0.38	124.62

Table 25: Case 4B

Loading Attachment Point	**Required Vertical Load V**	**Required Transverse Load H**	**Required Longitudinal Load**	**Rigging Weight**	**Vertical Resultant**
C1V	125	0	0	0.38	124.62

Table 26: Case 5

Loading Attachment Point	**Required Vertical Load V**	**Required Transverse Load H**	**Required Longitudinal Load**	**Rigging Weight**	**Vertical Resultant**
E1V	18.3	0	0	0.06	18.24
E2V	18.3	0	0	0.06	18.24
C1V	159	0	0	0.38	158.62
C2V	159	0	0	0.38	158.62
C3V	159	0	0	0.38	158.62

17.7 Algorithm Code Extracts

Algorithm

This class calls all the genetic operators and the analysis. It is written in a loop, the loop runs from zero to the maximum number of generations.

```java
import aho.math.linalg.Vector;
import fem.components.element.FrameElement;
import fem.components.element.TrussElement;
//import java.util.Set;
//import java.util.Iterator;
import java.util.LinkedList;

//The algorithm either creates a truss and uses FEM with it or optimises a test function
public class Algorithm{
    //Algorithm attributes
    Population population;
    TrussPopulation trusses;
    int entriesGA;
    FrameTruss trussInput;
    String[] dof;
    String[][] elementString;
    double[] largestDispl, nodesX, nodesY, nodesZ;
    int[] femIndexArray, femElementArray;
    double[] areaArray, IxxArray, IyyArray, lengthArray, rvvArray,
            ruuArray, JArray, axArray, tArray, bArray;
    double[][] forceArray, load;
    String databasePath;
    //For the case of a truss
    public Algorithm(ModelInput input, double poison, double emod, double density,
            int entries, FrameParameters gaParam, String databasePath, int columnArea,
            int b, int t, int columnRvv, int columnRuu,
            int columnJ, int columnax, double fy, double span, Boolean isGroupedBoolean,
            int numberOfElements, int columnGrouping, Boolean is3D, double deflectionLimit,
            double stressLimit, Boolean staticFitness, Boolean dynamicFitness,
            Boolean normalisedFitness, Boolean useSANS, Boolean isCHS, Boolean isEqualL,
            Boolean useDeflectionLimit, Boolean findRedundantElements, Boolean isFrameElement){

        MemberAttributes attributes = new MemberAttributes(databasePath, columnArea, fy);

        //Set the arrays, the array can be set for the whole algorithm, the database does not change
        //Set these arrays outside of the algorithm to save on computation time
        //Only members of class three is included
        setArrays(gaParam, attributes, input, b, t, columnArea,
                columnRvv, columnRuu, columnJ, columnax, entries, numberOfElements,
                columnGrouping, isCHS, isEqualL, useSANS);

        //Initialies Population
        population = new Population(input, density, this.entriesGA, gaParam, areaArray,
                lengthArray, rvvArray, ruuArray, emod, fy,
                JArray, tArray, bArray, axArray, span, numberOfElements,
                femIndexArray, femElementArray, deflectionLimit, stressLimit,
                staticFitness, dynamicFitness, normalisedFitness, useSANS, isCHS,
                isEqualL, useDeflectionLimit, findRedundantElements); //, largestDispl, forceArray

//          population.createFemArray();

        //Create initial FEM
            trusses = new TrussPopulation(input, emod, poison, density, population.currentArrayOfIndividuals,
                    this.entriesGA, numberOfElements, load, nodesX,
                    nodesY, nodesZ, dof, elementString, isGroupedBoolean, is3D,
                    isFrameElement);
            trusses.createFemModels(femIndexArray, femElementArray, numberOfElements);

            for (int j = 0; j < gaParam.POPULATION_SIZE; j++){
```

```
        trusses.analyse(trusses.trussmodelArray[j]);
    }

    LinkedList<TrussElement> elts = new LinkedList<TrussElement>();
    LinkedList<FrameElement> eltsFrame = new LinkedList<FrameElement>();

    if(isFrameElement == false){
        for (int j = 0; j < gaParam.POPULATION_SIZE; j++){
            elts.clear();
            elts = trusses.getElements(trusses.trussmodelArray[j]);
            int p = 0;
            largestDispl[j] = 0;
            for(TrussElement te: elts){
              //Obtain axial forces in truss members
                forceArray[j][p] = te.getElementResultVector().get(0);        //CHECK
                if(is3D == false){

//                        System.out.println("Displacement Array ");

                    double minDisp = Math.abs(te.getDOFDisplacements().minEntry()*1000);

                    double maxDisp = Math.abs(te.getDOFDisplacements().maxEntry()*1000);
                    double displacement = Math.max(minDisp, maxDisp);

                    if (displacement > Math.abs(largestDispl[j])){
                        largestDispl[j] = displacement;
                    }
//                        System.out.println("max displ " + largestDispl[j]);
                }

                else{
```

```
//                        System.out.println("Displacement Array ");
                            double minDisp = Math.abs(te.getDOFDisplacements().minEntry()*1000);
                            double maxDisp = Math.abs(te.getDOFDisplacements().maxEntry()*1000);
                            double displacement = Math.max(minDisp, maxDisp);

                            if (displacement > Math.abs(largestDispl[j])){
                                largestDispl[j] = displacement;
                            }
//                        System.out.println("max displ " + largestDispl[j]);

                        }

                        p++;
                    }
                }
            } else{
                for (int j = 0; j < gaParam.POPULATION_SIZE; j++){
                    eltsFrame.clear();
                    eltsFrame = trusses.getElementsFrame(trusses.trussmodelArray[j]);
                    int p = 0;
                    largestDispl[j] = 0;
                    for(FrameElement te: eltsFrame){
                      //Obtain axial forces in truss members
                        forceArray[j][p] = te.getElementResultVector().get(0);        //CHECK
                        if(is3D == false){

//                        System.out.println("Displacement Array ");

                            double minDisp = Math.abs(te.getDOFDisplacements().minEntry()*1000);

                            double maxDisp = Math.abs(te.getDOFDisplacements().maxEntry()*1000);
                            double displacement = Math.max(minDisp, maxDisp);

                            if (displacement > Math.abs(largestDispl[j])){
                                largestDispl[j] = displacement;
                            }
//                        System.out.println("max displ " + largestDispl[j]);
                        }

                        else{
//                        System.out.println("Displacement Array ");
                            double minDisp = Math.abs(te.getDOFDisplacements().minEntry()*1000);
                            double maxDisp = Math.abs(te.getDOFDisplacements().maxEntry()*1000);
                            double displacement = Math.max(minDisp, maxDisp);

                            if (displacement > Math.abs(largestDispl[j])){
                                largestDispl[j] = displacement;
                            }
//                        System.out.println("max displ " + largestDispl[j]);

                        }

                        p++;
                    }
                }
            }
        GA(gaParam, input, emod, poison, density, elts, eltsFrame, isGroupedBoolean, numberOfElements,
                is3D, deflectionLimit, stressLimit, isFrameElement);

    }

    public void setArrays(FrameParameters gaParam, MemberAttributes attributes,
            ModelInput input, int b, int t, int columnArea, int columnRvv, int columnRuu,
            int columnJ, int columnax, int entries, int numberOfElements, int columnGrouping,
            Boolean isCHS, Boolean isEqualL, Boolean isSANS){
                areaArray = new double[entries];
                areaArray = attributes.getAreaArray(entries, b, t, columnArea, isCHS, isEqualL, isSANS);         //mm^2
                this.entriesGA = attributes.getEntries();
//                IxxArray = new double[entries];
//                IxxArray = attributes.getIxxArray(entries, columnIxx);
//                IyyArray = new double[entries];
//                IyyArray = attributes.getIyyArray(entries, columnIyy);
                rvvArray = new double[this.entriesGA];
                rvvArray = attributes.getrvvArray(this.entriesGA, columnRvv);
                ruuArray = new double[this.entriesGA];
                ruuArray = attributes.getruuArray(this.entriesGA, columnRuu);
                JArray = new double[this.entriesGA];
                JArray = attributes.getJArray(this.entriesGA, columnJ);
                axArray = new double[this.entriesGA];
                axArray = attributes.getaxArray(this.entriesGA, columnax);
                forceArray = new double[gaParam.POPULATION_SIZE][numberOfElements];
                tArray = new double[this.entriesGA];
                tArray = attributes.gettArray(this.entriesGA, t);
                bArray = new double[this.entriesGA];
                bArray = attributes.getbArray(this.entriesGA, b);
                largestDispl = new double[gaParam.POPULATION_SIZE];
                femIndexArray = new int[numberOfElements];
                femIndexArray = input.getFEMIndices();
                femElementArray = new int [numberOfElements];
                femElementArray = input.getFemGroupElements();
                lengthArray = new double[gaParam.NUMBER_OF_CHROMOSOMES];
                lengthArray = input.getLength();       //mm
```

```
          //To truss
          load = input.getLoad();  //N
          nodesX = input.getNodesX(); //m
          nodesY = input.getNodesY(); //m
          nodesZ = input.getNodesZ();
          dof = input.getDOF();
          elementString = input.getElementsString();
    }

    public void GA(FrameParameters gaParam, ModelInput input, double emod,
        double poison, double density, LinkedList<TrussElement> elts, LinkedList<FrameElement> eltsFrame,
        Boolean isGroupedBoolean, int numberOfElements, Boolean is3D,
        double deflectionLimit, double stressLimit, Boolean isFrameElement){
      int eliteNumber =0;
      for(int i = 0; i < gaParam.MAX_GENERATION; i++){


  //       System.out.println("------------------------------------------------");
  //       System.out.println("Generation "+ (i+1) );
  //       System.out.println("------------------------------------------------");
        System.out.print(i+1 + ",");

        //Selection occurs within crossover
        //Crossover creates temp array
        //Find fitness array
        population.objectiveArray(population.currentArrayOfIndividuals, largestDispl, forceArray, (i+1));
        //Find fittest individual
        population.fittestIndividualObjective();
        //Find weakest individual
        population.weakestIndividualObjective();
        //Find fitness array
```

```
            population.populationFitness(population.currentArrayOfIndividuals, largestDispl, forceArray, (i+1));
            //Compute statistics
            population.statistics();
//            largestCurrentDisplElite = largestDispl;
            population.scaleFitnessFEM(largestDispl, forceArray, (i+1));
            //Perform crossover
            population.crossover();

            //Mutate
            for(int j = 0; j < gaParam.POPULATION_SIZE; j++){
                population.mutate(population.tempArrayOfIndividuals[j]);
            }

            //Must perform FEM again on new population before elitism can be performed
             //Create FEM
//            population.createFemArray();
            for (int w = 0; w < gaParam.POPULATION_SIZE; w++)
                trusses.clearTrussModel(trusses.trussmodelArray[w]);
            trusses = new TrussPopulation(input, emod, poison, density, population.tempArrayOfIndividuals,
                    this.entriesGA, numberOfElements, load, nodesX,
                    nodesY, nodesZ, dof, elementString, isGroupedBoolean, is3D,
                    isFrameElement);
            trusses.createFemModels(femIndexArray, femElementArray, numberOfElements);

            for (int j = 0; j < gaParam.POPULATION_SIZE; j++){
                trusses.analyse(trusses.trussmodelArray[j]);
            }
            if(isFrameElement){
                for (int j = 0; j < gaParam.POPULATION_SIZE; j++){
                    elts.clear();
                    elts = trusses.getElements(trusses.trussmodelArray[j]);
                int p = 0;
                largestDispl[j] = 0;
                for(FrameElement te: eltsFrame){
                    forceArray[j][p] = te.getElementResultVector().get(0);        //CHECK

                    double minDisp = Math.abs(te.getDOFDisplacements().minEntry()*1000);
                    double maxDisp = Math.abs(te.getDOFDisplacements().maxEntry()*1000);
                    double displacement = Math.max(minDisp, maxDisp);

                    if (displacement > Math.abs(largestDispl[j])){
                        largestDispl[j] = displacement;
                    }
                    p++;
                }
            }
        }else{
            for (int j = 0; j < gaParam.POPULATION_SIZE; j++){
                elts.clear();
                elts = trusses.getElements(trusses.trussmodelArray[j]);
                int p = 0;
                largestDispl[j] = 0;
                for(TrussElement te: elts){
                    forceArray[j][p] = te.getElementResultVector().get(0);        //CHECK

                    double minDisp = Math.abs(te.getDOFDisplacements().minEntry()*1000);
                    double maxDisp = Math.abs(te.getDOFDisplacements().maxEntry()*1000);
                    double displacement = Math.max(minDisp, maxDisp);

                    if (displacement > Math.abs(largestDispl[j])){
                        largestDispl[j] = displacement;
                    }
                p++;
            }
        }
    }
    //Perform elitism
    if(gaParam.ELITISM){                //position 2
        eliteNumber = (int)(Math.random() * population.POPULATION_SIZE);
        population.elitism(largestDispl, forceArray, (i+1), eliteNumber);
    }

    //For the case where elitism did occur, a fem analysis is needed again
    //to obtain displacement of the new individual in the temporary array

    if (population.didElite()){

        for (int w = 0; w < gaParam.POPULATION_SIZE; w++)
            trusses.clearTrussModel(trusses.trussmodelArray[w]);
        trusses = new TrussPopulation(input, emod, poison, density, population.tempArrayOfIndividuals,
                this.entriesGA, numberOfElements, load, nodesX,
                nodesY, nodesZ, dof, elementString, isGroupedBoolean,
                is3D, isFrameElement);
        trusses.createFemModels(femIndexArray, femElementArray, numberOfElements);

        for (int j = 0; j < gaParam.POPULATION_SIZE; j++){
            trusses.analyse(trusses.trussmodelArray[j]);
        }
        if(isFrameElement){
            for (int j = 0; j < gaParam.POPULATION_SIZE; j++){
                eltsFrame.clear();
                eltsFrame = trusses.getElementsFrame(trusses.trussmodelArray[j]);
                int p = 0;
                largestDispl[j] = 0;
                for(FrameElement te: eltsFrame){
                    forceArray[j][p] = te.getElementResultVector().get(0);        //CHECK

                    double minDisp = Math.abs(te.getDOFDisplacements().minEntry()*1000);
                    double maxDisp = Math.abs(te.getDOFDisplacements().maxEntry()*1000);
                    double displacement = Math.max(minDisp, maxDisp);

                    if (displacement > Math.abs(largestDispl[j])){
                        largestDispl[j] = displacement;
                    }
                    p++;
                }
            }
        }else{
            for (int j = 0; j < gaParam.POPULATION_SIZE; j++){
                elts.clear();
                elts = trusses.getElements(trusses.trussmodelArray[j]);
                int p = 0;
                largestDispl[j] = 0;
                for(TrussElement te: elts){
                    forceArray[j][p] = te.getElementResultVector().get(0);        //CHECK

                    double minDisp = Math.abs(te.getDOFDisplacements().minEntry()*1000);
                    double maxDisp = Math.abs(te.getDOFDisplacements().maxEntry()*1000);
                    double displacement = Math.max(minDisp, maxDisp);

                    if (displacement > Math.abs(largestDispl[j])){
                        largestDispl[j] = displacement;
                    }
                    p++;
```

```
                        }
                    }
                }
            }
//            System.out.println();
//            for(int w =0; w < population.POPULATION_SIZE; w++){
//                System.out.print(largestDispl[w] + ",");
//            }
//            System.out.println();
//            System.out.println();
            //Replace the current population with the temporary population
            population.replace();
            System.gc();
            Vector.clearVectors();
        }
        System.exit(0);
    }

}
```

Listing 1: Class Algorithm

Population

This class creates two arrays of individuals. This class also contains the code for the genetic operators such as selection, scaling, crossover, mutation and elitism.

```
package GeneticAlgorithm;

import java.util.Set;
import java.util.HashSet;
```

```java
public class Population{
    //create two arrays to for the temp and current populations
    Individual[] currentArrayOfIndividuals, femPopulationArray;
    Individual[] tempArrayOfIndividuals;
    Individual fittestCurrentIndividual, weakestCurrentIndividual, fittestTempIndividual;
    double[] popObjectiveValues, populationFitnessArray, areaArray,
            lengthArray, rvv, ruu, JArray, t, b, ax;
    //double[][] forceArray;
    ModelInput input;
    boolean ELITISM, REPLACEMENT, staticFitness, dynamicFitness, normalisedFitness,
            didElite, useSANS, isCHS, isEqualL, useDeflectionLimit,findRedundantElements;
    double density, CROSSOVER_PROBABILITY,MUTATION_PROBABILITY, SCALING_CONSTANT, span, fy, emod,highestFitness;
    int entries, NUMBER_OF_CHROMOSOMES, POPULATION_SIZE, SUBSTRING_LENGTH, TOTAL_STRING_LENGTH,
            CROSSOVER_POINTS, MAX_GENERATION, columnArea;
    double strongestMinObjective, weakestMaxObjective, aveFitness, sumFitness, maxFitness; //, largestDispl
    double deflectionLimit, stressLimit;
    int countChildren = 0;
    int eliteNumber;
    int[] femIndexArray, femElementArray;
    int fittestTempCount, fittestCurrentCount, numberOfElements;
    Set<Individual> selected = new HashSet<Individual>();
    Set<Integer> num = new HashSet<Integer>();
    //FEM
    public Population(ModelInput input, double density, int entries, FrameParameters gaParam,
            double[] areaArray, double[] lengthArray,
            double[] rvv, double[] ruu, double emod, double fy,
                    double[] J, double[] t, double[] b, double[] ax, double span,
                    int numberOfElements, int[] femIndexArray, int[] femGroupArray,
                    double deflectionLimit, double stressLimit, Boolean staticFitness,
                    Boolean dynamicFitness, Boolean normalisedFitness, Boolean useSANS,
            Boolean isCHS, Boolean isEqualL, Boolean useDeflectionLimit,
            Boolean findRedundantElements){
    this.useSANS = useSANS;
    this.isEqualL = isEqualL;
    this.isCHS = isCHS;
    this.findRedundantElements = findRedundantElements;
    this.useDeflectionLimit = useDeflectionLimit;
    this.deflectionLimit = deflectionLimit;
    this.staticFitness = staticFitness;
    this.dynamicFitness = dynamicFitness;
    this.normalisedFitness = normalisedFitness;
    this.stressLimit = stressLimit;
    this.rvv = rvv;
    this.ruu = ruu;
    this.emod = emod;
    this.fy = fy;
    this.JArray = J;
    this.t = t;
    this.b = b;
    this.ax = ax;
    this.span = span;
    this.numberOfElements = numberOfElements;
    this.femIndexArray = femIndexArray;
    this.femElementArray = femGroupArray;
    this.input = input;
    this.density = density;
    this.entries = entries;
    this.NUMBER_OF_CHROMOSOMES = gaParam.NUMBER_OF_CHROMOSOMES;
    this.POPULATION_SIZE = gaParam.POPULATION_SIZE;
    this.SUBSTRING_LENGTH = (int)Math.ceil(Math.ceil(Math.log(entries)*Math.pow(Math.log(2), -1)));
    this.TOTAL_STRING_LENGTH = NUMBER_OF_CHROMOSOMES * SUBSTRING_LENGTH;
```

```
        this.CROSSOVER_POINTS = gaParam.CROSSOVER_POINTS;
        this.CROSSOVER_PROBABILITY = gaParam.CROSSOVER_PROBABILITY;
        this.ELITISM = gaParam.ELITISM;
        this.MAX_GENERATION = gaParam.MAX_GENERATION;
        this.MUTATION_PROBABILITY = gaParam.MUTATION_PROBABILITY;
        this.SCALING_CONSTANT = gaParam.SCALING_CONSTANT;
        this.REPLACEMENT = gaParam.REPLACEMENT;
        this.areaArray = areaArray;
        this.lengthArray = lengthArray;
        popObjectiveValues = new double[gaParam.POPULATION_SIZE];
        currentArrayOfIndividuals = new Individual[gaParam.POPULATION_SIZE];
        tempArrayOfIndividuals = new Individual[gaParam.POPULATION_SIZE];
        fittestCurrentIndividual = new Individual(NUMBER_OF_CHROMOSOMES, SUBSTRING_LENGTH,
                areaArray, input, entries, lengthArray, femIndexArray, staticFitness,
                dynamicFitness, normalisedFitness, numberOfElements, useSANS, isCHS,
                isEqualL, useDeflectionLimit, findRedundantElements, JArray);
        fittestTempIndividual = new Individual(NUMBER_OF_CHROMOSOMES, SUBSTRING_LENGTH,
                areaArray, input, entries, lengthArray, femIndexArray, staticFitness,
                dynamicFitness, normalisedFitness, numberOfElements, useSANS, isCHS,
                isEqualL, useDeflectionLimit, findRedundantElements, JArray);
        femPopulationArray = new Individual[POPULATION_SIZE];
        populationFitnessArray = new double[POPULATION_SIZE];
        //Populate intitial population FEM
        for (int i = 0; i < currentArrayOfIndividuals.length; i++){
            currentArrayOfIndividuals[i] = new Individual(NUMBER_OF_CHROMOSOMES, SUBSTRING_LENGTH,
                    areaArray,  input, entries, lengthArray, femIndexArray, staticFitness,
                    dynamicFitness, normalisedFitness,numberOfElements, useSANS, isCHS,
                    isEqualL, useDeflectionLimit, findRedundantElements, JArray);
        }
//
//        System.out.println("currentArrayOfIndividuals ");
//       for (int i = 0; i < currentArrayOfIndividuals.length; i++){
//           for (int j = 0; j < currentArrayOfIndividuals[i].TOTAL_STRING_LENGTH; j++)
//           System.out.print(currentArrayOfIndividuals[i].individual[j]);
//           System.out.println();
//       }
   }

  public double[] objectiveArray(Individual[] individuals, double[] largestDispl,
      double[][] elementForceArray, int generation){
    //Finds the fitness of each individual in an array
//       System.out.println("population objectives");
    for(int i = 0; i < popObjectiveValues.length; i++){
      //System.out.println("Individual " + (i+1));
      popObjectiveValues[i] = individuals[i].findObjectiveFunctionValue(individuals[i],
          elementForceArray[i], density, rvv, ruu, emod, fy, JArray, t, b, ax,
          largestDispl[i], span, generation, deflectionLimit, stressLimit);
//           System.out.print(individuals[i].findFitnessFunctionValue(individuals[i], density) + " + ");
//           System.out.println();
//           System.out.println(popObjectiveValues[i]);
    }
    return popObjectiveValues;
  }

  public Individual weakestIndividualObjective(){
    weakestMaxObjective = 0;
    for (int i = 0; i < POPULATION_SIZE; i++){
      if (popObjectiveValues[i] > weakestMaxObjective){
        weakestMaxObjective = popObjectiveValues[i];
        weakestCurrentIndividual = currentArrayOfIndividuals[i];
      }
    }
```

```
        return weakestCurrentIndividual;
    }

    //Regarding objective function
    public Individual fittestIndividualObjective(){
        //Finds the fittest individual from the current population array
        strongestMinObjective = weakestMaxObjective;
        for (int i = 0; i < POPULATION_SIZE; i++){
            if (popObjectiveValues[i] < strongestMinObjective){
                strongestMinObjective = popObjectiveValues[i];
                fittestCurrentIndividual = currentArrayOfIndividuals[i];
                fittestCurrentCount = i;
            }
        }
        return fittestCurrentIndividual;
    }

    //Regarding fitness function
    public Individual findFittestIndividual(Individual[] individuals, double[] largestDispl,
            double[][] elementForceArray, int gen){
        //Finds the fittest individual an array of individuals
        fittestTempCount = 0;
        highestFitness = 0;
        for (int i = 0; i < POPULATION_SIZE; i++){
            if (individuals[i].findFitnessFunctionValue(individuals[i], elementForceArray[i],
                    density, rvv, ruu, emod, fy, JArray, t, b, ax, largestDispl[i],
                    span, gen,strongestMinObjective, weakestMaxObjective, deflectionLimit,
                    stressLimit) > highestFitness){
                highestFitness = individuals[i].findFitnessFunctionValue(individuals[i],
                        elementForceArray[i], density, rvv, ruu, emod, fy, JArray, t, b, ax,
                        largestDispl[i], span, gen,strongestMinObjective, weakestMaxObjective,
                    deflectionLimit, stressLimit);
                fittestTempIndividual = individuals[i];
                fittestTempCount = i;
            }
        }
        return fittestTempIndividual;
    }

    public void populationFitness(Individual[] individuals, double[] largestDispl,
            double[][] elementForceArray, int generation){
        for (int i = 0; i < POPULATION_SIZE; i++){
            populationFitnessArray[i] = individuals[i].findFitnessFunctionValue(individuals[i],
                    elementForceArray[i], density, rvv, ruu, emod, fy, JArray, t, b, ax,
                    largestDispl[i], span, generation, strongestMinObjective,
                    weakestMaxObjective, deflectionLimit, stressLimit);
        }
    }

    public void statistics(){
        //Find the maximum fitness and the fittest individual
        //Fittest individual of current population, fittest individual of new population determined in mutation
        //as it is the last operator to process new population

        //Find the sum fitness
        sumFitness =0;
        for (int i = 0; i < currentArrayOfIndividuals.length; i++){
            sumFitness += populationFitnessArray[i];
        }

        //Find the average fitness
        aveFitness = sumFitness * Math.pow(POPULATION_SIZE, -1);
        //Fittest individual with regard to fitness
        maxFitness = 0;
        for (int i = 0; i < POPULATION_SIZE; i++){
            if (populationFitnessArray[i] > maxFitness){
                maxFitness = populationFitnessArray[i];
                fittestCurrentIndividual = currentArrayOfIndividuals[i];
            }
        }
        //Print the statistics
//        NumberFormat formatter = new DecimalFormat("#0.000");
//        System.out.println("Population maximum fitness:   " + formatter.format(strongestMinObjective));
//        System.out.println("Population sum fitness:       " + formatter.format(sumFitness));
//        System.out.println("Population average fitness:   " + formatter.format(aveFitness));
        //aveFitnessArray[gen] = aveFitness;

    }

    public void scaleFitnessFEM(double[] largestDispl, double[][] elementForceArray, int gen){
       //Do not scale positions in excel sheet, only scale answers according to fitness from FEM
       //Apply linear scaling as in Goldberg's little genetic algorithm
       //scaled fitness = a * currentFitness + b; subjected to aveFitness and scalingConstant (user defined)
//       System.out.println("In scaledFitness");


       double[] temp = new double[POPULATION_SIZE];
       //CHECK fittestcount!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
//       System.out.println("Scaling Constant = " + SCALING_CONSTANT);
       if (SCALING_CONSTANT != 0 && (findFittestIndividual(currentArrayOfIndividuals,
               largestDispl, elementForceArray, gen).findFitnessFunctionValue(findFittestIndividual(currentArrayOfIndividuals,
               largestDispl, elementForceArray, gen), elementForceArray[fittestTempCount],
            density, rvv, ruu, emod, fy, JArray, t, b, ax, largestDispl[fittestTempCount], span,
            gen, strongestMinObjective, weakestMaxObjective, deflectionLimit, stressLimit)
            - aveFitness > 0) && (maxFitness != aveFitness)){
         sumFitness = 0;
         double constant1 = ((SCALING_CONSTANT - 1) * aveFitness) *
                 Math.pow((findFittestIndividual(currentArrayOfIndividuals,largestDispl,
                 elementForceArray, gen).findFitnessFunctionValue(findFittestIndividual(currentArrayOfIndividuals,
                 largestDispl, elementForceArray, gen), elementForceArray[fittestTempCount], density, rvv, ruu, emod,
                 fy, JArray, t, b, ax, largestDispl[fittestTempCount], span, gen, strongestMinObjective, weakestMaxObjective,
                 deflectionLimit, stressLimit)
                 - aveFitness), -1);
         double constant2 = (1 - constant1) * aveFitness;

         //Apply scaling
         for(int i = 0; i < POPULATION_SIZE; i++){
             temp[i] = constant1 * populationFitnessArray[i] + constant2;
             if(temp[i] < 0){
                 temp[i] = 0;                                              //Avoid negative values
             }
             sumFitness = sumFitness + temp[i];                            //New Sum fitness
         }
         //Copy temp into population fitness
         System.arraycopy(temp, 0, populationFitnessArray, 0, POPULATION_SIZE);

         aveFitness = sumFitness * Math.pow(POPULATION_SIZE, -1);

    }
}

 public double getFittest(){
     return highestFitness;
    }

    //FEM selection
    public Individual select(){
        //Roulette wheel selection
        double sum = 0;
        int index = 0;
        double rouletteWheel = Math.random() * sumFitness;
        while (sum < rouletteWheel && index < POPULATION_SIZE ) {
            sum += populationFitnessArray[index];               //!!! perhaps currentArrayOfIndividuals.fitness again???
            index += 1;
        }
        index -= 1;
        if (index == POPULATION_SIZE)
            return currentArrayOfIndividuals[index - 1];
        else
            return currentArrayOfIndividuals[index];

    }

    public void crossover(){
        //Perform crossover only crossoverProbability of the times
        selected.clear();
        num.clear();

        //crossover points
        int point;
        int[] crossoverPoints = new int[CROSSOVER_POINTS + 2];       //for beginning and end positions
        crossoverPoints[0] = 0;
        crossoverPoints[CROSSOVER_POINTS + 1] = TOTAL_STRING_LENGTH;
        countChildren = 0;
```

```
//place holders for multiple crossover
int begin = 0;
int counter = 0;
int end;

//Check
if(CROSSOVER_POINTS >= TOTAL_STRING_LENGTH){
    System.err.println("!Too many cross sites!");
}

//Define parent strings
Individual parent1;
Individual parent2;

//With or without replacement?
for (int a = 0; a < POPULATION_SIZE; a = a + 2){
    int i = 0;
    if(REPLACEMENT == true){
        parent1 = select();
        parent2 = select();
    }else{
        parent1 = select();
        parent2 = select();
        while((parent1 == parent2) || (selected.contains(parent1) == true) || (selected.contains(parent2) == true)){
            parent1 = select();
            parent2 = select();
        }
        selected.add(parent1);
        selected.add(parent2);
    }
```

```
//Initiate children
Individual child1 = new Individual(NUMBER_OF_CHROMOSOMES, SUBSTRING_LENGTH,
 areaArray, input, entries, lengthArray, femIndexArray, staticFitness,
        dynamicFitness, normalisedFitness, numberOfElements, useSANS, isCHS,
        isEqualL, useDeflectionLimit, findRedundantElements ,JArray);
Individual child2 = new Individual(NUMBER_OF_CHROMOSOMES, SUBSTRING_LENGTH,
 areaArray, input, entries, lengthArray, femIndexArray,staticFitness,
        dynamicFitness, normalisedFitness, numberOfElements, useSANS, isCHS,
        isEqualL, useDeflectionLimit, findRedundantElements, JArray);
countChildren = countChildren + 2;

if (Math.random() <= CROSSOVER_PROBABILITY){
    //Use set to check that two points do not fall on the same place
    //populate num
    for(int w = 1; w <= TOTAL_STRING_LENGTH; w++){
        num.add(w);
    }
    while(i < CROSSOVER_POINTS){
        //System.out.println("i = " + i);
        point = (int)(Math.round(Math.random() * TOTAL_STRING_LENGTH));
        if(num.contains(point)){
            crossoverPoints[i+1] = point;
            num.remove(point);
        }else{
            i -= 1;
        }
        i+=1;
    }

    //Perform crossover
    for(int k = 0; k < CROSSOVER_POINTS + 1; k++){
        begin = crossoverPoints[k];
        end = crossoverPoints[k+1];
        if(k>0)
            end = crossoverPoints[k+1] - 1;  //check for overlap
        if(k==CROSSOVER_POINTS)              //check for last point
            end = crossoverPoints[k+1];
        if(counter % 2 == 0){
            for(int m = begin; m < end; m++){
                    child1.individual[m] = parent1.individual[m];
                    child2.individual[m] = parent2.individual[m];
                }
        }
        else{
            for(int m = begin; m < end; m++){

                child1.individual[m] = parent2.individual[m];
                child2.individual[m] = parent1.individual[m];
            }
        }
        counter++;
    }
    counter = 0;
    //Place children in temporary population
    tempArrayOfIndividuals[countChildren - 2] = child1;
    tempArrayOfIndividuals[countChildren - 1] = child2;
}
else{
    //No crossover
    for (int q = 0; q < TOTAL_STRING_LENGTH; q++ ){
        child1.individual[q] = parent1.individual[q];
```

```java
                child2.individual[q] = parent2.individual[q];
            }
            tempArrayOfIndividuals[countChildren - 2] = child1;
            tempArrayOfIndividuals[countChildren - 1] = child2;
        }
    }

    selected.clear();
    num.clear();
}

public void mutate(Individual individual){
    for (int i = 0; i < TOTAL_STRING_LENGTH; i++){
         if (Math.random() <= MUTATION_PROBABILITY){
             if (individual.individual[i] == 0)
                 individual.individual[i] = 1;
             else
                 individual.individual[i] = 0;
         }
     }
}

    //Apply elitism at end after temp array of individuals has been created
 public void elitism(double[] largestDispl,double[][] elementForceArray, int gen, int eliteNumber){
     //Check whether the fittest individual has a lower fitness of an elite member from a previous generation
     //In the case where the above is true, then a randomly selected member is replaced by the elite member
     Individual elite = findFittestIndividual(tempArrayOfIndividuals,largestDispl,
             elementForceArray, gen);
     double tempFittest = elite.findFitnessFunctionValue(elite, elementForceArray[fittestTempCount],
             density,rvv,ruu, emod, fy, JArray, t, b, ax, largestDispl[fittestTempCount],
             span,gen,strongestMinObjective,weakestMaxObjective,deflectionLimit, stressLimit);
        didElite = false;
        if ( maxFitness > tempFittest) {
            didElite = true;
//              eliteNumber = (int)(Math.random() * POPULATION_SIZE);

            if(eliteNumber == POPULATION_SIZE){
                tempArrayOfIndividuals[eliteNumber - 1] = fittestCurrentIndividual;
                eliteNumber -=1;
            } else{
                tempArrayOfIndividuals[eliteNumber] = fittestCurrentIndividual;
            }
            System.out.print(fittestCurrentIndividual.displace+ ",");
            System.out.print(maxFitness);
            System.out.print(", " + fittestCurrentIndividual.penalty*250);
            System.out.print(", " + fittestCurrentIndividual.mass);
//              System.out.print(", " + fittestCurrentIndividual.maxAllowableForce);
//              System.out.print(" Selected Areas: ");
            for (int i = 0; i < NUMBER_OF_CHROMOSOMES; i++)
            System.out.print(", " + fittestCurrentIndividual.selectedArea[i]);
            System.out.println();
        } else{
            didElite = false;
            System.out.print(findFittestIndividual(tempArrayOfIndividuals, largestDispl,
                        elementForceArray,gen).displace + ",");
            System.out.print(tempFittest);
            System.out.print(", " + findFittestIndividual(tempArrayOfIndividuals, largestDispl,
                        elementForceArray,gen).penalty*250);
            System.out.print(", " + findFittestIndividual(tempArrayOfIndividuals, largestDispl,
                        elementForceArray,gen).mass);
//              System.out.print(", " + fittestCurrentIndividual.maxAllowableForce);
//              System.out.print(" Selected Areas: ");
```

```java
            for (int i = 0; i < NUMBER_OF_CHROMOSOMES; i++)
            System.out.print(", " + findFittestIndividual(tempArrayOfIndividuals, largestDispl,
                        elementForceArray,gen).selectedArea[i]);
            System.out.println();
        }
    }

    public Boolean didElite(){
        return didElite;
    }

    public int getEliteNumber(){
        return eliteNumber;
    }

   public void replace(){
       System.arraycopy(tempArrayOfIndividuals, 0, currentArrayOfIndividuals, 0, POPULATION_SIZE);
   }
}
```

Listing 2: Class Population

Individual

This class creates an array of chromosomes and also contains the code for the fitness and objective functions.

```java
package GeneticAlgorithm;

//import fem.components.element.TrussElement;
//import fem.model.FemModel;
```

```java
//Individual is made up from a number of decodeChromosomesGA
public class Individual {
    int[] individual, arrayDecodedChromosomesGA,femIndexArray;
    int SUBSTRING_LENGTH, TOTAL_STRING_LENGTH, NUMBER_OF_CHROMOSOMES, entries;
    int numberOfElements;
    double[] areaArray ;
    double[] length, JArray;
    double[] selectedArea;
    double maxAllowableForce;
    double individualFitness, objectiveValue, penalty, mass, displace;
    int v;
    Boolean staticFitness, dynamicFitness, normalisedFitness, useSANS, isCHS,
            isEqualL, useDeflectionLimit, findRedundantElements;
    double objective;

    //Individual for FEM
    public Individual(int NUMBER_OF_CHROMOSOMES, int SUBSTRING_LENGTH,
            double[] areaArray, ModelInput input, int entries, double[] length,
            int[] femIndexArray, Boolean staticFitness, Boolean dynamicFitness,
            Boolean normalisedFitness, int numberOfElements, Boolean useSANS,
            Boolean isCHS, Boolean isEqualL, Boolean useDeflectionLimit,
            Boolean findRedundantElements, double[] JArray){
        this.findRedundantElements = findRedundantElements;
        this.useSANS = useSANS;
        this.useDeflectionLimit = useDeflectionLimit;
        this.isCHS = isCHS;
        this.isEqualL = isEqualL;
        this.NUMBER_OF_CHROMOSOMES = NUMBER_OF_CHROMOSOMES;
        this.SUBSTRING_LENGTH      = SUBSTRING_LENGTH;
        TOTAL_STRING_LENGTH        = NUMBER_OF_CHROMOSOMES * SUBSTRING_LENGTH;
        this.entries = entries;
        this.areaArray = areaArray;
        this.JArray = JArray;
        this.numberOfElements = numberOfElements;
        this.femIndexArray = femIndexArray;
        this.staticFitness = staticFitness;
        this.dynamicFitness = dynamicFitness;
        this.normalisedFitness = normalisedFitness;
        individual = new int[TOTAL_STRING_LENGTH];
        //Create an array of decodeChromosomesGA which is an individual
        //in order to decode individual
        arrayDecodedChromosomesGA = new int[NUMBER_OF_CHROMOSOMES];
        selectedArea = new double[NUMBER_OF_CHROMOSOMES];
        //Length create in individual as the length of the members will not change
        this.length = length;

        //Randomly populate individual for first generation
        for (int i = 0; i < individual.length; i++){
            if (Math.random() > 0.5)
                individual[i] = 1;
            else
                individual[i] = 0;
        }
    }


    public void printIndividual(){
        for (int i = 0; i < individual.length; i++)
            System.out.print(individual[i]);
    }

    //Finds the decoded values for an individual
```

```
public int[] decodeChromosomesGA(Individual individual, int entries){
   int intUnknown;

   int real;
   double maxString = Math.pow(2, SUBSTRING_LENGTH) - 1;
   double m = (maxString)*Math.pow(entries,-1);
   double c = m;                                   //y intercept at zero
   int i = TOTAL_STRING_LENGTH - 1;
       for (int j = 0; j < NUMBER_OF_CHROMOSOMES; j++){
           intUnknown = 0;
           real = 0;
           //obtain int value (binary code)
           for (int k = 0; k < SUBSTRING_LENGTH; k++){
               if (i > -1){
                   if (individual.individual[i] == 1){
                       intUnknown += Math.pow(2, k);
                   }
               }
               i--;
           }
           //Check that selected member is in database
           //y = mx+c
           real = (int)((Math.round((intUnknown - c)*Math.pow(m, -1))));

           if (real == entries + 1)
                   real = entries;
           if (real < 0)
               real = 1;
           //REFERS TO ROW NUMBER IN EXCEL SPREAD SHEET
           individual.arrayDecodedChromosomesGA[NUMBER_OF_CHROMOSOMES - j - 1] = real;
```

```
        }
        for (int j = 0; j < individual.arrayDecodedChromosomesGA.length; j++){
        }
    return individual.arrayDecodedChromosomesGA;
}

//m^2
public double[] getAreaIndividual_meters(Individual individual, int entries){
        double[] area = new double[NUMBER_OF_CHROMOSOMES];
        for(int i = 0; i < NUMBER_OF_CHROMOSOMES; i++){
            area[i] = areaArray[individual.decodeChromosomesGA(individual, entries)[i]]*Math.pow(1000000, -1);
        }
    return area;
}

public double[] getJIndividual_meters(Individual individual, int entries){
    double q = 1000;
        double[] J = new double[NUMBER_OF_CHROMOSOMES];
        for(int i = 0; i < NUMBER_OF_CHROMOSOMES; i++){
            J[i] = JArray[individual.decodeChromosomesGA(individual, entries)[i]]*Math.pow((q+1000000000), -1);
        }
    return J;
}

public double findObjectiveFunctionValue(Individual individual, double[] elementForceArray,
        double density, double[] rvv, double[] ruu, double emod, double fy, double[] J,
        double[] t, double[] b, double[] ax, double largestDispl, double span, int gen,
        double deflectionLimit, double stressLimit){

    double al = 0;
    v = 0;
    penalty = 0;
    mass = 0;
    displace = largestDispl;
    maxAllowableForce = 0;
    double penaltyParameter = 250;
    double allowableForce = 0;
    double largestForce = 0;
    boolean tension;
    boolean slender;

    int[] indexArray = new int[NUMBER_OF_CHROMOSOMES];
    int index = 0;          //refers to position in indexArray and selectedArea
    double  deflectionLimitSANS;

    //For the case of SANS

    if (useSANS){
        //create area array
        for ( int q = 0; q < NUMBER_OF_CHROMOSOMES; q++){
                indexArray[q] = individual.decodeChromosomesGA(individual, entries)[q];
                selectedArea[q] = areaArray[indexArray[q]];
        }
        if(isCHS){  //For CIRCULAR HOLLOW SECTIONS

            //check for slenderness
            //Check for tension
            // k = 1 for pinned connections
//          index = 0;
            for(int i = 0; i < numberOfElements; i++){
                //Check current group
                if (i == 0){
                    //First group
                    index = 0;
                }
                else if(femIndexArray[i] != femIndexArray[i-1]){
                    //New group
                    index++;
                }
//
                if (elementForceArray[i]<0){
                    tension = true;
                    if (length[indexArray[index]]*Math.pow(rvv[indexArray[index]], -1) <= 300)
                        slender = false;
                    else{
                        slender = true;
                        penalty = penalty + length[indexArray[index]]*Math.pow(rvv[indexArray[index]], -1)*Math.pow(300,-1)
                            -1;
                    }
                }
                else{
                    if(elementForceArray[i] == 0){
                        System.err.println("---------!Redundant Elements Present!---------");
                        System.err.println("Element number " + i);
                        System.exit(0);
                    }
                    tension = false;
                    if (length[indexArray[index]]*Math.pow(rvv[indexArray[index]], -1) <= 200)
                        slender = false;
                    else{
                        slender = true;
                        penalty = penalty + length[indexArray[index]]*Math.pow(rvv[indexArray[index]], -1)*Math.pow(200,-1)
                            -1;
```

```
          }
        }

        //Allowable force for element
        allowableForce =0;
        if (tension){                    //Tension
//         selectedArea[i] = areaArray[index];
          allowableForce = 0.9*areaArray[indexArray[index]]*fy; //*Math.pow(1000,-1); //kN
//         System.out.println("allowable force tension " + allowableForce);
          if(Math.abs(maxAllowableForce) < Math.abs(allowableForce))
            maxAllowableForce = allowableForce;
        }
        else{                                   //Compression
//         selectedArea[i] = areaArray[index];

          double fey = (Math.pow(Math.PI, 2)*emod)*Math.pow(length[indexArray[index]]*
              Math.pow(ruu[indexArray[index]],-1), -2);

          double r02 = 2 * ruu[indexArray[index]]*
              ruu[indexArray[index]];

          double fez = GAMain.G*J[indexArray[index]]*
              Math.pow(areaArray[indexArray[index]]*r02, -1);

          double omega = 1;

          double fex = (Math.pow(Math.PI, 2)*emod)*Math.pow(length[indexArray[index]]*
              Math.pow(rvv[indexArray[index]],-1), -2);

          double fe;
          if(fex<fey && fex<fez)
```

```
                        fe = fex;
                    else if(fez<fex && fez<fey)
                        fe = fez;
                    else
                        fe = fey;

                    double lamda = Math.sqrt(fy*Math.pow(fe,-1));

                    allowableForce = 0.9*areaArray[indexArray[index]]*fy*
                            Math.pow((1+Math.pow(lamda, 2*1.34)), -1*Math.pow(1.34,-1));//*Math.pow(1000, -1); //kN
                    if(Math.abs(maxAllowableForce) < Math.abs(allowableForce))
                        maxAllowableForce = allowableForce;

                }



                //Stress penalty
                if (Math.abs(largestForce) < Math.abs(elementForceArray[i])){
                        largestForce = Math.abs(elementForceArray[i]);
                    }
                if (Math.abs(elementForceArray[i]) > Math.abs(allowableForce)){  //N
                    penalty = penalty + Math.pow(Math.abs(elementForceArray[i])*
                            Math.pow(Math.abs(allowableForce), -1), 2) - 1;
                    v += 1;
                }
            }
        } else if (isEqualL){     //FOR EQUAL ANGLE SECTIONS
            //check for slenderness
            //Check for tension
            // k = 1 for pinned connections
//          index = 0;
```

```
      for(int i = 0; i < numberOfElements; i++){
        //Check current group
        if (i == 0){
          //First group
          index = 0;
        }
        else if(femIndexArray[i] != femIndexArray[i-1]){
          //New group
          index++;
        }
//
        if (elementForceArray[i]<0){
          tension = true;
          if (length[indexArray[index]]*Math.pow(rvv[indexArray[index]], -1) <= 300)
            slender = false;
          else{
            slender = true;
            penalty = penalty + length[indexArray[index]]*Math.pow(rvv[indexArray[index]], -1)*Math.pow(300,-1)
              -1;
          }
        }
        else{
          if(elementForceArray[i] == 0 && findRedundantElements){
            System.err.println("---------!Redundant Elements Present!---------");
            System.err.println("Element number " + i);
            System.exit(0);
          }
          tension = false;
          if (length[indexArray[index]]*Math.pow(rvv[indexArray[index]], -1) <= 200)
            slender = false;
          else{
              slender = true;
              penalty = penalty + length[indexArray[index]]*Math.pow(rvv[indexArray[index]], -1)*Math.pow(200,-1)
                -1;
            }
          }

          //Allowable force for element
          allowableForce =0;
          if (tension){                                 //Tension
  //           selectedArea[i] = areaArray[index];
            allowableForce = 0.9*areaArray[indexArray[index]]*fy; //*Math.pow(1000,-1); //kN
           if(Math.abs(maxAllowableForce) < Math.abs(allowableForce))
              maxAllowableForce = allowableForce;
//             System.out.println("allowable force tension " + allowableForce);
          }
          else{                                                 //Compression
  //           selectedArea[i] = areaArray[index];

            double fey = (Math.pow(Math.PI, 2)*emod)*Math.pow(length[indexArray[index]]*
                Math.pow(ruu[indexArray[index]],-1), -2);

            double r02 = 2*Math.pow((ax[indexArray[index]] -
                t[indexArray[index]] *
                Math.pow(2, -1)), 2) + rvv[indexArray[index]]*
                rvv[indexArray[index]] + ruu[indexArray[index]]*
                ruu[indexArray[index]];

            double fez = GAMain.G*J[indexArray[index]]*
                Math.pow(areaArray[indexArray[index]]*r02, -1);

            double omega = 1 - (2*Math.pow((ax[indexArray[index]]
        -t[indexArray[index]]*Math.pow(2, -1)), 2))*Math.pow(r02, -1);

double feyz = (fey+fez)*Math.pow(2*omega, -1)*(1-Math.sqrt(1-((4*fey*fez*omega)*Math.pow((fey+fez)*(fey+
    fez), -1))));

double fex = (Math.pow(Math.PI, 2)*emod)*Math.pow(length[indexArray[index]]*
        Math.pow(rvv[indexArray[index]],-1), -2);

double fe;
if(fex<feyz)
    fe = fex;
else
    fe = feyz;

double lamda = Math.sqrt(fy*Math.pow(fe,-1));

allowableForce = 0.9*areaArray[indexArray[index]]*fy*
        Math.pow((1+Math.pow(lamda, 2*1.34)), -1*Math.pow(1.34,-1));//*Math.pow(1000, -1); //kN


if(Math.abs(maxAllowableForce) < Math.abs(allowableForce))
    maxAllowableForce = allowableForce;

//Check the class of the section
//Implement area reduction for class 3 sections

if ((b[indexArray[index]]*Math.pow(t[indexArray[index]], -1))
     <= (200*Math.pow(Math.sqrt(fy), -1))){     //if condition holds, then class 3
     double refArea;
     double W = b[indexArray[index]] * Math.pow(t[indexArray[index]], -1);
     double f = allowableForce * Math.pow((0.9*areaArray[indexArray[index]]), -1);
```

```
                double Wlim = 0.644 * Math.sqrt((0.43*210000) * Math.pow(f,-1));
                if (W <= Wlim){
                    refArea = areaArray[indexArray[index]];
                } else{
                    double newB = 0.95* t[indexArray[index]] * Math.sqrt((0.43*210000) * Math.pow(f,-1)) *
                            (1-(0.208*Math.pow(W, -1)) * Math.sqrt((0.43*210000) * Math.pow(f,-1)));
                    refArea = areaArray[indexArray[index]] - (b[indexArray[index]] - newB) * t[indexArray[index]];
                }
                allowableForce = 0.9 * refArea * f;
            }

        }

        //Stress penalty
        if (Math.abs(largestForce) < Math.abs(elementForceArray[i])){
              largestForce = Math.abs(elementForceArray[i]);
            }
        if (Math.abs(elementForceArray[i]) > Math.abs(allowableForce)){  //N
            penalty = penalty + Math.pow(Math.abs(elementForceArray[i])*
                    Math.pow(Math.abs(allowableForce), -1), 2) - 1;
            v += 1;
        }
    }
}

for (int j = 0; j < individual.numberOfElements; j++){
    al = al + selectedArea[femIndexArray[j]] * length[j];
}

//Displacement penalty
if (useDeflectionLimit){
```

```
                deflectionLimitSANS = Math.abs(deflectionLimit);
            } else {
                deflectionLimitSANS = span*Math.pow(180, -1);
            }

            if (Math.abs(largestDispl) > deflectionLimitSANS){
                penalty = penalty + Math.pow(Math.abs(largestDispl)*Math.pow(deflectionLimitSANS, -1), 2)-1;
//                System.out.print(" penalty " + penalty);
                v += 1;
            }
        }

        //For the case of prescribed constriants
        else{
             //create statement for when force == 0
        index = 0;
            for(int i = 0; i < NUMBER_OF_CHROMOSOMES; i++){
                index = individual.decodeChromosomesGA(individual, entries)[i];
                if (elementForceArray[i]<0){
                    tension = true;
                }
                else{
                    tension = false;
                }

                //Allowable force for element
                allowableForce =0;
                if (tension){
                    selectedArea[i] = areaArray[index];
                    allowableForce = stressLimit*areaArray[index]*(-1);
                }
        else{   //Compression
            selectedArea[i] = areaArray[index];
            allowableForce = stressLimit*areaArray[index];
        }

        //Stress penalty
        if (Math.abs(largestForce) < Math.abs(elementForceArray[i])){
                largestForce = Math.abs(elementForceArray[i]);
            }
        if (Math.abs(elementForceArray[i]) > Math.abs(allowableForce)){
            penalty = penalty + Math.pow(Math.abs(elementForceArray[i])*
                    Math.pow(Math.abs(allowableForce), -1), 2) - 1;
            v += 1;

        }
    }

    for (int j = 0; j < individual.numberOfElements; j++){
        al = al + selectedArea[femIndexArray[j]] * length[j];
    }

    //Displacement penalty
    if (Math.abs(largestDispl) > deflectionLimit){
        penalty = penalty + Math.pow(Math.abs(largestDispl)*Math.pow(deflectionLimit, -1), 2)-1;
        v += 1;
    }
}

//Fitness function
    mass = density * Math.pow(1000000000,-1) * al;
```

```
//              for (int p = 0; p < NUMBER_OF_CHROMOSOMES; p++){
//                  System.out.println(selectedArea[p]);
//              }
//              for (int p = 0; p < NUMBER_OF_CHROMOSOMES; p++){
//                  System.out.println(length[p]);
//              }
//              System.out.println("forces = " );
//              for (int p = 0; p < NUMBER_OF_CHROMOSOMES; p++){
//                  System.out.println(elementForceArray[p]*Math.pow(1000, -1)); //kN
//              }


            objectiveValue = (density  * Math.pow(1000000000,-1) * al + gen*penaltyParameter*penalty);    //Mass (Defined as
                weight in literature)
        return objectiveValue;
    }

        //This method uses the array and only accesses the database once throughout the search
    public double findFitnessFunctionValue(Individual individual, double[] elementForceArray,
            double density, double[] rvv, double[] ruu, double emod, double fy,
            double[] J, double[] t, double[] b, double[] ax, double largestDispl, double span,
            int gen, double strongestMinObjective, double weakestMaxObjective,
            double deflectionLimit, double stressLimit){
        //The fitness of a solution is determined by its weight, Sum of {pAL} over all the elements
        //Penalty is added to this fitness function to take into account all constraints
        //The objective is to find the min weight of the structure
        //The fitness is to maximise max weigth - objective
        //Therefore the greater the fitness, the less the weight of the structure

        objective = individual.findObjectiveFunctionValue(individual, elementForceArray,
                density, rvv, ruu, emod, fy, J, t, b, ax, largestDispl, span, gen,
                deflectionLimit, stressLimit);
        if (normalisedFitness){
            individualFitness = 1*Math.pow((objective*(1000*v+1)), -1);     //Coello
        } else if (dynamicFitness){
            individualFitness = strongestMinObjective + weakestMaxObjective - objective;
        } else {
            individualFitness = 1000000000 - objective;                         //Goldberg
        }

        return individualFitness;
    }

}
```

Listing 3: Class Individual

Truss Population

This class creates a population of trusses (femModels) based on the information provided by the algorithm. It serves as some interface between the finite element analysis program and the genetic algorithm.

```
package GeneticAlgorithm;

import aho.math.linalg.Matrix;
import aho.math.linalg.Vector;
import fem.analysis.Analysis;
import fem.analysis.FirstOrderLinearAnalysis;
import fem.calculation.Dof;
import fem.components.CoordinateSystem;
import fem.components.CrossSection;
import fem.components.Material;
```

```java
import fem.components.Node;
import fem.components.Support;
import fem.components.element.FrameElement;
import fem.components.element.TrussElement;
import fem.components.load.LoadCase;
import fem.components.load.NodeLoad;
import fem.components.load.VolumeLoadSet;
import fem.model.FemModel;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Map;

public class TrussPopulation {

    FemModel trussModel;
    Analysis analysis;
    //NodalInput input;
    double material, poison, density, emod;
    int numberOfNodes, numberOfElements;
    Individual[] individuals;
    int entries;
    FemModel[] trussmodelArray;
    String[] dof;
    String[][] elementString;
    double[] nodesX, nodesY, nodesZ;
    LinkedList<TrussElement> elts;
    LinkedList<FrameElement> frameElts;
    //first area array only has chromosome number of entries,
//femArray has an element number of entries
double[] areas, femAreas, femJ;;
double[][] load;
Boolean isGrouped, is3D, isFrameElement;

public TrussPopulation(ModelInput input, Double emod, Double poison, Double density,
        Individual[] individuals, int entries,int numberOfElements,
        double[][] load, double[] nodesX, double[] nodesY, double[] nodesZ,
        String[] dof, String[][] elementString, Boolean isGrouped, Boolean is3D,
        Boolean isFrameElement) {
    analysis = new FirstOrderLinearAnalysis();
    this.isGrouped = isGrouped;
    this.is3D = is3D;
    this.isFrameElement = isFrameElement;
    //this.input      = input;
    this.emod = emod * Math.pow(10, 6);                              //Convert to Pa
    this.poison = poison;
    this.density = density;
    this.numberOfNodes = input.getNumberOfNodes();
    this.individuals = individuals;
    this.entries = entries;
    this.numberOfElements = numberOfElements;
    this.load = load;                                                      //N
    this.nodesX = nodesX;                                                  //m
    this.nodesY = nodesY;
    this.nodesZ = nodesZ; //m
    this.dof = dof;
    this.elementString = elementString;
    trussmodelArray = new FemModel[individuals.length];
    elts = new LinkedList<TrussElement>();
    frameElts = new LinkedList<FrameElement>();
```

```
}

public LinkedList<TrussElement> getElements(FemModel trussModel) {
    Iterator<TrussElement> iter = trussModel.iterator(TrussElement.class);
    while (iter.hasNext()) {
        elts.add(iter.next());
    }
    return elts;
}

public LinkedList<FrameElement> getElementsFrame(FemModel trussModel) {
    Iterator<FrameElement> iter = trussModel.iterator(FrameElement.class);
    while (iter.hasNext()) {
        frameElts.add(iter.next());
    }
    return frameElts;
}
//Create map for translations and rotations
private static Map<String, Dof> supportDOFs = new HashMap<String, Dof>();

static {
    supportDOFs.put("X_TRANSLATION", Dof.X_TRANSLATION);
    supportDOFs.put("Y_TRANSLATION", Dof.Y_TRANSLATION);
    supportDOFs.put("Z_TRANSLATION", Dof.Z_TRANSLATION);

    supportDOFs.put("X_ROTATION", Dof.X_ROTATION);
    supportDOFs.put("Y_ROTATION", Dof.Y_ROTATION);
    supportDOFs.put("Z_ROTATION", Dof.Z_ROTATION);

    supportDOFs.put("XY_TRANSLATION", Dof.XY_TRANSLATION);
    supportDOFs.put("YZ_TRANSLATION", Dof.YZ_TRANSLATION);
```

```
        supportDOFs.put("XZ_TRANSLATION", Dof.XZ_TRANSLATION);
        supportDOFs.put("XY_TRANSLATION", Dof.XY_TRANSLATION);

        supportDOFs.put("ALL_TRANSLATION", Dof.ALL_TRANSLATION);
        supportDOFs.put("ALL_ROTATION", Dof.ALL_ROTATION);
        supportDOFs.put("ALL", Dof.ALL);
    }

    public FemModel[] createFemModels(int[] femIndexArray, int[] femElementArray, int numberOfElements) {
        String trussName;

            for (int i = 0; i < individuals.length; i++) {
                trussName = "truss" + (i + 1);
                trussmodelArray[i] = createTruss(individuals[i], trussName,
                        femIndexArray, femElementArray, numberOfElements);
            }

        return trussmodelArray;
    }

    //createTruss for every femTrussEntry in population
    public FemModel createTruss(Individual gaIndividual, String name,
            int[] femIndexArray, int[] femElementArray, int numberOfElements) {
        String nodeName;
        String elementName;
        String elementNodeName1;                //Define the nodes at ends of elements as per user defined
        String elementNodeName2;
        String loadName;
        String vectorLoad;
        String supportCondition, supportName;
        String crossSectionName;
```

```
trussModel = new FemModel(name);
LoadCase lc = new LoadCase("User Defined");

//Nodes and supports for each femTrussEntry's truss
for (int i = 0; i < numberOfNodes; i++) {
    nodeName = "n" + (i + 1);

    //Add nodes
    trussModel.add(new Node(nodeName, new double[]{nodesX[i], nodesY[i], nodesZ[i]}));

    //Loading in all 3 directions:
    for (int k = 0; k < 3; k++){
        loadName = "load" + (i + 1) + "," + (k);  //Load referencing load(node)(direction)
        vectorLoad = "{" + load[i][k] + "}";
        //Add loads
        if(k == 0)
            trussModel.add(new NodeLoad(loadName, nodeName, Vector.getVector(vectorLoad),
            Vector.getVector("{1,0,0}"), CoordinateSystem.GLOBAL_COORDINATE_SYSTEM));
        if(k == 1)
            trussModel.add(new NodeLoad(loadName, nodeName, Vector.getVector(vectorLoad),
            Vector.getVector("{0,1,0}"), CoordinateSystem.GLOBAL_COORDINATE_SYSTEM));
        if(k == 2)
            trussModel.add(new NodeLoad(loadName, nodeName, Vector.getVector(vectorLoad),
            Vector.getVector("{0,0,1}"), CoordinateSystem.GLOBAL_COORDINATE_SYSTEM));
    lc.add(loadName);
    }
    trussModel.add(lc);

    //Add supports
    supportCondition = dof[i];
    if (supportCondition != null) {
        supportName = "s" + (i + 1);
        trussModel.add(new Support(supportName, nodeName, supportDOFs.get(supportCondition)));
    }
}

//Add material
trussModel.add(new Material("mat", emod, poison, density));

//CHROMOSOMES VALUES FROM GA MUST GO HERE:
areas = new double[gaIndividual.NUMBER_OF_CHROMOSOMES];
areas = gaIndividual.getAreaIndividual_meters(gaIndividual, entries);
double[] J = new double[gaIndividual.NUMBER_OF_CHROMOSOMES];
J  = gaIndividual.getJIndividual_meters(gaIndividual, entries);
if (isGrouped){
    //Duplicate group elements
    femAreas = new double[numberOfElements];
    femJ = new double[numberOfElements];
    for (int i = 0; i < femAreas.length; i++){
        femAreas[femElementArray[i] - 1] = areas[femIndexArray[i]];
        femJ[femElementArray[i] - 1] = J[femIndexArray[i]];
    }
}
if(isFrameElement){
    if (isGrouped){
        for (int i = 0; i < numberOfElements; i++) {
            crossSectionName = "section" + (i + 1);
            trussModel.add(new CrossSection(crossSectionName, femAreas[i], 0.000000001, 0.000000001, femJ[i]));

        //Add truss elements
            elementName = "t" + (i + 1);
```

```
                crossSectionName = "section" + (i + 1);
                elementNodeName1 = "n" + elementString[i][0];                //Define the nodes at ends of elements as per
                    user defined
                elementNodeName2 = "n" + elementString[i][1];
                trussModel.add(new FrameElement(elementName, new String[]{elementNodeName1, elementNodeName2},
                        "mat", crossSectionName, is3D));
            }
        }   else{
            for (int i = 0; i < gaIndividual.NUMBER_OF_CHROMOSOMES; i++) {
                crossSectionName = "section" + (i + 1);
                trussModel.add(new CrossSection(crossSectionName, areas[i], 0.000000001, 0.000000001, femJ[i]));

            //Add truss elements
                elementName = "t" + (i + 1);
                crossSectionName = "section" + (i + 1);
                elementNodeName1 = "n" + elementString[i][0];                //Define the nodes at ends of elements as per
                    user defined
                elementNodeName2 = "n" + elementString[i][1];
                trussModel.add(new FrameElement(elementName, new String[]{elementNodeName1, elementNodeName2},
                        "mat", crossSectionName, is3D));
            }
        }
    }else{
        if (isGrouped){
            for (int i = 0; i < numberOfElements; i++) {
                crossSectionName = "section" + (i + 1);
                trussModel.add(new CrossSection(crossSectionName, femAreas[i], 0, 0));

            //Add truss elements
                elementName = "t" + (i + 1);
                crossSectionName = "section" + (i + 1);
                elementNodeName1 = "n" + elementString[i][0];              //Define the nodes at ends of elements as per
                    user defined
                elementNodeName2 = "n" + elementString[i][1];
                trussModel.add(new TrussElement(elementName, new String[]{elementNodeName1, elementNodeName2},
                        "mat", crossSectionName, is3D));
            }
        }   else{
            for (int i = 0; i < gaIndividual.NUMBER_OF_CHROMOSOMES; i++) {
                crossSectionName = "section" + (i + 1);
                trussModel.add(new CrossSection(crossSectionName, areas[i], 0, 0));

            //Add truss elements
                elementName = "t" + (i + 1);
                crossSectionName = "section" + (i + 1);
                elementNodeName1 = "n" + elementString[i][0];              //Define the nodes at ends of elements as per
                    user defined
                elementNodeName2 = "n" + elementString[i][1];
                trussModel.add(new TrussElement(elementName, new String[]{elementNodeName1, elementNodeName2},
                        "mat", crossSectionName, is3D));
            }
        }
    }
    return trussModel;
}

public void clearTrussModel(FemModel trussModel){
    trussModel.remove(trussModel);
}


//pass method models seperately
```

```
    public void analyse(FemModel trussModel) {
        analysis.perform(trussModel);
        trussModel.clearModelListeners();
    }

}
```

Listing 4: Class Truss Population