Genetic Optimisation of Structural Systems

Author:
GAVIN S. REYNOLDS

Supervisor:
DR. CHRIS PEARCE

A thesis submitted in partial fulfilment of the requirements for the degree of

BEng Civil Engineering

DEPARTMENT OF CIVIL ENGINEERING

THE UNIVERSITY OF GLASGOW

March 2009

Abstract

This work summarises the development of a Genetic Algorithm and its use in an investigation into the optimisation of structural systems. The Genetic Algorithm detailed in this work was implemented from scratch by the author in MATLAB.

An optimisation approach to structures is the inverse of more conventional structural design approaches. For example, considering a truss system, a conventional approach would be to design the structure, analyse it under given loads and manually redesign if design requirements were violated. An optimisation approach inverts this, by instead allowing the designer to pose the question "what **should** the design be, in order to optimally satisfy the requirements placed upon the structure?".

Many optimisation methods operate in a point to point fashion, evaluating a single point and then using some method or rule to determine the next point to be evaluated, thereby hopefully moving closer to the optimal solution with each iteration. This method is particularly prone to locating local optima. Genetic Algorithms (GAs) work with a population of diverse points simultaneously, increasing the likelihood of locating the global optimum of a given function. GAs are a heuristic global optimisation technique which draw inspiration from evolutionary biology and the Darwinian principle of Survival of the Fittest. By assigning an objective fitness value to each member of a population, fit individuals can be identified and used to evolve even fitter solutions. GAs use mechanisms similar to those which exist in evolutionary biology such as reproduction, inheritance, selection, crossover and mutation in order to evolve the optimum solution to a problem.

This report describes the theory and development of a GA. The generic nature of the Genetic Algorithm developed in this work enables it to be applied to new problems with minimal extra development. The application of the GA to the optimisation of two structural problems is presented herein: a steel truss system and a simply supported reinforced concrete beam. The effects of the genetic mechanisms in optimisation were investigated as well as the roles of the fitness evaluation functions. This work shows that Genetic Algorithms are a robust and efficient method for optimisation through their ability to quickly identify optimum solutions to a given problem.

Acknowledgements

I would like to thank Chris Pearce for his constant support, direction and enthusiasm throughout this project. His guidance & ideas challenged me to strive further in the development of this work.

This thesis is typeset using open source software: the LyX document processor and the LaTeX 2_ε document markup language for the TeX typesetting system.

Contents

List of Algorithms

List of Figures

List of Tables

Part I. Introduction

In applied mathematics, optimisation is the process of finding the maxima or minima of a given function, within a domain of acceptable input values. Optimisation has a long history and many techniques exist, such as simple Gauss gradient descent to more complicated techniques like Newton-Raphson and Sequential quadratic programming. Such techniques are greatly used in areas such as mathematics, applied sciences, economics and operational research.

An optimisation approach to structures is the inverse of more conventional structural design approaches. For example, considering a truss system, a conventional approach would be to design the structure, analyse it under given loads (for the deflections, stresses and moments) and manually redesign if design requirements, such as maximum deflection, were violated. An optimisation approach inverts this, by asking "what **should** the design be to optimally satisfy the requirements placed upon the structure?".

Rarely in structural engineering is a problem unconstrained by requirements. Again consider a structural design problem, such as minimising the weight of a truss. An unconstrained optimisation would result in a truss with the lightest members available, with no consideration given to the ability of the structure to support itself or carry loads. However, typically the objective of structural design is to find the most economical (the cheapest and therefore generally lightest) structure which is both safe & serviceable. This can be achieved by applying constraints to an optimisation, such as limiting overall deflection and stresses in members. The nature of any optimisation technique used for structural problems must cater for four main requirements [1]. The technique should:

1. be easily extended to optimise real structural design problems, as opposed to formulated benchmark problems;
2. use a minimum of auxiliary information to find an optimum solution - such as function derivatives - and preferably none at all;
3. attempt to find the global optimum and avoid local optima;
4. be able to solve problems with discrete variables.

Most classical mathematical programming techniques require auxiliary information such as the first derivative of a given function in order to find optima. Therefore such techniques cannot be used for functions which cannot be differentiated. These techniques are also essentially mathematical hill climbers and often tend to find local optima rather than global optima. Additionally many of these mathematical techniques require the search space to be continuous. They also require an equation to model a problem which is often impossible to formulate, especially in the case of discrete variables with complicated variable interaction. Hence for a problem which cannot be differentiated, with many local optima, or discrete variables - or indeed all three of these characteristics - traditional mathematical optimisation cannot be applied.

Finding a local optimum is relatively easy; however, finding global optima is more challenging. A simple, but highly inefficient technique, would be to evaluate the function at every available point - a process that may take some considerable time. Alternatively, the function could be evaluated at certain intervals or sampled at random,

however there is no guarantee of finding the global optimum with such an approach - especially in rapidly changing or chaotic functions. Many optimisation methods operate in this fashion, evaluating a single point and then using some method or rule to determine the next point to be evaluated. This point to point method is particularly prone to locating local optima.

For this reason a branch of applied numerical analysis called Global Optimisation has been developed to solve such problems. Within Global Optimisation there are a large number of search techniques which fall into several categories: deterministic, stochastic and heuristic. However the most powerful global search techniques use primary knowledge of the search space to inform their search i.e. heuristics. Genetic Algorithms, a global search heuristic technique, have been chosen for investigation in this paper as they not only satisfy the four requirements above they are also computationally simple, relatively easy to implement but exceptionally robust & powerful in their search for global optima. Genetic Algorithms do not work on a point to point basis, but operate instead on a set of points (known as a population) simultaneously. This reduces the likelihood of locating a local optima and by working on a population of diverse points, the Genetic Algorithm adheres to the adage that there is “safety in numbers”. This in part contributes to the robust & powerful nature of a Genetic Algorithm, as well as its inherent parallelism - by working on many solutions simultaneously [2, 3].

1 Genetic Algorithms

In Biology, evolution is the process by which a species improves itself over generations. At the most basic level, offspring inherit genetic traits from their parents in processes associated with reproduction. The genetic makeup of an individual of a species is stored in its DNA and it consists of a sequence of molecules - called bases - attached to double helix of long polymers, which provide its structure. Each base can be one of four types, which conventionally labelled A,C, G and T - and are stored in pairs at opposite positions on the double helix. At a simple level DNA can be thought of as a string which encodes complicated information using a long string consisting of pairs of these four types. To give an example of the length of a DNA string, human DNA is of the order of 3 billion base pairs.

During reproduction, an offspring's genes are created from the DNA of its parents via a combination of genetic crossover and mutation. Crossover combines the genetic material of the two parents and mutation introduces random variations in the genes of the offspring. Mutation introduces a random element of change into evolution and creates variation in the gene pool by introducing new novel genetic features by slightly modifying the DNA, which may or may not improve the fitness of an individual. If any resulting change from either crossover or mutation does improve the fitness of an individual, then they are more likely to survive to reproduce and consequently pass its DNA onto its offspring. The process of selection, in the form of survival of the fittest, ensures that genetic traits which improve fitness are more likely to survive to the next generation, whereas those which are ineffective or harmful are less likely to.

Genetic Algorithms are a global search method using evolutionary algorithms and are a subset of evolutionary computation, which is in turn a sub-field of artificial intelligence research. Evolutionary algorithms are inspired by evolutionary biology and Darwinian principles such as Survival of the Fittest. They use mechanisms similar to those which exist in biology such as reproduction, inheritance, selection, crossover and mutation in order to find the optimum solution to a problem. They require no auxiliary knowledge such as derivatives or other gradient information and are suited to solving problems involving discrete variables & many local optima.

The concept of GAs was originally developed by John Holland, of the University of Michigan, in the 1970s in the seminal book "Adaption in Natural and Artificial Systems" . Holland drew inspiration from the robustness, efficacy and efficiency of the natural processes. He believed that reproducing such processes would produce a technique for solving difficult problems in a robust manner [4, 2].

GAs differ in fundamental ways from other optimisation and search methods [5]:

1. GAs work with an abstract representation of the problem parameters, not the parameters themselves (i.e. a coding of the parameters).
2. GAs search from a population of solutions, alternatively called individuals or chromosomes.
3. GAs use an objective fitness function to drive the optimisation, rather than other auxiliary information such as derivatives.
4. GAs use stochastic - as opposed to deterministic - methods. However they are not random searches - they use random probabilistic choice as a tool to guide the optimisation process.

In his “Handbook of Genetic Algorithms” [2], Davis remarked that:

> "Evolution is a process that operates on chromosomes rather than on the living beings they encode."

Evolution operates blindly using elementary operations to manipulate the genetic material of a chromosome, to achieve exceptionally complicated life. Genetic Algorithms operate on a similar principle, using simple encoding and reproduction processes to achieve sophisticated behaviour.

Biological evolution is infinitely more complex than that modelled in a Genetic Algorithm. However, at least to some extent, genetic operations such as crossover, mutation and reproduction follow their biological counterparts. As stated, DNA uses a quaternary alphabet (A, C, G, T) to encode the instructions used to create living organisms. At this juncture, it is necessary to accept that Genetic Algorithms also work on the basis of codings, the simplest of which - and easiest to understand - is binary encoding. There are a number of other encodings used with GAs - which are less obvious - and these will be discussed later in Section 4.1 on page 18 [2, 6].

1.1 Crossover

Biological crossover, or recombination, is a process by which two chromosomes fuse together at a random point and then split at the join, exchanging their DNA by swapping the corresponding sections of material after the join. This results in two new chromosomes which differ from their parents. In genetics this occurs during meosis - the process by which cells divide.

GAs implement an operation which is analogous to the biological process. Pairs of chromosomes are selected at random from the mating pool and for each pair of strings, a random crossover point is selected. The mating pool for the next generation is selected from the current generation with the aid of the objective evaluation of the fitness of the members of that generation. The concepts of Fitness and Selection are covered in Sections 4.2 on page 21 and respectively.

A simple one-point crossover example is presented below.

Consider two binary chromosomes of 10 bits. $(l = 10)$

Chromosome A: 1111010111
Chromosome B: 0011111000

Select a random number between 1 and $(l - 1)$. Say $l = 6$. The crossover point is denoted by the pipe symbol (|).

Chromosome A: 111101 | 0111
Chromosome B: 001111 | 1000

The resulting crossed over chromosomes are shown below.

Chromosome AB: 1111011000
Chromosome BA: 0011110111

The resulting hybrid chromosomes will replace their parents in the population. Through a combination of crossover and survival of the fittest (otherwise known as Selection in

a GA) as each generation passes, the total fitness of a population will trend upwards, as selection will favour fitter chromosomes. However, with selection and crossover there will come a point where no further improvement in the total fitness of a population is possible. Nevertheless, this does not imply that the optimum solution has been found.

Survival of the fittest and crossover on their own will tend to cause the population to become homogeneous over time. This is because there is no mechanism to create new novel genetic features. The genetic features (i.e. binary bits) present in a population after a number of generations will only be composed of those which existed in the initial population. It is possible that one or several highly fit individuals could predominate causing crossover to be ineffective and produce no change. For example, crossing a pair of similar chromosomes may have very little effect and will certainly have no effect at all if both parents are the same (since any crossover will simply swap identical bits).

1.2 Mutation

To overcome the limitations of crossover an additional mechanism for change is required. In Biology, alterations to the gene sequence of an organism are caused by a number of factors, including errors in cell division and electromagnetic radiation. Despite the seemingly accidental nature of mutation, it can provide new genetic traits, which reproduce if they are advantageous to the species (or are lost if they are not).

In Genetic Algorithms, mutation is required since, although crossover and selection effectively recombine existing genetic traits, they do not protect potentially useful genetic material from being lost (i.e. individual bits at particular positions). Nor can they introduce new novel features into the gene pool i.e. genetic material which did not exist in the original population. Mutation is generally implemented as a probability that a given random binary bit in the genetic sequence will be inverted i.e. from 0 to 1 and vice versa.

It is noteworthy that mutation by itself provides no advantage over a random search. However, when combined with crossover and selection, it ensures that genetic diversity is maintained within the mating pool to avoid a homogeneous population. Consequently, mutation plays a secondary role in Genetic Algorithms[2].

2 Development Software

2.1 MATLAB

MATLAB, a contraction of “Matrix Laboratory”, is a cross-platform numerical & technical computing environment written in C and is a proprietary commercial product of The Mathworks, Inc.

The Mathworks website describes MATLAB as:

> ”... a high-level technical computing language and interactive environment for algorithm development, data visualisation, data analysis, and

> numeric computation. Using the MATLAB product, you can solve technical computing problems faster than with traditional programming languages, such as C, C++, and Fortran... This is because MATLAB offers capabilities such as matrix-based programming, advanced mathematical functions, and a simplified, flexible language that lets you focus more on solving problems and less on programming." [7]

It provides a high level base upon which engineers, scientists and mathematicians can rapidly develop algorithms and programs. Traditional programming training or knowledge of the workings of computer systems - although preferable - is not required. In common with all high level programming languages, there is strong level of abstraction from the technical details of the computer platform in use. Many mathematical functions are inbuilt, reducing the need to reinvent the wheel to implement common or advanced mathematical functionality. Additionally, MATLAB provides better handling of vectors, matrices, multi-dimensional arrays as well as other mathematical concepts than other general programming languages.

However, there a number of disadvantages to using MATLAB. It is a proprietary commercial product and as such is subject to hefty fees & restrictive licensing. (However, there is a freely available GNU General Public License - i.e. open source software - alternative named Octave that is mostly compatible with MATLAB's programming language M-code).

Additionally, unlike other programming languages where code executing on two different machines would reasonably be expected to execute in a similar manner, there is a level of ambiguity in MATLAB. This is due to its lack of a modern package architecture and reliance on the user setting the correct path. All functions share the same global name-space and it uses the definition of the users path to resolve conflicts between identically named functions. MATLAB is also an interpreted - as opposed to compiled - language and poor programming practises or inefficient code will inevitably cause slow performance - more so than with most modern compiled languages.

2.2 CALFEM

A common method for structural analysis is Finite Element Modelling. As this paper is concerned with the optimisation of structural systems, a method of structural analysis was required to inform the objective numerical evaluation made of an individual by the fitness function. See the section on Fitness Functions (section 4.2 on page 21) for further details.

Rather than reinventing the wheel and creating custom structural analysis solvers from scratch, off-the-shelf finite element modelling software was required. As the programming language used in this work is MATLAB, a compatible (or at least interfaceable) FEM program was required and for this purpose software called CALFEM was selected. CALFEM is a finite element toolbox written in MATLAB developed at the Division of Structural Mechanics, Lund University, Sweden [8].

> "CALFEM is an interactive computer program for teaching the finite element method (FEM). The name CALFEM is an abbreviation of "Computer Aided Learning of the Finite Element Method". The program can be used for different types of structural mechanics problems and field problems."

Whilst the use of an FEM package with an API (Application Programming Interface), such as Strand7, would have been possible, the use of a natively written MATLAB toolbox was chosen as this vastly simplifies programming. By using a fully MATLAB solution there are no outside dependencies (i.e. on other software) therefore the program only requires MATLAB to run..

3 Objectives

> "Genetic algorithms for optimisation of structural systems."

The purpose of this work is to develop a Genetic Algorithm for use in an investigation into the optimisation of structural systems. The primary objectives of this paper are:

- the development of a bespoke problem independent Genetic Algorithm using MATLAB.
- the development of problem specific code to allow the developed Genetic Algorithm to solve benchmark, proof of concept problems.
- to investigate and evaluate the appropriateness of using Genetic Algorithms for optimisation of structural systems.

To facilitate this investigation, two problems were considered: a ten bar truss and a simply supported reinforced concrete beam. These two different problems were selected to enable investigation into a wide range of aspects:

- the performance of Genetic Algorithms in the optimisation of different problems;
- the effect of the parameters of a Genetic Algorithm on the optimisation process: performance, efficiency and efficacy;
- and the ability of Genetic Algorithms to optimise different problem types.

Through evaluation of the results, the effectiveness of Genetic Algorithms for the optimisation of structural systems can be commented upon.

Additionally, the question is posed as to whether Genetic Algorithms could be genuinely useful to an engineer and form part of their design toolbox - similar to the invaluable contribution Finite Element Modelling has made to modern engineering.

The principle points for investigation are annotated on a full Genetic Algorithm flowchart in Appendix A on page 59. In Part II the development of the GA is split into its logical components and the investigative points related to its design are discussed, such as how to encode a problem and how to evaluate the fitness of a given solution. In Part III on page 39 the effect of the parameters of a Genetic Algorithm and the design of Fitness Functions on the optimisation process are examined.

Part II. Development

4 The Genetic Algorithm

Genetic Algorithms have a relatively long history in computing terms - almost as long as the microprocessor and UNIX. Although Evolutionary Algorithms date back to the mid 1950's, Genetic Algorithms were not invented until the mid 1970's by John Holland.

GAs remained largely a theoretical academic research interest until the late 1980s when the first genetic algorithms were used for industrial purposes. Today there are many implementations, in the form of standalone programs as well as numerous libraries in many programming languages.

At a particularly high level, genetic algorithm can be characterised into two processes: the initial generation of a random population and the improvement of that population. The improvement of the population takes place through reproduction - the process of breeding a fitter generation. This involves evaluation, selection and genetic operations such as crossover & mutation.

The classic high level pseudo-code and flowchart for GAs is presented in Algorithm 1 and Figure 1.

The Genetic Algorithm upon which this paper is based was entirely self-developed from scratch and with no reference to existing implementations or code. It was developed in MATLAB M-code by the author, without programming assistance, based on the classic high level pseudo-code for GAs.

The implementation of a Genetic Algorithm in this paper is intentionally problem agnostic and modular. This ensures that the algorithm can be applied to any problem, as well as being easier to develop & maintain. It is split into a number of procedural functions along logical lines: one overall controller and functions for each genetic operator. All relevant GA source code is provided in Appendix D on page 65.

- GA_Controller: The main function implementing the standard Genetic Algorithm pseudo-code, controlling all aspects of program.
- GA_Crossover: An implementation of Genetic Crossover, producing offspring from two parents by randomly selecting a crossover point and swapping the parents bit-strings from this point.
- GA_Mutation: An implementation of Genetic Mutation, randomly inverting a specified percentage of bits in the population of individuals.
- GA_Elitism: Selects a specified percentage of the fittest individuals to preserve for the next generation.
- GA_SelectionRouletteWheel: Uses the roulette wheel selection paradigm to select individuals from population.

A Genetic Algorithm - with its operators of crossover, mutation, selection etc - is generic procedural code and is independent of the problem domain. However, at this point Genetic Algorithms may seem entirely inapplicable to real engineering design,

Algorithm 1 Basic Genetic Algorithm Pseudo-code

1. Generate initial random population
2. Evaluate the fitness of each individual of the population
3. Repeat the following until termination
   (a) Select best ranked individuals to reproduce
   (b) Breed new generation via genetic operations (e.g. crossover & mutation)
   (c) Evaluate the fitness of each individual of the new population

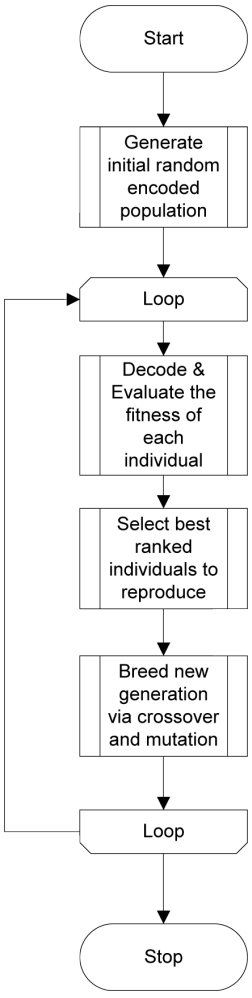

Fig. 1: GA Flowchart

more concerned with the manipulation of strings than the optimisation of structural systems.

In the words of Dr. Charlie Ellis of the University of Plymouth:

> "Whilst all this makes an incontrovertibly splendid party trick, it might seem to be rather far removed from the use of GAs for solving real problems." [9]

Nevertheless, it is relatively trivial to use a genetic algorithm to solve a specific problem. Two critical problem dependent requirements must be satisfied to apply a GA to a problem: encodability and evaluability. The first requirement, **encodability**, implies that each design can be uniquely represented by a coding scheme; the second requirement, **evaluability**, implies that the relative fitness of an individual can be represented by an objective numerical value. In a Genetic Algorithm these requirements are implemented respectively in the Encoding and Fitness Functions for a given problem. The evaluation of the fitness of an individual is then used to influence the genetic reproductive process, i.e. the method of **selection [9, 4]**.

The concepts below are discussed in the following sections:

- encoding a problem,
- evaluating a potential solution to a problem using fitness functions,
- selection of the fittest solutions
- and the termination of the optimisation process.

4.1 Encoding

One of the major stumbling blocks for the use of Genetic Algorithm is the need to genetically encode the search space so that the GA can use them. Each individual chromosome, or design in the case of structural optimisation,should represent a unique solution to the problem. These chromosomes will later be decoded and evaluated using the fitness function.

There are a number of schemes which can be used to provide a genetic representation of solutions in a problem domain and these fall into two rough categories: abstract encoding or direct encoding.

Using **abstract binary encoding**, a string consisting of binary 1 or 0 bits is used to represent a solution. This is a flexible encoding as it can be used to represent whether a particular characteristic is present (i.e. 1 for true, 0 for false), to represent an integer number in binary notation (i.e. 1001 for 12) or a character in binary notation (i.e. 01000001 for A, in 8-bit ASCII). These methods can be combined as needed into one bit-string. To decode a binary string into the original values, the string is simply parsed according to the number of bits used to represent each value and converted to their original form. Using binary encoding also simplifies genetic operations as Crossover & Mutation operations only need to be designed to handle binary bit-strings. Whilst binary bit-strings as a method is not problem dependent, the length of the bit-string required to represent a solution for a particular problem is.

This is due to the different number of variables required to be encoded to represent a solution for different problems.

Chromosome A: 10000101011110
Chromosome B: 10000100010001

One form of direct encoding is **value encoding**. This is often used when binary encoding is restrictive such as representing non-integer values such as real numbers or other complicated data structures. However a major hurdle with value encoding is that encoding specific genetic operations need to be designed to allow crossover & mutation of solutions.

Chromosome A: B D E 9.920 8.553 22.250 F
Chromosome B: A B C 1.250 1.256 12.001 G

A second form of direct encoding - **permutation encoding** - is only applicable to ordering problems, such as the Travelling Salesman Problem (TSP). TSP is a mathematical problem studied in theoretical computer science where the objective is to find the shortest route which visits a number of cities exactly once. Therefore, for ordering problems, it is more practical to use the sequence itself than encoding to a different form such as binary. In the example below, two possible solutions to an 8 city TSP problem are presented, i.e. the order in which the salesman visits the cities.

Chromosome A: 1 5 6 8 3 4 2 7
Chromosome B: 5 2 3 7 4 1 6 8

For the Genetic Algorithm developed in this paper binary encoding was chosen by the author for a number of reasons: it is easier to implement from a programming point of view, more obvious to the layperson for demonstration purposes and problem independent - ensuring that the base Genetic Algorithm is also problem domain independent. However, to re-factor the Genetic Algorithm program presented in this thesis to use a different encoding would be relatively straight forward by moving out encoding dependent code to separate functions. These modularised encoding functions could then be called to encode/decode solutions dependent on the encoding being used.

4.1.1 Discrete Variables

For problems using discrete variables - for example steel sections which are only manufactured in discrete section sizes - one option would be to encode the values associated with a particularly section (such as cross-sectional area, second moment of area, radius of gyration etc). However, a more practical method would be to maintain a table of cross-sectional values and encode a pointer - a position in the table - in each chromosome. An example of encoding discrete variables can be found below.

A two element structure, with four possible cross-sections.

Section	A	I	r
1	4.53	11.6	1.6
2	11.2	172	3.93
3	25.7	856	5.78
4	33.6	1928	7.57

Assuming binary bit-string encoding.

⇒To encode 4 possibilities in binary, 2 bits are required (binary 00 to 11 = decimal 0 to 3 = 4 possibilities)

⇒To represent 2 sets of 4 possibilities, 2 sets of 2 bits are required = 4 bits

⇒Therefore, to obtain a section in the table for the first element, parse the first two bits as binary and add 1. e.g. binary 00 = decimal 0 + 1 = Section 1

Given two chromosomes:

Chromosome A: 0010
Chromosome B: 1101

Chromosome A: ⇒ 00 10
i.e. 00 = Section 1 & 10 = Section 3

Chromosome B: ⇒11 01
i.e. 11 = Section 4 & 01 = Section 2

⇒To obtain the cross-sectional values for each element, look-up table e.g.
To find the section properties for the first element of Chromosome B, look-up Section 4 in the table:

⇒A = 33.6, I = 1928, r = 7.57

4.1.2 Continuous Variables

To genetically encode continuous or real variables consideration must be given to discretisation. A Genetic Algorithm can only approximately represent continuous variables and a resolution (or tolerance) must be defined by the programmer. For example, if a genetic algorithm involved representing a continuous dimension such as a diameter, the permissible values range from 0 to infinity. However in practise this range can be reduced by considering minimum and maximum values. Hence, if it was decided by the designer that the permissible diameters lay in the range of 0 to 5m, then this would be more practical to encode. However, the designer must also define a resolution to be applied to the range. This allows a continuous variable to be represented in an equivalent discrete manner, which is easier to represent.

For example, if the minimum tolerance of the construction process was 10mm, then it would be reasonable to discretise the range into 500 steps (i.e. $5000mm \div 10mm$). If binary encoding was to be used, a minimum of 9 binary bits would be required to represent the range (9 binary bits = 512 values). Thus, the binary bit-string ranges from 000000000 to 111111111. To obtain the diameter associated with a bit-string of 0000011111, convert to the binary to decimal (31), therefore $31 \times 10mm = 310mm$. As nine binary bits provides for 512 values and only 500 are required, there are 12 superfluous values. The designer could either decide to force these to be invalid solutions or to resolve to the same value as that of the upper limit (i.e. 5000mm).

4.2 Fitness Functions

The role of the fitness function in a Genetic Algorithm is to provide a measure of the quality of a solution, defined with respect to the other members of the population. By evaluating the fitness of each member of the population, the fittest solutions in a particular generation can be selected to form the genetic basis of the succeeding generation. As such, the fitness function is the key driver of a Genetic Algorithm with far reaching implications for its speed, efficiency and effectiveness.

There are a number of requirements for a fitness function [2, 5]. It should:

- be fast in terms of processor execution time & optimised for performance.
- ideally be unique, i.e. that two different solutions cannot be assigned the same fitness value;
- and ideally be on a ratio scale - i.e. an individual with a fitness value of 4000 should be twice as fit as an individual with fitness value of 2000. However, in reality this is often unachievable but should at least roughly hold true.

The major reason for ensuring the fitness function is fast is that as a genetic algorithm works with a population of solutions, a slow fitness function will have a cumulative effect in slowing down the optimisation. For example, if the typical evaluation time of a fitness function for one solution is 30 seconds, it takes 30,000 seconds or over 8 hours to do 1,000 evaluations. Assuming a population size of 20, this would only be 50 generations worth of evaluations - which is not an unreasonable example. Additionally, as the fitness function has the greatest influence on the effectiveness of a Genetic Algorithm (being the key driver of the optimisation by influencing the selection process) an ineffective fitness function will lead to an ineffective optimisation of a problem. [3]

In order to optimise structural systems, some measure of the relative performance of a structure must be obtained. Whilst the aim might be the minimisation of weight, in order to avoid the optimisation process resulting in the minimum section size for every member, additional constraints must be implemented. In general, a structural analysis of the system must be undertaken to allow the fitness of an individual to be modified based on the performance of the structure - in terms of deflection, stress, buckling etc. Additional constraints, such as reducing the number of cross-section types which meet at any one joint for construction purposes, could also be enforced.

For the fitness functions developed in this paper, problem specific finite element solvers using the CALFEM toolbox were developed. As the main aim was to optimise weight whilst taking into account of the performance of the structure, in this paper an approach to fitness was adopted where the weight of the structure was modified by a number of factors. The modifying factors were based on tests made on CALFEM output, such as checking the displacement of the structure was within acceptable limits, on a sliding scale of 0 to 1 (0 being complete failure of the test, therefore giving an overall fitness of 0; 1 being a perfect pass of the test).

Whilst the fitness functions and solvers are discussed later in this work, the relevant source code can be found in Appendix E on page 75

Algorithm 2 Roulette Wheel Selection Pseudo-code

1. Sum fitness of all individuals in the population = **S**
2. Generate a random number **r** in the interval **0** to **S**
3. Repeat the following until termination
   (a) Iterate through the population, summing fitnesses in running total **s**.
   (b) If sum **s** is greater than **r**, return the individual number.

4.3 Selection

For each generation, the fitness of each member of the population is evaluated using a fitness function. However, in order to meaningfully use the fitness values to drive the optimisation, the population for the next generation must be selected from the current generation. A number of genetic operators for selection have been developed. In rough terms these fall into two categories: stochastic and deterministic.

Fitness proportional selection is one of the most common selection methods. It is a stochastic method which ensures that whilst fitter individuals have the greatest chance to be selected, there is a chance that weaker individuals will also be selected. The method is more commonly known as **Roulette Wheel Selection** (RWS) as this provides a useful analogy for how it functions. In general, the greater the fitness, the greater the proportion of the wheel (i.e. the roulette wheel pocket size) an individual is consequently given a greater chance of being selected. As each selection is independent of others selecting individuals is consequently analogous to a random throw on a roulette wheel.

Pseudo-code for Roulette Wheel Selection to select a single individual, from a population sorted in descending order of fitness, can be seen in Algorithm 2.

When using the RWS technique it is important to note the fitness values assigned to chromosomes should be positive numbers, considering the cumulative fitness of the population is used. A worked example for Roulette Wheel Selection is presented below, using a population of four individuals and their corresponding fitness (calculated using some fitness function) .

Individual	Fitness	% of Total
1	8000	40.0
2	6000	30.0
3	4000	20.0
4	2000	10.0
Total	20000	100.0

Summing the individual fitnesses obtains a total of 20,000. The relative percentages of the total for each individual are also shown above. The corresponding weighted roulette wheel visualisation is shown in below

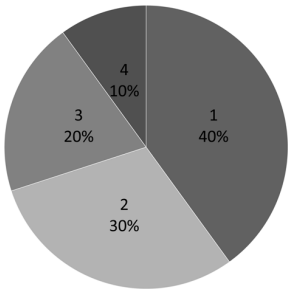

Fig. 2: Weighted Roulette Wheel

As individual 1 has a 40% share of the total fitness, with each "spin" of the roulette wheel it has a 40% chance of being selected. Similarly, although individual 4 only has a 10% share, it still has a chance of being selected for reproduction. In general, individuals with higher fitness with be selected to reproduce more often.

To select an individual, a random number between 0 and the total fitness is generated - in this case between 0 and 20,000. The RWS algorithm then iteratively sums the fitnesses of the population until the running total is greater than the random number.

Individual	Fitness	Cumulative Fitness
1	8000	8000
2	6000	14000
3	4000	18000
4	2000	20000

Therefore, if the random number generated was 16,000, the individual selected would be number 3 (as the running total at this point is 18,000). Similarly if the random number generated was 5,000, the individual selected would be number 1 (as the running total at this point is 8,000)

Deterministic selection methods, such as selecting a certain percentage of the fittest individuals, only allow the fittest members of a population survive and there is no element of randomness in their selection. Such methods generally cause population stagnation as weaker individuals will not be selected, causing the population to tend towards a limited set of genetic features i.e. homogeneity.

Whilst **elitism** is not strictly a selection method, it can be used to complement other genetic operators to ensure that the fittest members of a population survive to the next generation. With crossover and mutation, it is possible that the fittest member of the population could be lost. The use of elitism has a significant increase on the performance of a genetic algorithm and thus avoids regression of the population's fitness i.e. ensuring that fitness cannot decrease in successive generations.

For the algorithm developed in this paper, RWS was implemented as the selection method due to its effectiveness and ease of implementation. Elitism is also implemented and investigated in this work. Full MATLAB source code is available for Elitism and Roulette Wheel Selection in respectively Appendices D.4 on page 73 and D.5 on page 74.

4.4 Termination Conditions

With any optimisation process, the unknown is the optimum value of the function; it is not possible to simply terminate once "the optimum" has been found, since this would require prior knowledge of the optimum value. Therefore, the termination of the optimisation process must be based on other more subjective factors. The Genetic Algorithm created for this work implements two termination conditions: convergence and a generational limit.

- Convergence: If after a set number of generations no improvements are made to the fitness individual - i.e. the fittest solution does not improve - then the optimisation is terminated.
  - By default, if 20 generations pass without improvement in the fitness, the optimisation is terminated.
- Generations: After a set number of generations have elapsed, the optimisation is terminated.
  - This is set at 250 generations by default.

Other termination conditions not implemented in this work can also include computational time limits - most commonly in super-computer environments where access is charged by time.

4.5 Expanded GA Flowchart

In order to place the concepts of Encoding, Fitness Functions, Selection and Termination in context, the flowchart below was developed and expanded from Figure 1 on page 17. A complete flowchart can be found in Appendix A on page 59, annotated with the major investigative points of this work, as laid out in the Objectives section on page 15.

For structural optimisation with discrete variables, structural data look-up tables are required to decode an individual into a design solution, as seen in section 4.1.1 on page 19. For problems with continuous variables, structural data look-up tables are replaced by the conversion from a continuous value to a discrete value as detailed in section 4.1.2 on page 20.

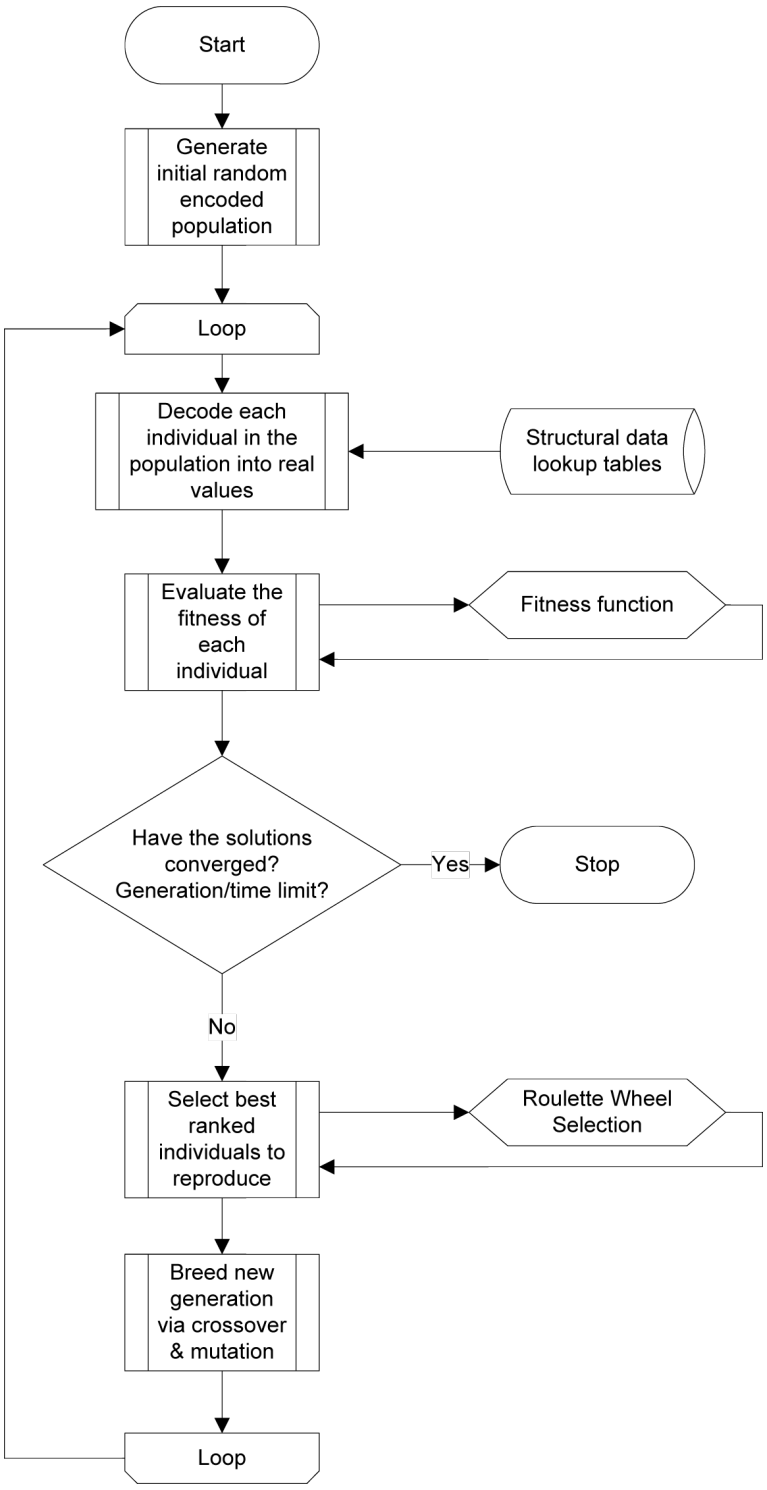

Fig. 3: Full Flowchart

5 Ten Bar Truss

A classic example of a structural optimisation problem with discrete variables is a truss system. Steel truss structures predominantly consist of sections bought from steel producers - for example Corus Group in the UK - and as such only standard sections are available to buy off-the-shelf. Accordingly if the aim is to optimise the weight of a truss, we must work with discrete design variables as defined by the steel sections available on the market.

Goldberg and Samtani - a postgraduate student of Goldberg - applied a genetic algorithm to the structural optimisation of a ten element plane truss. This particular example has been used as a benchmark in a number of research papers into structural optimisation using Genetic Algorithms and consequently is used as a benchmark problem for the Genetic Algorithm in this paper. The ten bar truss is presented below in Figure 4. However it has been adapted from the original Imperial units of 720 x 360 inches and point loads of 100kips into SI units of 18 x 9m and 450kN [10, 6, 11].

The objective of the Ten Bar Truss problem is to minimise the weight of the structure whilst subject to certain structural constraints on each member and on the performance of the structure as a whole. The main structural considerations applicable to a truss system are: buckling, yielding, and deflection. Therefore these three constraints are applied in the problem specific fitness function.

In Goldberg's original paper the areas of each of the ten elements could vary linearly between 0.1in^2 and 10in^2 (i.e. using continuous variables) . However, for the purposes of this paper the truss problem is approached using discrete variables. Each of the ten elements may be one of four distinct circular sections with associated sectional properties. Four circular hollow sections (CHS) were selected to approximately represent an even spread of sections available from the Corus group and in regular production (i.e. not rolled to order or special production). The sections were selected by the author from the Corus Celsius® 355 range, which are structural hollow sections (SHS) hot finished to EN10210: 2006. They have a minimum yield strength of 355MPa, a Young's Modulus of 210GPa and a density of 7850 kg/m^3.

The sectional properties of the chosen sections are presented below and this forms the basis of the structural look-up table used by the fitness function & solver for this problem.

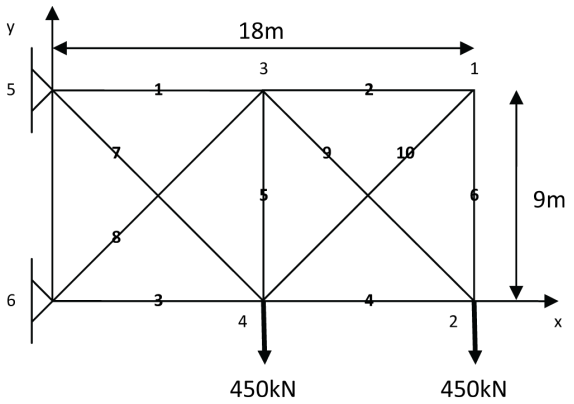

Fig. 4: Ten Bar Truss diagram

Cross Section Number	Outside Diameter D (mm)	Thickness T (mm)	Area A (cm^2)	Second moment of area I (cm^4)	Radius of gyration r (cm)
1	406.4	8.0	100	19874	14.1
2	355.6	8.0	87.4	13201	12.3
3	219.1	6.3	42.1	2386	7.53
4	114.3	3.2	11.2	172	3.93

Tab. 1: Ten Bar Truss Section Properties

To satisfy the **encodability** requirement, the binary bit-string method was chosen to encode the problem.. There are 10 elements in the structure and each element can be one of 4 sections. Therefore it follows that the bit-string length required to represent all possible combinations is 20 bits. The reasoning for length is presented below.

---

10 design variables (d.v.). Each design variable can be 1 of 4 values.

⇒Need to represent 4 choices in binary

⇒2 bits per d.v i.e. binary 00 to 11 = decimal 0 to 3, giving 4 distinct values

⇒10 d.v. x 2 bits = 20 bits.

To get cross section number for a particular element from the bit-string, select the appropriate pair of bits, convert from binary to decimal and add 1.

Example:

Chromosome A: 00000000000000000000

⇒decodes to represent all 10 sections with Cross Section 1 i.e. the 406.4 x 8.0 CHS

Chromosome B: 10101010101010101010

⇒decodes to represent all 10 sections with Cross Section 3 i.e. the 219.1 x 6.3 CHS

With an encoded bit-string length of 20 bits, the Genetic Algorithm will have to search among 1,048,576 (2^{20}) possible solutions, also known as the search space. The genetically encoded search space should match the number of possible solutions of the problem itself and with 4 possible values for 10 elements the number of combinations is 1,048,576 (4^{10}). Therefore, as expected, the binary encoding of the search space matches the number of combinations of the actual problem and this satisfies the **encodability** requirement.

5.1 Fitness Function

In order to produce practical and useful solutions to the problem, the fitness function for the Ten Bar Truss needs to penalise heavier structures & conversely reward lighter structures. However this must be done whilst imposing certain constraints from consideration of the buckling, yielding, and deflection of the structure. Accordingly the following scheme was devised:

$$Fitness = Weight\ fitness \times Displacement\ fitness \times Yielding\ fitness \times Buckling\ fitness$$

The displacement, yielding and buckling fitness values are factors on a range of 0 to 1 which modify the overall fitness of the structure depending on constraint violation (i.e. 0 for critical failure of a test and 1 for a pass within acceptable limits). Consequently if a single modifying factor is evaluated to be 0, the overall fitness of the structure will be found to be 0 and this implies a fatal flaw with a particular solution rendering it entirely unusable. The basis upon which the weight fitness and modifying factors were derived is presented below.

The MATLAB source code for the Ten Bar Truss fitness function is available in Appendix E.1.1 on page 76.

5.1.1 Weight Fitness

As stated, the objective of this problem is to minimise the weight of the truss system. However, a minor problem arises when trying to find a minimum with a Genetic Algorithm, in that they seek to maximise the fitness value. The solution adopted for the weight portion of the Ten Bar Truss fitness function was to create the weight fitness equation in the form of: $Weight\,Fitness = Constant - Weight$. The constant was determined by calculating the maximum weight of the structure (i.e. all elements using the heaviest section) and rounding the value upwards. With a maximum weight of the constant was found to be 9000. (Calculation details are given in Appendix B on page 60)

Consequently the weight fitness component was defined as:

$$Weight\,fitness = 9000 - \sum (Element\,weights\,in\,kg)$$

5.1.2 Displacement Fitness

To objectively evaluate the displacement fitness of the truss a scheme based upon imposing serviceable and ultimate displacement limits was created. As established above, the modifying factors are defined as a linear sliding scale from 0 to 1. Hence the following scheme was devised, on the assumption that if the overall displacement of the structure is less than the serviceable limit, that this in fitness terms is equal to a solution displacing exactly to the serviceable limit (and similarly with solutions exceeding the ultimate limit). The algorithm with which the displacement fitness is determined is shown below in Algorithm 3.

Algorithm 3 Displacement Fitness

if $maxDisplacement \leqslant slsLimit$
$\Rightarrow displacementFitness = 1.0$
else if $(ulsLimit < maxDisplacement > slsLimit)$
$\Rightarrow displacementFitness = \frac{(ulsLimit - maxDisplacement)}{slsLimit}$
else
$\Rightarrow displacementFitness = 0$
end-if

5.1.3 Stress Fitness

Whilst designing structural systems it is essential to ensure that under the expected loading there is an adequate factor of safety against failure of the structure. The stresses induced under the action of the loads to the structure must always be less than the strength of the material. However, it is also highly desirable to ensure that the structure does not permanently deform under the action of loading. Consequently, this means ensuring that the stresses acting within members must be less than the yield strength of the material. The yield stress is defined as the transition point between elastic & plastic behaviour. Before this point, the material is deforming elastically (i.e. will return to natural shape when loading is removed) and after it will begin to deform plastically (i.e. permanent changes of shape which remain after loading is removed).

In order to define a fitness scheme for yielding, the objective of the fitness function must be considered. Whilst it would be possible to penalise any solution which was found to have a number of elements yielding by assigning a fitness value of 0, this would be detrimental to the optimisation. This is because for a Genetic Algorithm effective at solving a problem there, in general, be a "hill to climb" i.e. there must be a way to converge to the solution - a simple right/wrong fitness measure does not allow for this. The corollary to this statement is that in the case of a severely flawed solution to a problem (as defined by constraint tests) it is appropriate to assign a fitness value of 0, as this will ensure the solution is not used in the formation of the next generation. [3]

Therefore, it would be preferable to penalise solutions on a sliding scale on the two main measures of yielding: number of elements yielded and ratio of stress to yield stress for each element i.e. the proximity to yielding of each element. It follows that a solution with a majority of elements yielded or close to yielding should be assigned a low fitness, whereas a solution with no elements yielding should be assigned a high fitness. However, even if a solution has no elements yielding, consideration must be made of how close to yielding the elements. This is to ensure that a solution where the stress in all the elements is at 99% of the yield stress is not assigned a high fitness. The resulting yielding fitness will always range between 0 and 1, however it will only reach 1 where no loading is applied to the structure and consequently when no element stresses are induced.

The algorithm used for Yielding Fitness can be seen below.

Algorithm 4 Ten Bar Truss Yielding Fitness

$numElementsYieldedPenalty = \left(1 - \frac{numElementsYielded}{10}\right)^2$

for each element ($i = 1 \rightarrow 10$**)**
$elementYieldingRatio = \frac{elementStress(i)}{yieldStress}$
if$(elementYieldingRatio > 1)$
$\Rightarrow elementYieldingRatio = 1$
end-if
$elementFitness(i) = (1 - elementYieldingRatio)^2$
loop

$yieldingFitness = \sqrt{\sum (elementFitness)/10} \times numElementsYieldedPenalty$

5.1.4 Element Force Fitness

As with yielding, when designing structural systems it is required to ensure that structural members do not fail under load. A structural member subjected to compressive stresses is at risk of buckling if compressive stresses exceed the ultimate compressive stress of the material.

A common buckling test is the Euler Buckling load, which calculates the maximum axial force a member can withstand without failing. It relates the modulus of elasticity E, the second moment of area I, the length of the column L and the effective length factor K. The effective length factor is based upon the end conditions of the element in question. However, as trusses are assumed to be pin jointed, both ends are pinned so $K = 1.0$.

$$F_{Euler} = \frac{\pi^2 EI}{(KL)^2}$$

The algorithm used for Buckling is similar to that of Yielding. However, instead of testing stresses against the yield stress, compressive element forces are compared to the Euler buckling load and penalised accordingly with the overall Buckling Fitness similarly ranging from 0 to 1.

5.2 CALFEM Solver

In order to provide the fitness function with the necessary structural analysis of a solution, a structural solver which wraps around & utilises CALFEM functions was developed.

The solver used for the Ten Bar Truss was extensively modified from an basic example (exs4) provided with the CALFEM toolbox. However, the solver was fully rewritten and significant alterations were made to improve performance, efficiency and functionality.

The main input to the solver is the 10×1 array of element cross-sectional areas A. This is then combined with the Young's Modulus of Steel to create the element properties matrix $e_i^p = [E\ A_i]$ (*Note: E is naturally constant for all elements*).

The solver returns:

- a global displacement vector $a\ (m)$
- a reaction vector $Q\ (kN)$
- an element element force matrix $N\ (kN)$
- an element stress matrix $O\ (kN/m^2)$
- a weight scalar $W\ (kg)$

The returned variables are consequently used in the fitness function described in the previous section.

The pseudo-code for the Ten Bar Truss solver is presented in Algorithm 5. (*Note: the naming convention of variables follows that of the CALFEM toolbox.*)

The full MATLAB source code for the Ten Bar Truss solver is available in Appendix E.1.2 on page 78.

Algorithm 5 Ten Bar Truss solver pseudo-code

1. Set finite element geometry of structure (topology matrix - mapping of elements to degrees of freedom - E^{dof} and nodal coordinates E^x, E^y).
2. Assemble load vector f.
3. Assemble element stiffness matrices K_i^e (where $i = 1 \rightarrow 10$) into global stiffness matrix K.
4. Set the boundary conditions and solve the system of equations for global displacement vector a & reaction vector Q.
5. Extract element displacement vector e^d from the global displacement vector according to the global topology matrix E^{dof}.
6. Compute element element forces matrix N, by relating E^x, E^y, the element properties e_i^p (where $i = 1 \rightarrow 10$) and the element displacement vector e_d.
7. Calculate element stresses $\sigma = \frac{N}{A}$, using element-by-element matrix algebra as both N and A are 10×1 matrices.
8. Calculate the weight of the structure W.

6 Reinforced Concrete Beam

The optimisation of a reinforced concrete simply supported beam was chosen by the author as a second problem to show the adaptability of Genetic Algorithms to different problems.

With a problem specific fitness function and finite element solver, the same generic Genetic Algorithm - as implemented by the author - was applied to the optimisation of this problem. No modifications were required to be made to the overall Genetic Algorithm to allow the beam problem to be optimised.

In common with the Ten Bar Truss problem, the objective is to find the minimum weight (i.e. the minimum beam cross-section required) subject to a number of constraints. The main considerations identified for an RC beam were deflection, reinforcement & moment capacity; these constraints are set out in the Fitness Function section.

Whilst Genetic Algorithms are ideally suited to solving problems with discrete variables (such as distinct steel cross-sections and their discontinuous cross-sectional properties), this does not exclude GAs from also solving problems with continuous variables (such as optimising dimensions). A dimension, such as a width or depth, can vary continuously between 0 and infinity. However, as discussed in the Continuous Variables section (4.1.2 on page 20), to encode such a variable requires consideration of discretisation & practical limits.

The initial problem was defined as a 9m simply supported RC beam, with loads of 1000kN at 3 & 6m and a cross-section varying between 0 to 1.5m by 0 to 1.5m. However, consideration was also given to flexibility of input i.e. allowing the user to specify a span, cross-section limits and a set of arbitrary loads. This will be covered later in section 7.1 on page 36.

As with the Ten Bar Truss, to satisfy the **encodability** requirement, the binary bit-string method was again chosen by the author. To encode a cross-sectional area uniquely, the height & width must be represented. If, say, the dimensions of the section are to be specified to 10mm accuracy, then a minimum of 150 possibilities are required. Therefore, for each dimension, requires 8 binary bits, (i.e. $2^7 = 128$ & $2^8 = 256 \Longrightarrow$8 bits to represent 150 values.) Consequently the bit-string length required is 16 bits. With a bit-string length of 16 bits, the search space is 65,536 (2^{16}) possible solutions.

However, as 8 binary bits provides 256 values, one option is to discount any values above 150. This would essentially cause approximately 40% of the search space to be negated and be specific to the problem at hand. Another option is to increase the resolution to that provided by 8 bits (i.e. $1500mm/256 \approx 5.8mm$) and subsequently round to the nearest 10mm.

The second option was selected as it is a more flexible method, given that if allowable cross-section size increases - e.g. a user inputs different cross-section size limits - the resolution scales proportionally.

A worked example - assuming limits of 0 to 1.5 by 0 to 1.5m - is provided below.

Example:

Each binary value represents $1500mm/256 = 5.859... = 5.86mm$

Chromosome A: 1111111100111110

⇒Breadth = 11111111, Height = 00111110
⇒Breadth = 256th value, Height = 63rd value
⇒$Breadth = 256 * 5.86 = 1500mm$, $Height = 63 * 5.86 = 370mm$ rounded to the nearest$10mm$

Chromosome B: 1100011001101110

⇒Breadth = 11000110, Height = 01101110
⇒Breadth = 199th value, Height = 111th value
⇒Breadth = $199 * 5.86 = 1170mm$, Height = $111 * 5.86 = 650mm$ rounded to the nearest$10mm$

6.1 Fitness Function

The fitness function used for the Simply Supported Beam problem is similar to that applied in the Ten Bar Truss. The weight fitness is the primary component and is modified by a number of other factors on a range from 0 to 1. *Note: the word "fitness" in the formula presented below has been shortened to "fit.", for typographical reasons.*

$$Fitness = Weight\ fit. \times Displacement\ fit. \times Plastic\ Moment\ fit. \times Moment\ Capacity\ fit. \times Rebar\ fit.$$

The weight fitness is obtained in a similar manner to that of the Ten Bar Truss. However it is based upon weight per unit length rather than total weight, as only one cross-section is used for the structure. The constant was again calculated from the maximum weight of the structure.

The displacement, moment capacity and plastic moment tests were applied in a similar style to the modifying factors in the Ten Bar Truss problem. The displacement fitness test is identical to that in the truss problem, although the limits applied are from BS8110, limiting deflection to $^{span}/_{250}$ and $^{span}/_{500}$ (Cl 3.4.6.3) [12]. The moment capacity fitness compares the maximum moment capacity of the reinforcement to the moment applied; it assigns a linear fitness value between 0 if the full capacity is used & 1 if no capacity is used. The plastic moment fitness assigns a value between 0 & 1 based on the moment applied to the section and the plastic moment $M_p = (b{*}h^2) \div 4 \times yieldStress$. In the interest of brevity, further details of the modifying factors have been omitted due to the similarity in implementation to those used in the Ten Bar Truss. However, full MATLAB code listings for the Simply Supported Beam fitness function are available in Appendix E.2.1 on page 83.

The major difference between the two problems presented in this paper - in terms of the fitness functions - is the requirement to provide reinforcement within the concrete beam section. The procedure used to design the reinforcement, for bending in the beam at the ultimate limit state, is taken from BS8110-1:1997 - specifically Cl 3.4.4.4: Design formulae for rectangular beams [12]. The procedure can be found in Algorithm 6.

The area of steel required, calculated using the Design formulae, is then used to select a reinforcement solution. The selection process roughly models that of the manual process.

The procedure is designed to find a solution which satisfies the area of re-bar required and the section width. It should be noted that the re-bar selection method will tend towards a solution with the smallest possible diameter bars. In real-world design, assuming adequate section width, engineering judgement would be called upon to decide between supplying 9T16, 6T20 or 4T25 bars (all of which provide approximately similar total areas of re-bar). However, automating such judgement is outwith the scope of this work and the method used provides an adequate, if not always entirely realistic re-bar selection. The pseudo-code for re-bar selection can be seen in Algorithm 7.

In order for the reinforcement selection process to influence the Genetic Algorithm, it is necessary to assign a fitness value based upon its results. However, it is overly problematic to design a meaningful scheme to evaluate the fitness of a re-bar selection, given that there could be many equally valid solutions. Additionally, as mentioned previously, a right/wrong fitness measure does not allow a genetic algorithm to converge effectively to a solution. Therefore, the reinforcement fitness is a qualitative measure of the reinforcement solution. It is assigned on an indirect “reward” basis rather than any direct calculation.

Initially the fitness is assumed to be 0 and as a number of checks are met, the value is incremented up to a maximum value of 1.0. The checks are:

- whether the reinforcement fits into the section with the minimum allowable centre to centre spacing;
- whether the design satisfies cover to reinforcement requirements;
- and whether a solution, which fits into the section, was found using the bar diameters available to the selection process.

Complete MATLAB source code for the beam reinforcement design process is available in Appendix E.2.4 on page 88.

Algorithm 6 Beam Reinforcement calculations pseudo-code

1. Calculate the maximum moment capacity of the section (corresponding to a neutral axis depth at ductile failure of 0.5d and only tensile reinforcement)

$$M_r = 0.156 \times bd^2 \times f_{cu}$$

2. Obtain coefficient K, which should not exceed 0.156.

$$K = \frac{M_a}{bd^2 f_{cu}}$$

3. Obtain lever arm z & neutral axis depth x and check they are within acceptable limits.

$$z = (0.5 + \sqrt{0.25 - \left(\frac{K}{0.9}\right)}d \quad x = \frac{(d-z)}{0.45}$$

4. Calculate required area of steel for the applied moment, enforcing a minimum area of steel required.

$$A_s = \frac{M_a}{0.95 \times f_y \times z} \quad A_{min} = \frac{0.13}{bh}$$

Algorithm 7 Re-bar Selection pseudo-code

1. Select minimum bar diameter available from range supplied (say 16, 20, 25, 32, 40 diameter bars)
2. Repeat the following re-bar selection algorithm until termination
   (a) Calculate number of bars, using currently selected diameter, required to provide required area of steel.
   (b) Required width is the total width of bars, shear links and minimum bar spacings.
   (c) **if** $\mathrm{Required\ width} \leq$ Section width
   Solution found, terminate.
   else
   Select next bar diameter up
   (or terminate if at maximum bar diameter allowable)
   end-if

6.2 CALFEM Solver

The CALFEM solver used to provide the fitness function with a finite element analysis of a solution was adapted from a basic example in the CALFEM toolbox (example exs3). As with the Ten Bar Truss solver, a number of extensive changes were made to the original example.

The two main inputs to the solver are the breadth b and height h.

The solver returns:

- a global displacement vector $a\ (m)$
- a reaction vector $Q\ (kN)$
- a section forces array e_s consisting of the element force N, shear force V and moment M for each finite element
- a weight per unit length scalar $W\ (kg/m)$

The major improvements made by the author to the solver, over the CALFEM example, is to allow Flexibility of Input. Loads can be specified at arbitrary points on the beam (i.e. not necessarily on finite element nodal points) and an arbitrary span can be specified . Flexibility of Input is discussed in section 7.1 on the next page.

The full MATLAB source code for the Simply Supported Beam solver is available in Appendix E.2.2 on page 85.

7 Graphical User Interface

Whilst the Genetic Algorithm developed in this work is fully functional from the MATLAB command line, a graphical user interface has also been developed as a user-friendly front-end. The GUI was built using M-code and the MATLAB Graphical User Interface Development Environment (GUIDE).

The primary reason for developing a GUI was to allow a user unfamiliar with the topic to experiment with genetic algorithms and their parameters, such as population size & mutation. A screen-shot of the GUI can be found in Figure 5.

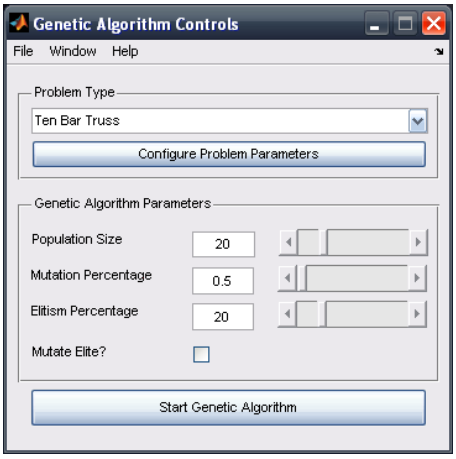

Fig. 5: Genetic Algorithm GUI

An additional advantage to providing a GUI is that the optimisation process can be exposed to the user, through visualisations rather than simply text based output.

For each optimisation run, the user is provided with three windows detailing the operation of the optimisation process:

1. a finite element model of the problem, showing the fittest solution and its deflected shape, updated each generation;
2. a fitness graph showing the fitness of the best individual, as well as the average fitness of the population, across generations;
3. and finally a visualisation of the best solution to the problem found in the optimisation run (i.e. showing section sizes and other details).

See appendix C on page 61 for examples of typical visualisations.

7.1 Flexibility of Input

To allow user interaction with the Ten Bar Truss and Simply Supported Beam problems, the ability to configure the parameters of the problem (within limits) was provided. This was developed as a proof-of-concept, to demonstrate the ability of Genetic Algorithms to adapt to different problem scenarios other than the default values provided within this work.

With the truss problem, the two point loads can be altered in magnitude and the different results observed. The truss problem options window can be found in Figure 6. By way of example, the results of two optimisation runs with pairs of 1000kN and 100kN loads are presented below in Figures 7 and 8 respectively.

A graphical user interface was also developed for the Simply Supported Beam problem. Almost all parameters of the problem are configurable, from the length of span to arbitrarily placed loads and the limits placed upon section size. A screen-shot of the beam problem options interface can be seen in Figure 9.

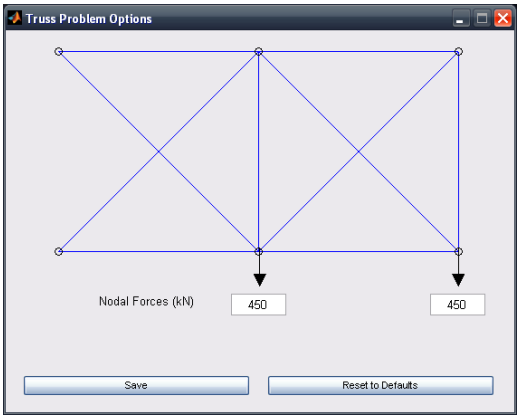

Fig. 6: Truss Parameter GUI

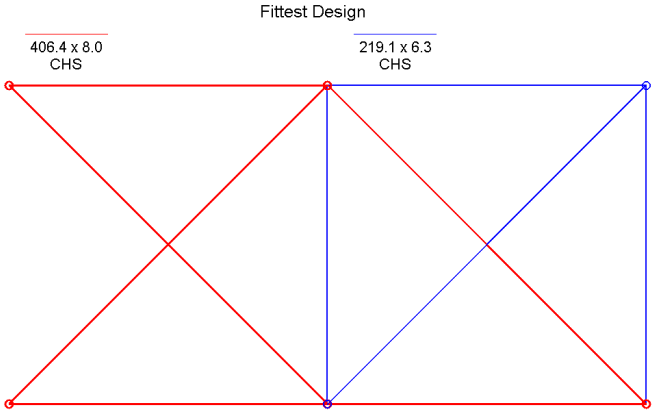

Fig. 7: Truss with loads of 1000kN, 1000kN

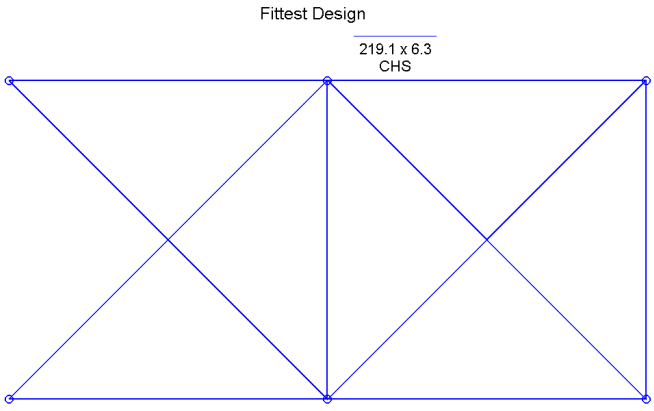

Fig. 8: Truss with loads of 100kN, 100kN

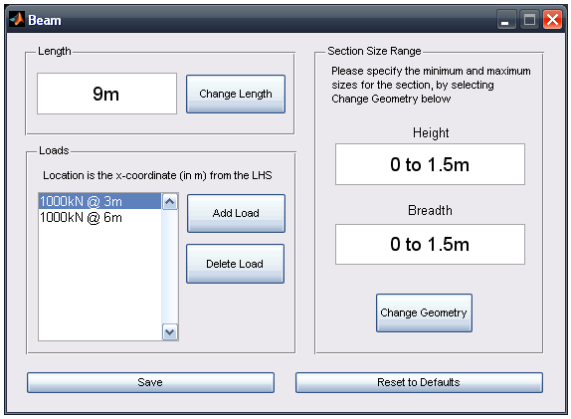

Fig. 9: Beam Parameter GUI

Part III. Results

8 Effect of Genetic Algorithm Parameters

With any computer program, whether in Engineering or not, care must be taken with its use. The adage of "Garbage in, Garbage out" always applies, whether with the use of spreadsheets or Finite Element Modelling. Similar to FEM, if a Genetic Algorithm is used without some understanding of its workings or the significance & effect of its parameters, then its output cannot be relied upon.

With this in mind, an investigation was undertaken by the author into the effect of the various Genetic Algorithm parameters on the optimisation of the Ten Bar Truss problem. Testing was undertaken by establishing baseline values for the four parameters: population size, mutation %, elitism % and whether elites should be mutated.

The baseline values were decided upon by the author based on his experience in developing the algorithm, as well as a brief re-cap on these parameters is presented below:

- Population size of 20 (controls the number of individuals with which the GA has to work with);
- Mutation of 0.5% (controls the probability that any one particular bit may be inverted);
- Elitism of 10% (controls the percentage of the fittest members of the current population which are preserved to the next generation without crossover/reproduction);
- Mutate elites disabled (controls whether members of the population selected as elites can be mutated or not).

For each permutation of an individual variable, the Genetic Algorithm was run 20 times and the results noted. It should be noted that as Genetic Algorithms are a stochastic process, using probabilistic methods to converge to a solution, any two result sets can differ. Consequently, there is an element of uncertainty in the results.

Nevertheless, rough trends should hold true, giving approximately similar results if an identical test was performed. For example, if the GA is run 20 times with a particular set of parameters, if the optimum solution is reached 5 times, it is reasonable to expect that if the GA with was run a further 20 times, that the optimum solution would be reached an approximately similar number of times.

Termination conditions chosen by the author were:

- terminate if 20 generations have passed since the last improvement in maximum fitness value
- terminate if 250 generations have passed

In all cases, the first set of results were taken. As Genetic Algorithms are a stochastic process, subsequent result sets could present an entirely different narrative. However, on average, result trends should hold roughly true.

Population size	Num. times optimum solution reached (of 20 runs)	Average number of generations to reach maximum fitness found	Average optimisation time (s)
10	1	33.00	3.87
20	6	29.25	4.63
50	13	22.40	6.86
100	19	17.85	10.73

Tab. 2: Effect of Population Size

8.1 Population Size

The theory of Genetic Algorithms suggests that the majority of their power is derived from the ability for crossover & selection. Crossover & selection effectively work in tandem to develop highly fit individuals, by innovating new individuals through combination of genetic material from across the population. As GAs are implicitly parallel - i.e. working from a large number of points simultaneously - if the population size is increased then it should follow that the efficacy of the optimisation increases [5].

By only modifying the population size and keeping other parameters constant at the chosen baseline values, the effect of population size on optimisation using Genetic Algorithms is readily apparent. In Table 2 below, the salient results are presented. *Note: The average number of generations to reach maximum fitness refers to the fittest individual found in that run, rather than the optimum solution.*

Whilst a low population size reduces the optimisation time, with less genetic material to work with, the algorithm finds the optimum solution rarely. A false optimum was found 19 times out of 20, with a population size of 10 individuals.

Increasing the population size, increases the average optimisation time. This is simply due to the additional computation required in processing larger population sizes. Some of the increase in optimisation time can be attributed to the need to perform genetic operations, such as crossover, on a larger population, which will necessarily take longer. However, the main factor in this increase will be the fitness function. For each generation, the fitness of each member of the population is evaluated by the fitness function. Therefore, if the number of individuals is increased, the fitness function is evaluated a proportionally larger number of times per generation.

Although the optimisation will take longer with a larger population size, running the GA with a population size of 100 resulted in the optimum solution 19 times out of 20. As mentioned previously, it is perfectly possible that a subsequent set of runs for a population of 100 could result in the optimum solution in all runs.

8.2 Mutation

As stated in the introduction to Genetic Algorithms, mutation plays a secondary but vital role. It serves to maintain diversity, by potentially creating new novel genetic features. Likewise, mutation can prevent the lost of potentially useful features during the normal process of crossover and selection [5].

Mutation (%)	Num. bits mutated per generation (out of 400)	Num. times optimum solution reached (of 20 runs)	Average number of generations to reach maximum fitness found	Average optimisation time (s)
0	0	1	11.30	3.01
0.5	2	4	27.00	4.47
2	8	10	34.40	5.14
5	20	12	38.40	5.56
10	40	7	33.45	5.10
20	80	0	30.20	4.75

Tab. 3: Effect of Mutation

With a population size of 20, a number of different rates of mutation were used. The pertinent results are shown in Table 5. *Note: As the Ten Bar Truss problem has a bit-string length of 20, the total number of bits per generation of* $20 \times 20 = 400$.

From the results it can be seen that without mutation, the optimisation performs poorly, only reaching the optimum result once.

The 0.5% mutation rate results in the optimum solution being reached 4 times and this roughly compares to a set of runs with identical settings in the population size results (With population size 20, and also mutation of 0.5%, the optimum was obtained 6 times). This shows that two sets of 20 optimisation runs with identical settings will result in similar but not exactly the same result.

As the mutation rates are increased, the number of times the optimum solution is reached increases. At a rate of 5%, mutation causes the optimum to be found 12 out of 20 times. However, increasing mutation rates further has a detrimental effect on the optimisation.

Therefore it can be inferred that minimal rates of mutation are beneficial to the optimisation process. Excessive rates cause overzealous mutation and the optimum solution is not found as often, due the loss of potentially useful genetic material.

As the mutation rates are increased, the average number of generations & length of optimisation follows a similar pattern to the number of times the optimum solution is reached. This correlation should be circumstantial and an indirect result of a optimisation process terminating early due to the lack of improvement made over a (false) optimum found early in the optimisation process.

8.3 Elitism

In the normal course of crossover, it is entirely possible that the fittest members of the population could be lost. For example, if by chance the fittest individual is crossed over with the least fit individual, the optimisation is setback. Consequently, there is no guarantee that the optimum - or any other highly fit individual - will survive in the population.

Elitism is used to complement the other genetic operations by ensuring that the optimisation is not setback by any loss of the fittest members of the population. In theory, it should improve the performance of the optimisation significantly.

Elitism (%)	Num. of Elites (in a population of 20)	Num. times optimum solution reached (of 20 runs)	Average number of generations to reach maximum fitness found	Average optimisation time (s)
0	0	0	193.55	18.77
10	2	7	35.00	5.16
20	4	2	23.00	4.04
40	8	3	28.25	4..45

Tab. 4: Effect of Elitism

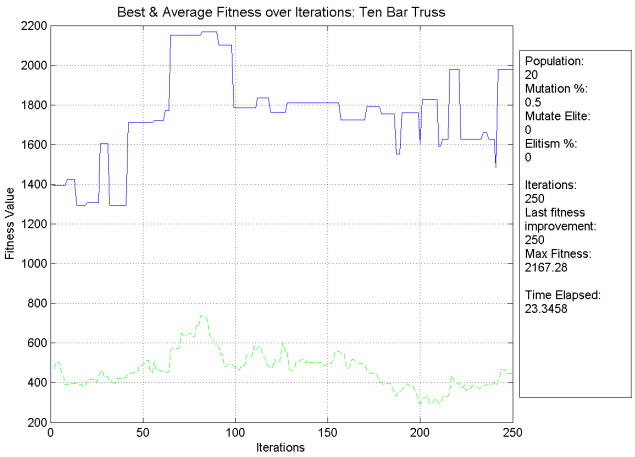

Fig. 10: Fitness graph over iterations (without Elitism)

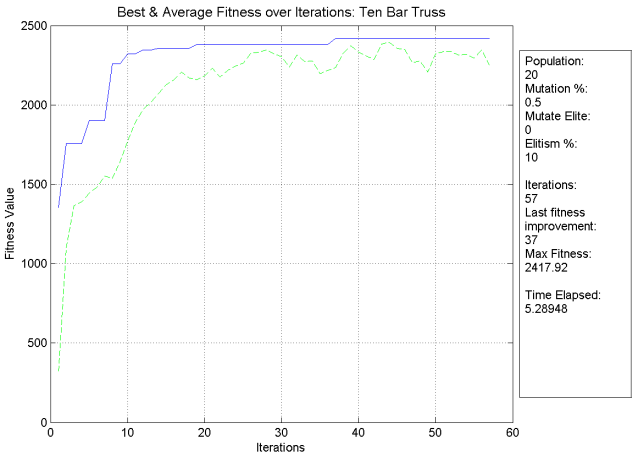

Fig. 11: Fitness graph over iterations (with Elitism)

As can be seen in the results in Table 4, using no elitism has a severely detrimental effect on the optimisation. In many cases the optimisation was terminated after reaching the maximum number of generations (250) and in no run was the optimum solution obtained. From Figure 11 it can be seen that the fittest individuals found in this run are lost numerous times, causing the graph to trend downwards. The average fitness of the population is erratic at best and no overall improvement is made.

Using elitism has a significant increase on the performance of a genetic algorithm, by avoiding regression of the population's fitness i.e. ensuring that fitness cannot decrease in successive generations. In Figure 11, elitism of 10% was used i.e. the fittest two individuals from each generation are preserved unaltered to the next generation.

8.3.1 Mutation of Elites

In the author's implementation of a Genetic Algorithm, mutation may be applied to all members of the population or to all but the fittest i.e. not those selected as elites. Theoretically by not allowing mutation of elites, which is the default behaviour of the GA, the fittest individuals are preserved entirely unchanged from one generation to the next. This can have two consequences:

- the fittest solution is preserved from detrimental mutation, ensuring no loss of fitness, ensuring a smooth optimisation process;
- or the fittest members may be denied beneficial mutation, which could potentially result in a fitter - or even the optimum - solution.

In Table 5 it can be seen that the two result sets are approximately similar. Whilst not mutating elites may occasionally prevent loss of fitness, it may also occasionally impede improvement. Given the stochastic nature of GAs, the author finds it difficult to comment on the relative effects of elite mutation, as in any particular optimisation run mutation of elites may have a positive, negative or no effect at all.

However, with a sufficient population size, from observation there is likely to be many copies of the fitter individuals, so not mutating elites may have little, if any, effect. Nevertheless, if a low population was used - for if example the fitness function was computationally expensive - not mutating elites may protect the fittest members of the population for potentially detrimental mutation.

Mutate Elites?	Num. times optimum solution reached (of 20 runs)	Average number of generations to reach maximum fitness found	Average optimisation time (s)
True	4	30.40	4.68
False	5	28.00	4.49

Tab. 5: Effect of Mutation of Elites

9 Ten Bar Truss Results

9.1 Fitness Function Testing

To demonstrate and test the Ten Bar Truss fitness function, the problem was optimised both with and without constraints in the following manner: optimising weight only; optimising for weight and one constraint; and finally optimising for weight and all constraints.

Based on the experience of the studies in the previous section, a standard set of genetic algorithm parameters was used in all cases: a population of 100, mutation rate of 0.5%, elitism of 5% and no mutation of elites. The loading applied was two $450kN$ point loads, as seen in Figure 4 on page 26. As with the genetic algorithm parameter testing, the optimisation process was run 20 times in order to clearly establish the optimum solution for the given constraints.

Optimising for weight only: The unconstrained optimisation of the truss - i.e. the minimisation of weight with no regard for structural requirements - consistently resulted in a design where the minimum cross-section - the $114.3 \times 3.2\, CHS$ - was used for all elements. The optimum solution was found in 20 optimisation runs out of 20. The maximum vertical deflection was found to be $41mm$ and the weight of the structure was found to be $912kg$. The solution can be found in Figure 12.

Optimising for weight with displacement constraint: Optimising for the minimum weight whilst constraining the displacement resulted in the optimum solution in 16 runs out of 20. The maximum vertical deflection of the structure was found to be $5mm$ and the weight was $4138kg$. The design chosen by the GA provides larger members along expected load paths and also bracing to limit deflection. The resulting structure can be seen in Figure 13.

Optimising for weight with stress constraint: The two point loads of 450kN do not produce sufficient stresses to cause any member of the structure to approach yielding. Therefore, optimising for weight with a constraint on stresses produces a similar solution to optimising for weight only.

Optimising for weight with element force constraint: The optimum solution for weight with the element force constraint was found to have a total weight of $5465kg$ and the maximum deflection was found to be $6mm$. The design provides a fully braced base using $355.6 \times 8.0\, CHS$ sections near the supports with lighter non-load-carrying members as well as a $355.6 \times 8.0\, CHS$ section on the expected load path from the lower right hand node. The resulting design can be found in Figure 14.

Optimising for weight and all constraints: The optimum design - for the default load case - was found to the same as that found by optimising for weight with the element force constraint. With two $450kN$ point loads, the buckling fitness - the ratio of the each elements axial force to its Euler buckling load - is the primary influence on the optimum solution. The optimum solution was found - on average - in 10.8s.

Notably, the second-most optimum solution, was found to have as weight of $5785kg$ and a maximum deflection of $5mm$. The additional $355.6 \times 8.0\, CHS$ in this solution - which adds $320kg$ to the structure - brings relatively little benefit to the structure as it is unlikely to be carrying significant load and will only stiffen the structure marginally. Although this design results in marginally a lower deflection, the added weight causes this solution to be judged less fit than the optimum solution. The design can be seen in Figure 15.

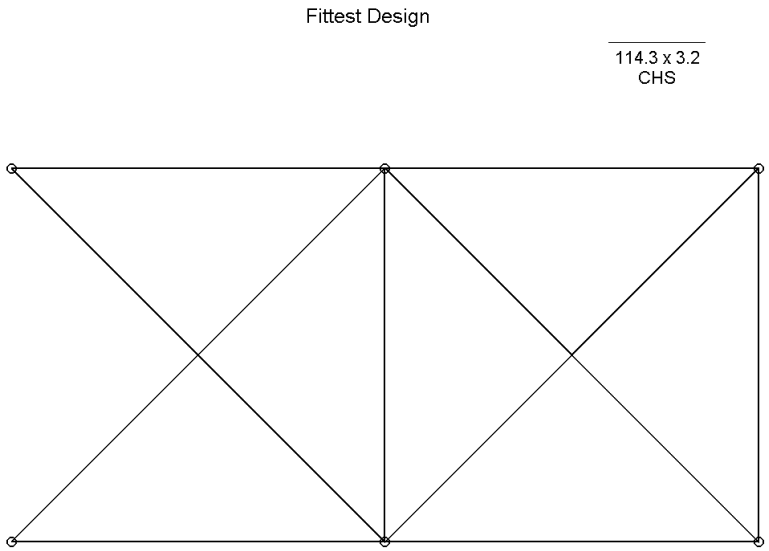

Fig. 12: Optimum truss design for weight only

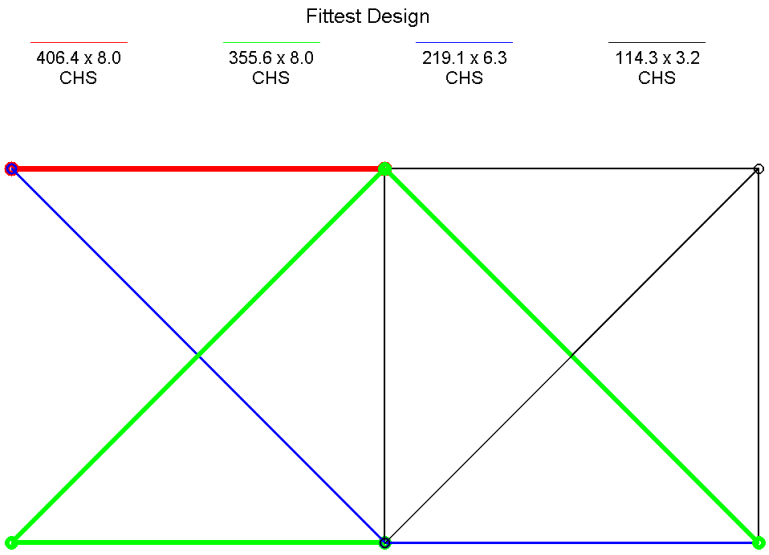

Fig. 13: Optimum truss design for weight & displacement

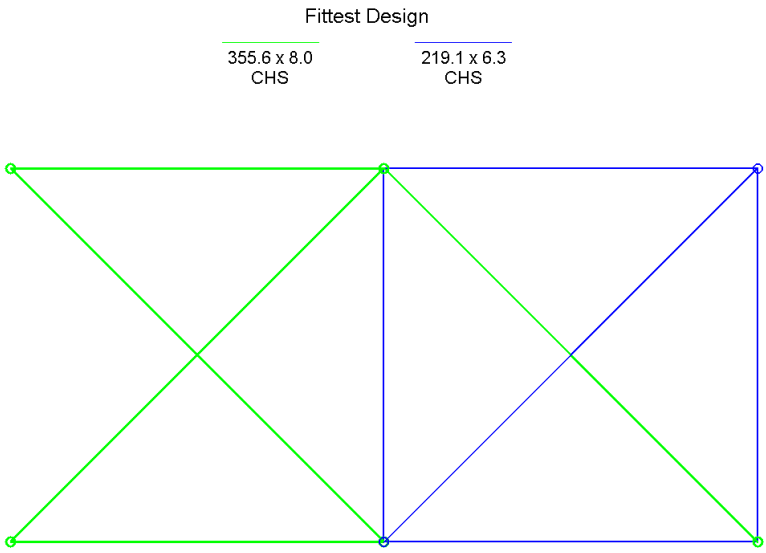

Fig. 14: Optimum truss design for weight & buckling

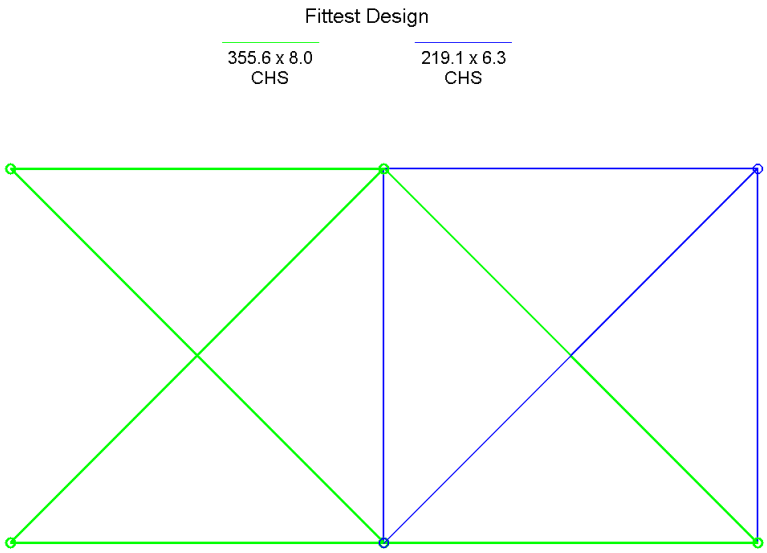

Fig. 15: Second-most optimum truss design for complete fitness function

9.2 Search Space

As has been seen, the Genetic Algorithm is able to converge to the optimum solution - for a majority of runs - in approximately 10 seconds (with a large population, using elitism and applying mutation sparingly). However the efficiency of a Genetic Algorithm in traversing the search space has not - so far - been demonstrated.

A worked calculation has been provided below, using results obtained during Population Size testing (section 8.1 on page 40:

For a population of 100 individuals, the optimum result was returned for 95% of optimisation runs (19 out of 20 times).

An average of 17.85 generations was required to reach convergence, with a minimum of 10 and maximum of 30 generations required. The fitness function was therefore evaluated on average $100 \times 17.85 = 1785$ times, (consequently a minimum of 1000 and maximum 3000 fitness evaluations).

The size of the search space is $2^{20} + 1 = 1048576 + 1 = 1048577$ possible solutions.

Therefore, the GA can be said conservatively to have evaluated on average only $1785 \div 1048577 = 0.17\%$ of the possible solutions (with minimum of 0.095% and maximum of 0.29%).

The calculation above is however based upon the assumption that each evaluation is of a unique individual. It is exceptionally unlikely that the same individual has not been evaluated more than once - particularly the higher fitness individuals. Consequently, the GA has evaluated on average less than 0.17% of the search space, whilst locating the optimum solution (and in all likelihood, significantly less than that figure).

The majority of the computational demand from any GA, is its fitness function. The more times the fitness function is evaluated, the longer the optimisation process. To illustrate the efficiency of genetic algorithms in traversing the search space, in terms of computational time, the fitness function was evaluated for every possible solution. The brute-force method evaluated all 1,048,577 solutions in 4094 seconds, approximately 68 minutes. This compares to a typical optimisation run using the Genetic Algorithm, of between 10 and 20 seconds with a large population.

The results of the brute-force method are displayed below as a fitness landscape (or search space) plot with fitness value on the y-axis and individual number on the x-axis.

Note: due to scaling issues with displaying 2^{20} data-points and plotted lines, the graph is misleading in that many more invalid or 0 solutions exist than can be seen below.

The fitness landscape is exceptionally multi-modal, with many local optima. This demonstrates the optimisation challenge the Genetic Algorithm must overcome to locate the optimum solution. If Figure 16 is re-ordered in ascending order by fitness value, a more coherent shape appears. *Note: the x-axis in figure 17 below is meaningless, given that the data-points have been re-ordered.*

From the Figures 16 and *17*, the task of locating the optimum solution for the Ten Bar Truss with a search space of 1,048,577 possible solutions - and the efficiency of a Genetic Algorithm in achieving this - becomes apparent.

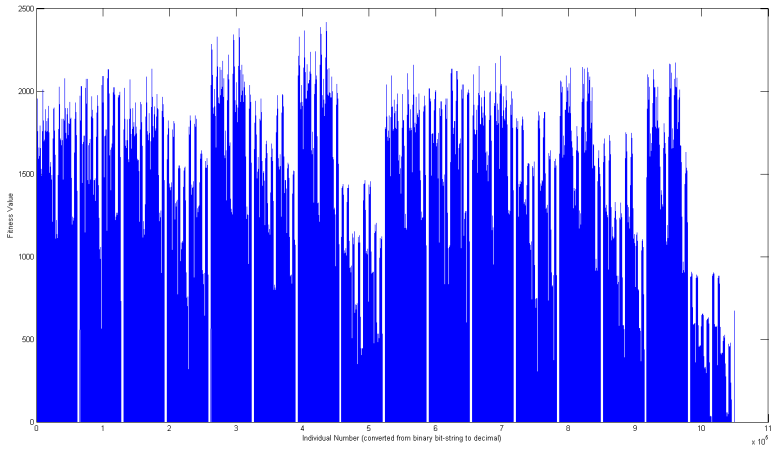

Fig. 16: Truss search space map (sorted by individual number)

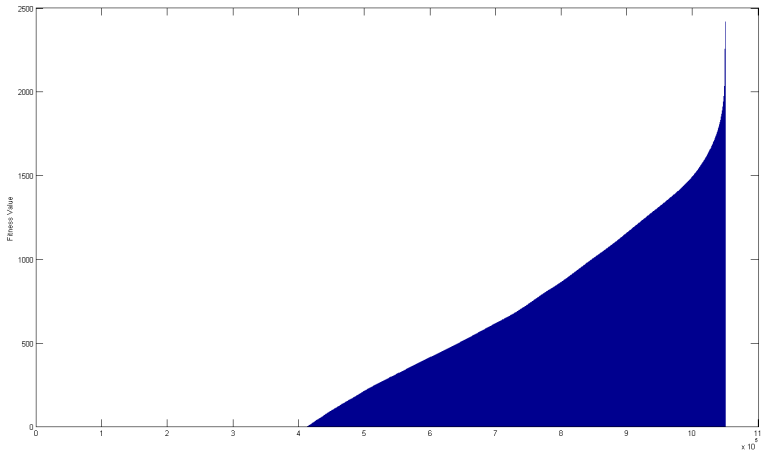

Fig. 17: Truss search space map (sorted by fitness value)

10 Simply Supported Beam Results

10.1 Fitness Function Testing

The Simply Supported Beam Problem was optimised for a number of scenarios, similar to the testing undertaken with the truss problem. The loading applied to the $9m$ beam were point loads of $1000kN$ at the $1/3^{rd}$ points and the optimisation process was again run 20 times in order to establish the optimum solution for the given scenario.

Optimising for weight only

With no consideration to other constraints and optimising for the minimum weight, resulted in a section size of $0.2 \times 1.5m$ - the maximum depth allowed. This section size was selected by the GA to maximise the second moment of area of the beam, whilst minimising weight. For a rectangular section the second moment of area is $I = \frac{bd^3}{12}$. By maximising the depth, the greater the second moment of area and the greater the bending resistance of the beam. The maximum deflection over the 9m span was found to be $3mm$ with this section size.

Optimising for weight with displacement constraint This resulted in the same section size as optimising for weight only as the standard loads do not cause sufficient deflection for the displacement constraint to apply i.e. well within the BS8110 limits of $span/500 = 18mm$ and $span/250 = 36mm$.

Optimising for weight and moment constraint The moment constraint does not apply for the default problem as the two point loads of $1000kN$ do not cause sufficient moments to reach the plastic moment of the beam.

Optimising for weight and all constraints & reinforcement The optimum solution to the Simply Supported Beam problem was found to be a section size of $0.4 \times 1.24m$ with $6T32$ re-bar. The maximum deflection over the span of $9m$ was found to be $3mm$. The optimum solution was found in 4 to 8 seconds on average.

The optimum section was selected by the GA because it has sufficient width to provide reinforcement and is sufficiently deep to provide adequate bending resistance from concrete alone. This result is inefficient for one major reason, no account is taken of the effect of the reinforcement upon the stiffness of the structure. This is due to a lack of time to fully tackle the effect of reinforcement upon the bending stiffness. Therefore, the beam design is unnecessarily deep.

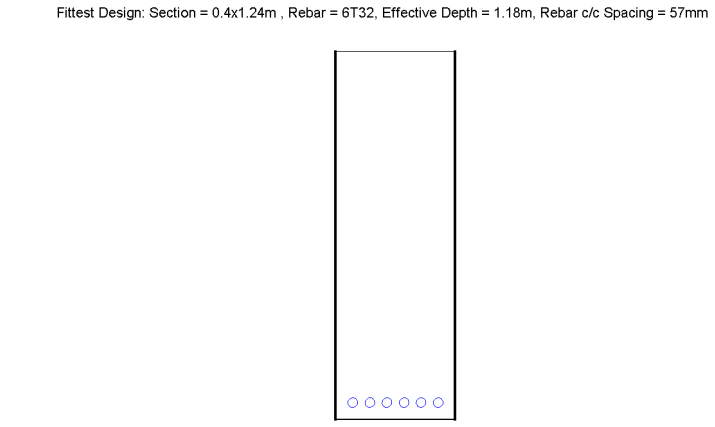

Fig. 18: Optimum beam design for complete fitness function

10.2 Search Space

Although the Simply Supported beam problem has a smaller search space than that of the truss problem, the fitness function is still exceptionally multi-modal. As with the Ten Bar Truss, for comparison purposes, the fitness function was evaluated for every possible solution ($2^{16} + 1 = 65536$ solutions) using brute force. Whilst the Genetic Algorithm, using a large population size returns a solution in under 10 seconds, the brute force method took approximately 160s. This confirms results found with the Ten Bar Truss problem, in that the GA conducts a highly efficient search of the search space in locating the optimum solution.

Similar to the two figure produced for the truss problem, two fitness maps were created to illustrate the multi-modality of the fitness landscape for the Simply Supported Beam problem.

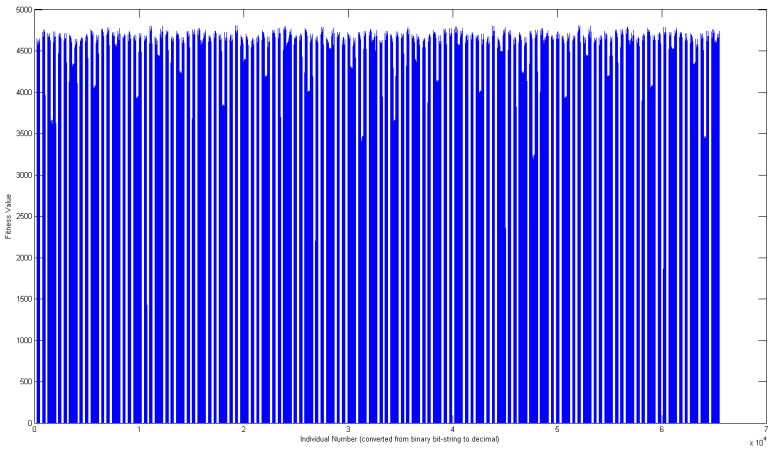

Fig. 19: Beam search space map (sorted by individual number)

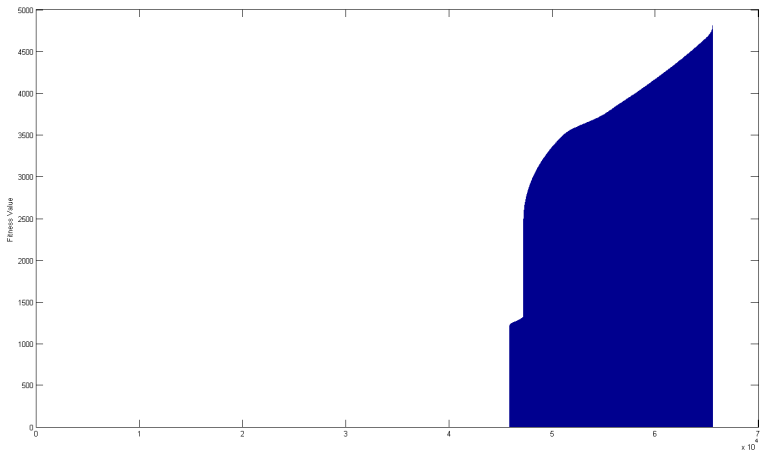

Fig. 20: Beam search space map (sorted by fitness value)

Part IV. Discussion & Conclusions

11 Discussion

In order to apply optimisation to structural systems, the method selected must be easily applied to different design problems, use a minimum of auxiliary information, attempt to find the global optimum and be able to solve problems with discrete variables. Genetic Algorithms were selected as the optimisation method for this work because they not only satisfy these criteria, they are a robust & powerful global search technique which is relatively easy to implement.

GAs use relatively simple principles inspired from evolution, namely survival of the fittest and genetic reproduction. By excluding problem specific information from the optimisation process, a Genetic Algorithm is generic and can be applied to solve different problems. To apply a GA to a problem, all that is required is the creation of a fitness function, which models the problem and objectively assigns a fitness value to a solution. GAs exploiting nothing than the similarities between highly fit individuals and using probabilistic experimentation to evolve fitter solutions; and by working from a diverse populations of designs simultaneously, the implicit parallelism of a genetic algorithm ensures that local optima are ignored and the global optimum is obtained. [3, 2]

The Genetic Algorithm used in this work was developed from scratch, with no reference to existing GA implementations and using only high-level conceptual pseudocode. The performance, efficiency and apparent intelligence of the GA has at times surprised the author; the complicated, powerful and oftentimes surprising behaviour resulting from the author's handiwork is something akin to watching a child grow. To quote Davis, in his "Handbook of Genetic Algorithms" , he remarked that:

> "... from their beginnings in Holland's research, [genetic algorithms] have inspired passion in their adherents. Perhaps this is because they surprise us in interesting ways and because they exemplify processes we believe may have lead to our own existence. Whatever the reason, there is something profoundly moving about linking a genetic algorithm to a difficult problem and returning later to find that the algorithm has evolved a solution that is better than the one a human found." [2]

Despite their seemingly unrelated origins, using principles from biology rather than engineering, in this work the ability for Genetic Algorithms to quickly find the global optimum in large and exceptionally multi-modal search spaces has been shown. The classical truss optimisation problem was implemented as a benchmark for the GA and with a search space of 2^{20} solutions, on average significantly less than 0.5% of potential solutions were evaluated in identifying the global optimum. However, considering that an exhaustive search of all 1,048,577 potential solutions for the Ten Bar Truss takes marginally over an hour, the advantages of a Genetic Algorithms seem to be outweighed by the effort involved in implementing or using them.

However, the example problems presented in this paper are relatively simple, with fitness functions which can evaluate an individual in milliseconds. In more realistic applications, such as the optimisation of large structures under many load cases, the evaluation of the fitness of a single potential design may take minutes or even hours.

Additionally, for problems with many variables and/or where many potential solutions exist, the size of the search space increases rapidly. Even a modest increase in the length of the bit-string, leads to an exponential growth in the size of the search space. The Ten Bar Truss uses a bit-string of length 20 to represent 10 elements with a choice of 4 cross-sections. For example, a 50% increase in bit-string length to 30 bits - perhaps greater choice of cross-sections or additional variables - would result in 1,073,741,825 potential solutions, a 1024% increase in search space. Even generously assuming the same average execution time of the Ten Bar Truss (5 milliseconds), evaluating a search space of this size would take 1492 hours or a little over 62 days. It is scenarios such as this where Genetic Algorithms truly become useful, where the difference in time between using a GA and an exhaustive search is measured in hours, days or even weeks. To continue the example, evaluating say 0.5% of the search space in 8 hours with a GA is infinitely more preferable to exhaustively searching 100% of solutions in a little over 2 months.

Nevertheless, the role of Genetic Algorithms in the optimisation of problems - let alone structural problems - is not restricted to those which would take a long time to find the optimum via brute-force. With a well formed fitness function which effectively implements the design criteria or requirements, a solution can often be found by a genetic algorithm which is better than that which could be manually designed - or at least found in less time.

Clearly a Genetic Algorithm is an effective optimisation method in terms of locating global optima both in terms of efficiency and efficacy. They have been used in a wide range of applications - from parametric design of aircraft to integrated systems design as well as process optimisation and scheduling [13, 2].

In the author's opinion, the major factor in the lack of use of optimisation methods in engineering is that of practicality. To apply a genetic algorithm to a problem, a fitness function must be created and, being the key to optimisation, it must be carefully developed. This is to ensure it reflects the design requirements and enforces design constraints. Additionally, it must be able to effectively evaluate the fitness of an individual design based upon the results of a FEM analysis. Currently if the design of a steel framed multi-storey structure was to be optimised using GAs, time and effort must be invested in developing a fitness function for this type of problem. To then subsequently optimise the design of a stadium or a suspension bridge, new fitness functions would need to be created.

Some Genetic Algorithm products do exist - such as *Evolver* add-in for Excel from Palisade - however these tend to be of limited use to those seeking to optimise problems outside of a spreadsheet environment. Similarly the author believes it is likely that Genetic Algorithms, as well as other optimisation methods, will gradually be subsumed into other software to become standard components in design packages. For optimisation methods to be of as much use to engineering as FEM is today, then they must be integrated with existing tools. For complicated problems, generic optimisation methods are likely to be of less practicality than those more specialised to the task at hand. For example, if an FEM analysis package integrated an structural design optimisation function then this is more immediately useful to the average engineer than some bespoke standalone solution. As with FEM, the need for education and training would be required to enable GAs to be used by engineers. However, hopefully much of detail on the operation of the GA would be abstracted away from the user - similar to how FEM is presented today.

12 Conclusions

12.1 Detailed Conclusions

As can be seen from the results, the Genetic Algorithm is able to quickly locate the optimum solution even in exceptionally complicated search spaces. In any given optimisation run, the percentage of possible solutions evaluated is negligible. For example, with the Ten Bar Truss, less than 0.17% of possible solutions were evaluated on average in the process of locating the optimum solution. Considering that the search space for the Ten Bar Truss is $2^{20} + 1 = 1048577$, this represents less than 2000 evaluations. The average optimisation run took approximately 10 seconds - compared to 68 minutes to evaluate every possible solution by brute force. Similar results were noted for the Simply Supported Beam, although approximately scaled for its significantly smaller search space. This remarkable performance is the key strength of Genetic Algorithms.

Both the truss and beam problems have relatively computationally trivial fitness functions, which can evaluate the fitness of an individual in milliseconds. Consider, however, if the Genetic Algorithm was applied to problems whose fitness function was more computationally demanding; for example if it called some complex mathematical model or finite element analysis and took several minutes to evaluate the fitness of an individual. The performance benefit of using a Genetic Algorithm and only needing to evaluate a small percentage of the search space is clear. Alternative methods such as evaluating every possible solution or randomly sampling the search space are time consuming and inefficient. Other optimisation approaches which use derivatives require the function to be differentiable and in many cases it many not be. Irregardless, even if the fitness function was differentiable, the point to point methods of many mathematical optimisation techniques are prone to locating local optima. Genetic Algorithms use no problem specific or auxiliary information, such as derivatives. They are highly adaptable, robust and implicitly parallel. The population based approach of a Genetic Algorithm, combined with the powerful heuristics which result from reproduction and selection, is able to efficiently locate the optimum using only similarities between highly fit solutions.

The parameters of a Genetic Algorithm have great effect upon the optimisation process. The population size used with a GA driven optimisation process is the key factor in determining its effectiveness. As the results show, the greater the population size, the more likely the algorithm is to converge to the optimum solution. Smaller population sizes reduces the diversity of potential solutions, reduces the mating pool for genetic operations and reduces the likelihood that an optimal solution will be evolved. For the fitness functions presented in this work, a large population size imposes minimal extra computational demand due to their triviality. Therefore for these problems, there is no advantage to using a smaller population size and this also introduces the risk that the algorithm may converge to a sub-optimal solution. However, again considering a problem whose fitness function is computationally expensive, a lower population will reduce optimisation times significantly, whilst at the risk of returning a sub-optimal solution. In this case it may be preferable to use a reasonably sized population, balancing the need for genetic diversity and a manageable optimisation time.

In terms of the genetic operators of the algorithm, mutation and elitism play a crucial but complementary role to that of crossover. In combination with selection, crossover

provides the Genetic Algorithm with the majority of its ability to locate the optimum solution. However, a limited rate of mutation ensures that the diversity of the population is maintained, introducing new novel genetic features into the gene pool. Elitism ensures that the fittest solutions are not accidentally lost in the evolutionary process and can dramatically increase the performance of the algorithm. The loss of a fit solution can cause regression in the overall fitness of the population and this has a detrimental effect on optimisation times.

The most effective GA parameters will vary from problem to problem and determining these will require some experimentation for a given problem. Nevertheless, in general, using a population size appropriate to the problem at hand as well as elitism and sparing mutation will ensure the GA is effective at solving a given problem.

The fitness functions are also a fundamental and central part of the optimisation process. The optimisation results from the Ten Bar Truss problem are generally sensible structures from a design point of view and this results from the design of the fitness function. The designs are driven by the implementation of the fitness function, which prioritises the minimisation of weight, constrained by certain structural considerations. The optimum result as seen in Figure 14 on page 46 is a reasonable structure considering the loads applied and the load paths; elements not essential to the support of the loads are a smaller section size than those which are. It should be emphasised that the algorithm has no structural design knowledge or understanding of load paths; it simply applies the measure of structural fitness as defined by the designer.

The beam optimisation results are also reasonably sensible designs, however some inconsistencies exist. Due to a lack of time to tackle the issue fully, beam designs rely entirely upon the capacity of the concrete to resist the loads applied. Whilst reinforcement is designed to support the maximum moment, this only ensures that adequate space is available in the cross-section to fit adequate reinforcement. The stiffness of the reinforcement is not taken into consideration in the design of the section. Therefore, the beam sections produced are unnecessarily deep since no account of the re-bar is made in the bending resistance of the section. However, the problem does serve as a proof of concept, showing that Genetic Algorithms can optimise problems with continuous variables, as well as those with discrete variables such as the truss.

On the whole, Genetic Algorithms have the potential to make a hugely effective contribution to engineering. The intelligent application of computers to solve design problems is, in the author's opinion, likely to be a field of increasing importance in the future of engineering.

12.2 General Conclusions

The objectives of this work were to develop a bespoke Genetic Algorithm from scratch, use it to solve design problems and, in doing so, investigate the effectiveness of GAs for the optimisation of structural systems. The author believes that the implementation of the Genetic Algorithm was a success, given it was implemented in a relatively short time in a language with which the author had limited experience. The problem agnostic design of the Genetic Algorithm enabled the author to rapidly implement the problem specific code required to investigate two different problems.

Admittedly the current need to develop problem specific code is prohibitive to more widespread use of Genetic Algorithms, in that the majority of engineers have no programming experience. However, it is a relatively small leap of imagination to suggest that if optimisation methods became part of Finite Element Modelling packages, then GAs would become another useful tool in the engineer's design toolbox. The ability for engineers to define an FEM model, define design requirements and arrive at an optimum solution for the given constraints could be invaluable. By replacing much of the manual iterative process (designing a solution, analysing it and then iteratively redesigning) with an intelligent & automated process which would instead answer the question: "what **should** the design be, in order to optimally satisfy the requirements placed upon the structure?". Nevertheless, optimisation methods will not replace engineers; common sense, design judgement, engineering knowledge and skills cannot be replaced by a computer program.

Today Finite Element Modelling is almost invaluable in many aspects of engineering. FEM enables structures to be designed faster, more efficiently and with increased accuracy over manual methods. It also permits more complicated and elaborate structures to be constructed with improved confidence in the design. With optimisation the ability to rapidly prototype many potential designs at the speed of a computer rather than manually is likely to be of tremendous advantage to engineering design.

This work has shown that whilst Genetic Algorithms are relatively simple to implement and use relatively simple methods, their complex, powerful and often impressive behaviour is particularly effective at optimisation. It is the authors opinion that optimisation will play an increasing role in the field of structural engineering as it becomes more accessible and more practical.

Whilst in some cases problem specific optimisation techniques may be more suitable, as an all round technique Genetic Algorithms outstrip most other optimisation techniques. Their ability to robustly handle exceptionally complicated problems, with discrete variables and non-linear interaction between variables, is almost unparallelled. GAs are remarkably intelligent in their traversal of even the most complex search space, evaluating as few potential solutions as possible in locating an optimum design. As the complexity of problems considered increases, the greater the attractiveness of Genetic Algorithms.

In summary, the author believes that there is great promise in using GAs as a design optimisation method for structural systems, in conjunction with existing tools.

12.3 Future perspectives

With the inexorable advance of technology and the exponential increase in computational power available, the use of computers is very likely to continue to increase in engineering design. Whilst this will enable more complicated FEM models to be analysed quicker or more detailed multi-scale modelling, it will also allow the application of optimisation to the design of large structures to become common place.

The fields of computational engineering and high performance computing represent a considerable opportunity to improve engineering. However to fully capitalise on these advances, new techniques and approaches must be considered. It would be - and is - a terrible misuse of the power of computers to simply transfer existing manual processes into an electronic form. Whilst FEM has almost entirely replaced manual analysis within engineering design, the advantages computers have to offer are yet to

be exploited fully. As computers work many times faster and more efficiently than humans can, with sufficient guidance, computers can evaluate many potential designs, locating the optimum design without requiring the constant attention of an engineer. By automating much of the laborious design process, more effective utilisation of resources and cost savings can be made.

Part V. References

References

[1] V.V. Toropov and S.Y. Mahfouz. Design optimization of structural steelwork using a genetic algorithm, fem and a system of design rules. *Engineering Computations*, 18(3/4):437–459, 2001.

[2] Lawrence Davis. *Handbook of Genetic Algorithms*. Van Norstrand Reinhold, 1991. ISBN 0-442-00173-8, 1-23 & 99-101.

[3] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press. ISBN 0-262-63185-7, 1-31.

[4] John H. Holland. *Adaption in Natural and Artificial Systems*. The University of Michigan, 1975. ISBN 0-262-58111-6, 1-32.

[5] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Company, 1989. ISBN 0-201-15767-5, 1-29 & 309-312.

[6] D.E. Goldberg and M. P. Samtani. Engineering optimization via genetic algorithm. *Proceedings of the Ninth Conference on Electronic Computation*, pages 471–482, 1987.

[7] Mathworks. Matlab product description page, 2009. http://www.mathworks.com/products/matlab/.

[8] Sweden Division of Structural & Solid Mechanics, Lund University, 2009. http://www.gorkon.byggmek.lth.se/Calfem/.

[9] C. Ellis. A bluffer's guide to genetic algorithms. *Engineering Design Newsletter, Science and Engineering Research Council*, 1993.

[10] S. Ravjeev and C.S. Krishnamoorthy. Discrete optimization of structures using genetic algorithm. *J.S. Engng, ASCE*, 118(5):1233–1250, May 1992.

[11] M.R. Ghasemi, E. Hinton, and R.D. Wood. Optimization of trusses using genetic algorithms for discrete and continuous variables. *Engineering Computations*, 6(3):272–301, 1999.

[12] British Standards Institution. *BS8110-1:1997 Structural use of concrete - Part 1: Code of Practice for design and construction*, 1997.

[13] Charles L. Karr and L. Michael Freeman. *Industrial Applications of Genetic Algorithms*. CRC Press. ISBN 0-849-39801-0, 1-14 & 35-48.

The reference list above is generated & formatted automatically using LaTeX 2_ε and the default BibTeX numerical style.

Part VI. Appendices

A Annotated GA Flowchart

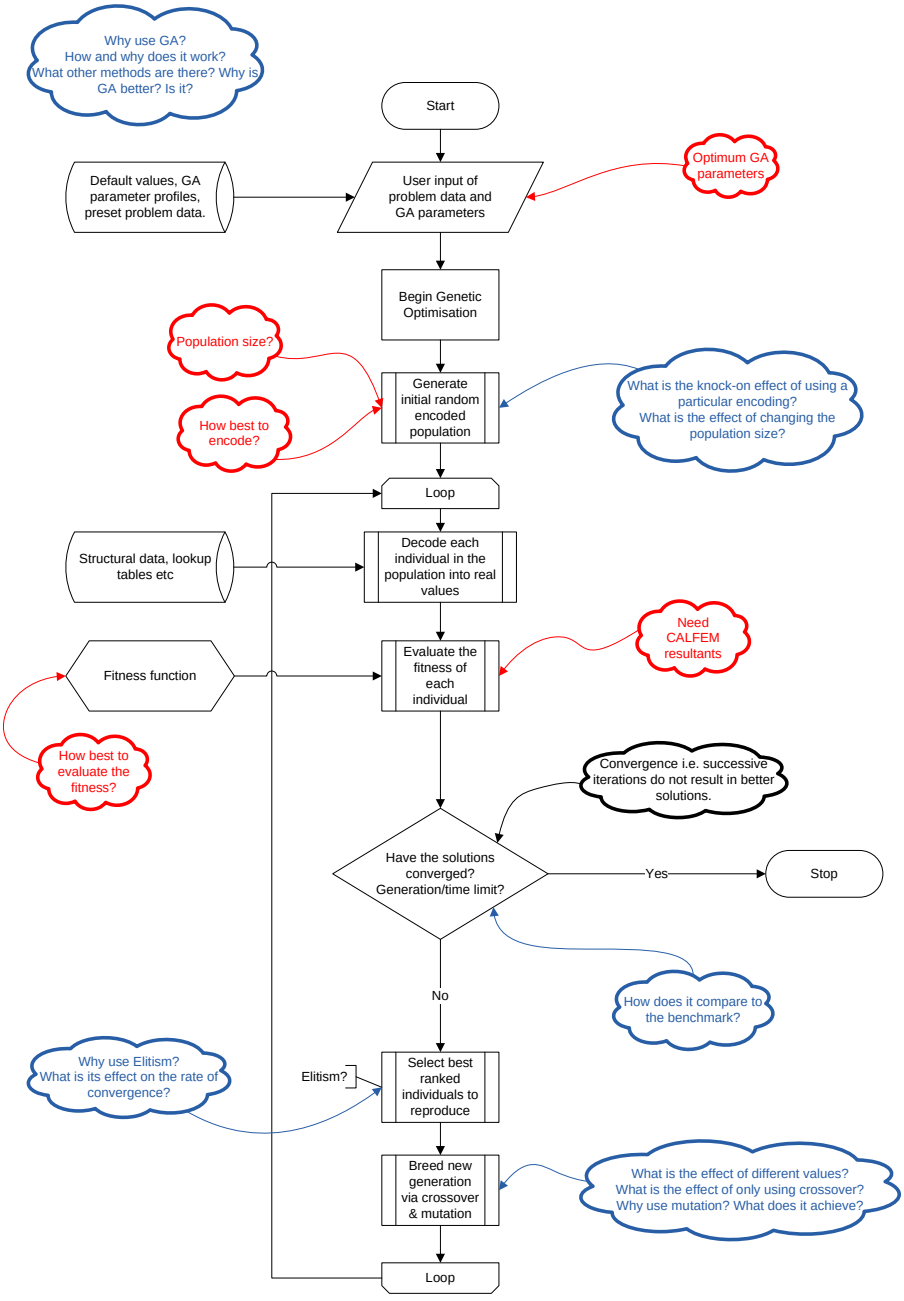

Fig. 21: Annotated GA Flowchart

B Ten Bar Truss weight fitness calculation

$Density = 7850kg/m^3$
$A_{min} = 11.2cm^2$
$A_{max} = 100cm^2$
$W = DAL$

Element	Length (m)	W_{min} (kg)	W_{max} (kg)
1	9.00	79.13	706.50
2	9.00	79.13	706.50
3	9.00	79.13	706.50
4	9.00	79.13	706.50
5	9.00	79.13	706.50
6	9.00	79.13	706.50
7	12.73	111.90	999.14
8	12.73	111.90	999.14
9	12.73	111.90	999.14
10	12.73	111.90	999.14
	$\sum$	**922.38**	**8235.57**

For the weight constant, say round max weight from 8235.57 to 9000?
$\Rightarrow$ Take weight constant as 9000.

Consequently, fitness values range from 764.3 to 8077.62 for the heaviest & lightest possible structures respectively.

C Typical Visualisations

C.1 Truss

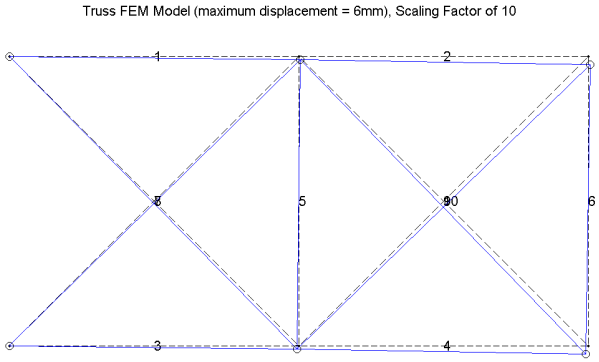

Fig. 22: Truss FEM Visualisation

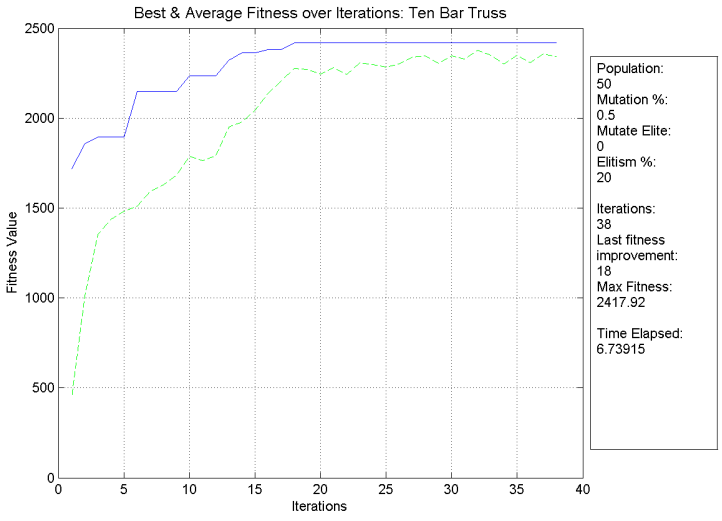

Fig. 23: Truss Fitness Graph over Iterations

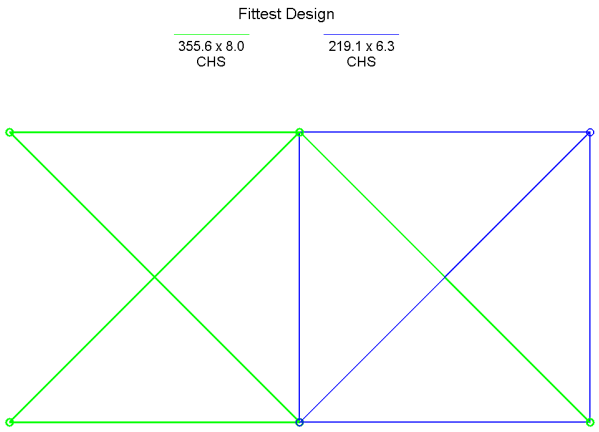

Fig. 24: Truss Final Design Visualisation

C.2 Beam

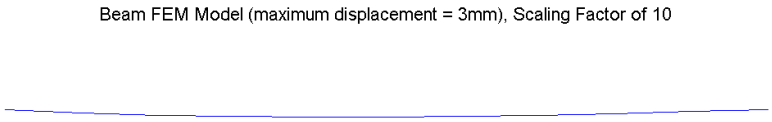

Fig. 25: Beam FEM Visualisation

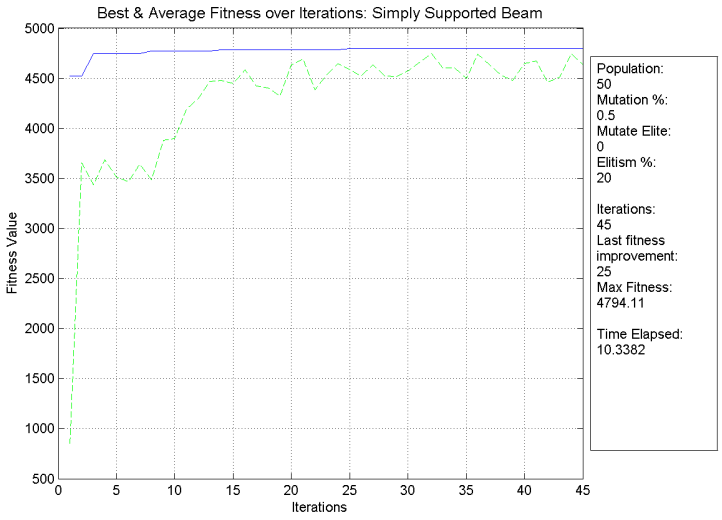

Fig. 26: Beam Fitness Graph over Iterations

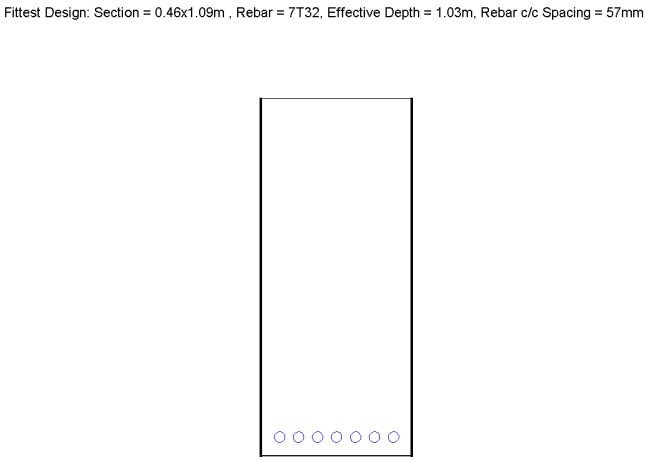

Fig. 27: Beam Final Design Visualisation

D GA MATLAB Code

- GA_Controller (page 66): Generic procedural code representing the standard GA algorithm.
- GA_Crossover (page 70): Implementation of genetic crossover.
- GA_Elitism (page 73):Selects top x number individuals from the population array to preserve for the next generation.
- GA_Mutation (page 72): Randomly mutate (i.e. invert) a percentage of bits in the new population.
- GA_SelectionRouletteWheel (page 74): Uses the roulette wheel selection paradigm to select individuals from population

D.1 /ga/GA_Controller.m

```
% Written by: Gavin Reynolds, 2008-2009

% Copyright (c) 2009 Gavin Reynolds
%
% This program is free software; you can redistribute it and/or
% modify it under the terms of the GNU General Public License
% as published by the Free Software Foundation; either version 3
% of the License, or (at your option) any later version.
%
% This program is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
% GNU General Public License for more details.

% You should have received a copy of the GNU General Public License
% along with this program.  If not, see <http://www.gnu.org/licenses/>.

%% GA Controller: Generic procedural code representing the standard Genetic algorithm.
function GA_Controller(problemType, maxNumIterations, populationSize, mutationPercent,
     elitismPercent, mutateElite, options)
%   1. Generate initial population
%   2. Loop until completion (as determined by this controller)
%       a) Evaluate fitness of the population members
%       b) Select fittest members of the population to reproduce
%       c) Give birth of offspring via genetic operations to form new pop.
%   3. Display and save results.
%Example: GA_Controller('Ten Bar Truss', 250, 40, 0.005, 0.05, 0)

if (exist('options', 'var') == 0)
    options = [];
end

try
    resultDir = evalin('base','resultDir');
catch ME
    resultDir = '';
end


%Ensuring that the population size is a multiple of 2.
if (mod(populationSize,2) ~= 0)
    %this will only be seen if using GA_Controller directly and the user specifies an
         odd population size
    disp('The population size must be a multiple of 2.')
    return;
end;

%% Set fitness function based on problem type
switch problemType
    case 'Ten Bar Truss'
        fitnessFunction = @fitness_Truss;
        sectionProperties = @getCALFEMtruss_sectionProperties;
        structureDiagram = @create_TrussDiagram;
        saveResultsAs = 'tenbartruss';
        %bitstringLength comes from 2 bits per design variable (i.e. cross-sectional
             area),
        %10 elements, therefore 2x10 = 20 char bitstring
        bitstringLength = 20;
        disp('Solving 10 element truss problem type as in Ghasemi et al.')

    case 'Simply Supported Beam'
        fitnessFunction = @fitness_Beam;
        sectionProperties = @getCALFEMbeam_sectionProperties;
        structureDiagram = @create_BeamDiagram;
        saveResultsAs = 'simplysupportedbeam';
        %bitstringLength comes from 8 bits per design variable (i.e. breadth and depth),
        %therefore, 2x8 = 16 char bitstring
        bitstringLength = 16;
        disp('Solving 3 element beam problem type.')

    otherwise
        disp('Unknown problem type')
        errordlg('Unknown problem type','Error');
        return
```

```
end

setFigures();

%% Generate initial random population
%A matrix of size populationSize by bitstringLength with values between 0 and 1
populationArray = logical(floor(rand(populationSize, bitstringLength) * (2)));
%populationArray = randint(populationSize, bitstringLength, [0 1]);

%% Begin loop, set termination conditions, allowing for early terminiation
numIterations = 0;
continueLoop = true;
countSinceLastFitnessImprovement = 0;
iterationBestFitnessArray = [];
tic; %start the clock

while continueLoop == true

    % Increment the iteration number (there is no increment operator...)
    numIterations = numIterations + 1;
    disp('Iteration:')
    disp(numIterations)

    disp('Population Array:')
    disp(populationArray)

    fitnessArray = zeros(populationSize,2, 'double');

    %% For each member/row of the population array, evaluate fitness
    for popNum=1:populationSize
        individual = populationArray(popNum, :);

        fitnessArray(popNum) = popNum;
        fitnessArray(popNum,2) = fitnessFunction(individual, popNum, options);
    end;

    %% GA Operations
    %sort the population array by the fitness value
    [sortedFitnessArray, sorter] = sort(fitnessArray(:, 2), 'descend');
    populationArray = populationArray(sorter,:);

    disp('Sorted Population:')
    disp(populationArray)

    fittestIndividual = populationArray(1,:);
    bestFitness = sortedFitnessArray(1,:);

    %Store the fittest individual if better than previous fitnesses
    if (bestFitness >= max(iterationBestFitnessArray))
        fittestIndividualOfRun = fittestIndividual;
    end

    disp('Fitness Array:')
    disp(fitnessArray(:,2))

    disp('Sorted Fitness Array:')
    disp(sortedFitnessArray)

    iterationBestFitnessArray(numIterations,:) = bestFitness;
    iterationAverageFitnessArray(numIterations,:) = mean(sortedFitnessArray);


    newPopRemaining = populationSize;
    %% Elitism
    %Elitism is specified as a percentage (0.0 to 1.0) of the population
    %Get x fittest members and add straight to new population.
    %The number selected via elitism must be multiples of 2.
    if elitismPercent ~= 0
        elitism = ceil(elitismPercent * populationSize/2)*2;
        [newPopulationArray, newPopRemaining] = GA_Elitism(elitism, populationArray,
            bitstringLength, newPopRemaining);
    else
        elitism = 0;
        newPopulationArray = populationArray;
    end;
```

```
    %% Selection/Survival of the Fittest
    % Using Roulette Wheel Selection
    % Calculate sum of all chromosome fitnesses
    totalFitness = sum(sortedFitnessArray);

    % Select the next population (minus the number selected through via
    % elitism). Those selected via elitism can still be selected again by
    % the roulette wheel
    selectedPopNumArray = zeros(newPopRemaining,1, 'double');
    for i=1:newPopRemaining
        [selectedPopNum] = GA_SelectionRouletteWheel(sortedFitnessArray, totalFitness,
             populationSize);
        selectedPopNumArray(i,:) = selectedPopNum;
    end;

    selectedPopulationArray = populationArray(selectedPopNumArray,:);

    %% Crossover
    %A random chance to get crossed over, and a random point within each
    %crossed over pair for crossover
    numCrossoverPairs = newPopRemaining/2;
    %disp('Crossover Pairs:')
    %disp (numCrossoverPairs)

    % Do the crossover
    [newPopulationArray] = GA_Crossover(numCrossoverPairs, bitstringLength,
         selectedPopulationArray, newPopulationArray);

    %% Mutation
    [newPopulationArray] = GA_Mutation(mutationPercent, elitism, mutateElite,
         populationSize, bitstringLength, newPopulationArray);

    %% New Population
    populationArray = newPopulationArray;

    %% Termination conditions
    if (numIterations == maxNumIterations), continueLoop = false; end

    if (numIterations > 1) && max(iterationBestFitnessArray) ==
         iterationBestFitnessArray(numIterations-1)
        %if the best fitness has not improved then increment the count
        countSinceLastFitnessImprovement = countSinceLastFitnessImprovement + 1;
    elseif (numIterations > 1) && max(iterationBestFitnessArray) ~=
         iterationBestFitnessArray(numIterations-1)
        %if the best fitness has improved, reset the count
        countSinceLastFitnessImprovement = 0;
    end;

    if countSinceLastFitnessImprovement == 20, continueLoop = false; end

end;

executionTime = toc;

disp('Num Iterations:')
disp(numIterations)
disp('Num Iterations since last fitness improvement:')
disp(countSinceLastFitnessImprovement)

%% Diagrams
%Fitness diagram
gaParameters = [populationSize; mutationPercent; elitismPercent; mutateElite];
results = [numIterations; countSinceLastFitnessImprovement; max(
     iterationBestFitnessArray); executionTime];
create_FitnessGraph(iterationBestFitnessArray, iterationAverageFitnessArray,
     gaParameters, results, problemType)

if (max(iterationBestFitnessArray)~=0)
    %Structure diagrams
    fittestSolution = sectionProperties(fittestIndividualOfRun, options);
    structureDiagram(fittestSolution, options);
end

%% Result saving
%No UNIX/Epoch time in MATLAB...
epochStart = [1970 01 01 0 0 0];
```

```
now =  clock();
%UNIX Time is a signed 64bit integer...
runTime = int64(floor(etime(now, epochStart)));

%Save results to csv form, in specified results dir if provided or into
%current working directory otherwise (i.e. if resultsDir var is blank)

save_Results(runTime, gaParameters, results, saveResultsAs, resultDir, options);

%Save figures to png images, in specified results dir if provided or into
%current working directory otherwise (i.e. if resultsDir var is blank)
saveFigures(runTime, 'png', saveResultsAs, resultDir);
```

D.2 /ga/GA_Crossover.m

```
%% GA_Crossover: Implementation of genetic crossover.
% Select x crossover pairs from the remaining population (or all if no
% elitism), for each pair of parents select a crossover point and swap the bits
% after the crossover point, creating the offspring/children.
% Written by: Gavin Reynolds, 2008-2009
function [newPopulationArray] = GA_Crossover(numCrossoverPairs, bitstringLength,
     selectedPopulationArray, newPopulationArray)

newPopRemaining = numCrossoverPairs*2;
avaliableParents = (1:newPopRemaining)';
for pairNum = 1:numCrossoverPairs

    %disp('Crossover Pair:')
    %disp (pairNum)

    %% Parent numbers
    %get the parent numbers, ensuring no identical numbers are selected

    parentNums(1,:) = floor(rand * (newPopRemaining)) + 1;
    %parentNums(1,:) = randint(1,1,[1 newPopRemaining]);
    j = floor(rand * (newPopRemaining)) + 1;
    %j = randint(1,1,[1 newPopRemaining]);
    while j == parentNums(1)
        j = floor(rand * (newPopRemaining)) + 1;
        %j = randint(1,1,[1 newPopRemaining]);
    end;
    parentNums(2,:) = j;

    %disp('Parent Nums:')
    %disp (parentNums)

    %% Crossover point
    %Crossover point is between 0 (i.e. parents swap all bits) to
    %bitStringLength i.e. nothing is swapped
    crossoverPoint = floor(rand * (bitstringLength+1));
    %crossoverPoint = randint(1,1,[0 bitstringLength]);

    %disp('Crossover Point:')
    %disp (crossoverPoint)

    parents = zeros(2, bitstringLength);
    %% Extracting parents
    for i = 1:2
        parents(i, :) = selectedPopulationArray(parentNums(i), :);
    end;

    % remove the two parents from the population array after they have been copied out
       ...
    avaliableParents(parentNums(1),:) = [];
    %ensure the correct row is removed, i.e. if the first row removed
    %was before the second row then remove the second row num - 1
    if parentNums(1) < parentNums(2)
        avaliableParents(parentNums(2)-1,:) = [];
    else
        avaliableParents(parentNums(2),:) = [];
    end;

    selectedPopulationArray = selectedPopulationArray(avaliableParents, :);

    newPopRemaining = newPopRemaining - 2;
    avaliableParents = (1:newPopRemaining)';

    %disp('Parents:')
    %disp (parents)

    %% Do the crossover

    switch crossoverPoint
        case {0, bitstringLength}
            %if the crossover point is 0 or the bitstringLength just
            %copy the parents through as the children. The order
            %makes no difference i.e. in the case of 0 where the
            %parents should really be switched, but doesn't matter
            children = parents;
```

```
        otherwise
            %Take the first crossoverPoint bits and transfer straight to children
            children = parents(:,1:crossoverPoint);
            %Swap the remaining bits
            children(1,crossoverPoint:bitstringLength) = parents(2,crossoverPoint:
                bitstringLength);
            children(2,crossoverPoint:bitstringLength) = parents(1,crossoverPoint:
                bitstringLength);
    end;

    %disp('Children:')
    %disp(children)
    %disp('=======')

    %append children to the new population array
    newPopulationArray = logical(vertcat(newPopulationArray,children));
end;
end
```

D.3 /ga/GA_Mutation.m

```
%% GA_Mutation: Randomly mutate (i.e. invert) a percentage of bits in the new population
      array.
% Introduces new novel genetic features into the population and maintains genetic
     diversity.
% Written by: Gavin Reynolds, 2008-2009
function [newPopulationArray] = GA_Mutation(mutationPercent, elitism, mutateElite,
     populationSize, bitstringLength, newPopulationArray)

%Calculate the number of bits to mutate
mutateNumBits = ceil(mutationPercent*populationSize*bitstringLength);

%Set mutation limits
%If mutateElite is off, stop mutation of the Elites by setting the
%rantint limits to 1 + the number of elites
if mutateElite == 0
    mutateRowStart = 1 + elitism;
else
    mutateRowStart = 1;
end;

%Select mutateNumBits bits at random and invert value
for i = 1:mutateNumBits
    mutateRow = floor(rand * (populationSize+1-mutateRowStart)) + mutateRowStart;
    %mutateRow = randint(1,1,[mutateRowStart bitstringLength]);
    mutateCol = floor(rand * (bitstringLength))+1;
    %mutateCol = randint(1,1,[1 populationSize]);
    currentBitValue = newPopulationArray(mutateRow, mutateCol);
    switch currentBitValue
        case 0
            newBitValue = 1;
        case 1
            newBitValue = 0;
    end;
    newPopulationArray(mutateRow, mutateCol) = newBitValue;
end;
end
```

D.4 /ga/GA_Elitism.m

```
%% GA_Elitism: Selects top x number individuals from the population array to preserve
     for the next generation.
% Written by: Gavin Reynolds, 2008-2009
function [newPopulationArray, newPopRemaining] = GA_Elitism(elitism, populationArray,
     bitstringLength, newPopRemaining)

%create new population array. false() = logical(zeros())
newPopulationArray = false(elitism, bitstringLength);
for eliteNum = 1:elitism
    newPopulationArray(eliteNum,:) = populationArray(eliteNum,:);
end;
newPopRemaining = newPopRemaining - elitism;
%disp (newPopulationArray)
end
```

D.5 /ga/GA_SelectionRouletteWheel.m

```
%% GA_SelectionRouletteWheel: Uses the roulette wheel selection paradigm to select
     individuals from population
% The greater the fitness the greater the proportion of the "wheel" an individual is
     given therefore greater chance of being selected.
% However, does not exclude lower fitness individuals from being selected i.e. they have
      a lower chance of being selected
% Written by: Gavin Reynolds, 2008-2009
function [popNum] = GA_SelectionRouletteWheel(sortedFitnessArray, totalFitness,
     populationSize)

r = rand*totalFitness;
s = 0;
for popNum=1:populationSize
    %Go through the population and incrementally sum fitnesses
    %until equal to or greater than r
    s = s + sortedFitnessArray(popNum,:);
    if (s >= r)
        %return this population number
        return
    end;
end;
end
```

E Problem Specific MATLAB Code

- Ten Bar Truss
  - fitness_Truss.m (page 76): Returns an objective fitness value for the supplied individual truss design
  - solver_CALFEMtruss.m (page 78): CALFEM Solver for 10 element truss, adapted from CALFEM example exs4
  - getCALFEMtruss_geometry.m (page 80): Static function providing geometry for the 10 element truss.
  - getCALFEMtruss_sectionProperties.m (page 81): Returns section properties for the 10 elements based upon the bit-string of an individual
- Simply Supported Beam
  - fitness_Beam.m (page 83): Returns an objective fitness value for the supplied individual beam design
  - solver_CALFEMbeam.m (page 85): CALFEM Beam solver, adapted from CALFEM example exs3
  - getCALFEMbeam_sectionProperties.m (page 87): Returns section size based upon the bit-string of an individual
  - getCALFEMbeam_reinforcement.m (page 88): Get CALFEM beam reinforcement solution for a given breadth, depth and maximum moment

E.1 Ten Bar Truss

E.1.1 /fitness/fitness_Truss.m

```
%% fitnessTenBarTruss: Returns an objective fitness value for the supplied individual
     truss design
% Written by: Gavin Reynolds, 2008-2009
function [fitness] = fitness_Truss(individual, popNum, options)

elementArray = getCALFEMtruss_sectionProperties(individual);

%Solver_CALFEMtruss solves for and returns:
%a: Displacement (including boundary values) (m)
%Q: Reaction force vector (kN)
%N: Element forces (kN)
%O: Stresses (kN/m^2)
%W: Weight (kg)

%Additionally, for convenience, it returns the following constants:
%L: Element Length (m)
%E: Youngs Modulus (currently a scalar) (kN/m2)
[a,Q,N,O,W,L,E]=solver_CALFEMtruss(elementArray, popNum, options);

%% Euler Buckling Load check
% Effective length = Length
% Pinjointed => K=1.0
% Have load, length, therefore Euler Buckling Load
% Two possible methods:
% 1) F=(pi^2*E*I)/(K*L)^2
% 2) Assuming K=1.0, O = F/A = (pi^2*E)/(L/r)^2 where O = Stress
% => F = ((pi^2*E)/(L/r)^2)*A
%
%Using method 1...
%r = cell2mat(elementArray(:,3));
I = cell2mat(elementArray(:,2));

eulerF = (pi^2*E.*I)./(1.0.*L).^2;

%Method 2:
%A = elementArray(:,1);
%F = ((pi^2*E)./(L./r).^2).*A;

%Using Relational operators, to perform an element by element comparison of
%the two arrays. Returns a logical array of the same size, with elements set to logical
     1 (true)
%where the relation is true, and elements set to logical 0 (false) where it is not.

isBuckled = abs(N) >= eulerF;
numElementsBuckled = sum(isBuckled);
numElementsBuckledPenalty = (1-numElementsBuckled/10)^2;

bucklingRatio=zeros(10,1, 'double');
for i=1:10
    elementBucklingRatio = abs(N(i,:))/eulerF(i,:);
    if (elementBucklingRatio > 1)
        elementBucklingRatio = 1;
    end;
    bucklingRatio(i) = elementBucklingRatio;
end;

elementFitness = (1-bucklingRatio).^2;

bucklingFitness = sqrt(sum(elementFitness)/10)*numElementsBuckledPenalty;

%% Yield Stress Check
%An approximate Steel Yield Stress is 250MPa = 250N/m2
%Corus Celsius 355 products have Yield Stress 355Mpa = 355e9 N/m2 = 355e6 kN/m2
yieldStress = 355e6;
isYielded = abs(O) >= yieldStress;
numElementsYielded = sum(isYielded);
numElementsYieldedPenalty = (1-numElementsYielded/10)^2;

yieldingRatio = zeros(10,1, 'double');
for i=1:10
    elementYieldingRatio = abs(O(i,:))/yieldStress;
```

```
    if (elementYieldingRatio > 1)
        elementYieldingRatio = 1;
    end;
    yieldingRatio(i) = elementYieldingRatio;
end;

elementFitness = (1-yieldingRatio).^2;
yieldingFitness = sqrt(sum(elementFitness)/10)*numElementsYieldedPenalty;

%% Displacement limit
%Say use two limits, approximately ULS and SLS?
%Displacements over ULS limit penalised severely.
%Displacements between SLS and ULS limits penalised.
%Displacements under SLS limit OK.
ulsDisplacementLimit = 60e-3;
slsDisplacementLimit = 30e-3;

maxDisplacement = max(abs(a));

if (maxDisplacement <= slsDisplacementLimit)
    displacementFitness = 1.0;
elseif (maxDisplacement > slsDisplacementLimit && maxDisplacement < ulsDisplacementLimit
    )
    displacementFitness = (ulsDisplacementLimit - maxDisplacement)/slsDisplacementLimit;
else
     displacementFitness = 0;
end;

%% Fitness calculation
weightFitness = 9000-sum(W);

fitness = weightFitness * displacementFitness * yieldingFitness * bucklingFitness;
end
```

E.1.2 /solvers/solver_CALFEMtruss.m

```
%% CALFEM Solver for 10 element truss, adapted from CALFEM example exs4
% Adapted and developed by: Gavin Reynolds, 2008-2009
function [a,Q,N,O,W,L,E]=solver_CALFEMtruss(elementArray, popNum, options)
% return a - Displacements (m)
% return Q - Reactions (kN)
% return N - Normal forces (kN)
% return O - Stresses (kN/m^2)
% return W - Weight (kg)
% return L - Length of elements (m)
% return E - Youngs Modulus (kN/m2)

A = cell2mat(elementArray(:,1));

%STEEL Young's Modulus is 210GPa = 210e9 N/m2 = 210e6 kN/m2
E = 210e6;
% STEEL Density is 7850 kg/m3 => Weight = Density * Volume
D = 7850;

%----- Get geometry of the structure from include ---------------

[Edof, Ex, Ey]= getCALFEMtruss_geometry();

%----- Stiffness matrix K and load vector f ---------------------

K=zeros(12);
%Force in kN
%options Array contains nodal forces in [DOF4 DOF8] format

f=zeros(12,1);

if (isempty(options))
    %no options provided therefore use defaults
    f(4)=-450;  f(8)=-450;
else
    %options provided therefore use specified loads
    f(4) = options(1)*-1; f(8) = options(2)*-1;
end

%----- Assemble Ke into K ---------------------------------------

for i=1:10
    %passing ep values for each element (E is constant, A is variable)
    ep=[E A(i)];
    Ke=bar2e(Ex(i,:),Ey(i,:),ep);
    K=assem(Edof(i,:),K,Ke);
end;

%----- Solve the system of equations ----------------------------

boundaryConditions = [9 0;10 0;11 0;12 0];
[a,Q]=solveq(K,f,boundaryConditions);

%----- Element forces -------------------------------------------

Ed=extract(Edof,a);

% Preallocating the N array, else it will grow incrementally...
N=zeros(10,1, 'double');

for i=1:10
    ep=[E A(i)];
    N(i,:)=bar2s(Ex(i,:),Ey(i,:),ep,Ed(i,:)); %(kN)
end


%----- Element stresses -----------------------------------------

O = N./A; %kN/m^2

%----- Element weight -------------------------------------------

%Length = sqrt((x2-x1)+(y2-y1))
L = sqrt((Ex(:,2) - Ex(:,1)).^2 + (Ey(:,2) - Ey(:,1)).^2); %(m)
```

```
%Weight = Density * Volume = Density * Length * Area
W = D.*A.*L; %(kg)

%----- Graph the structure -------------------------------------
if (popNum == 1)
    %Draw the displacements if this is the first x members of the population

    figure(1), clf, axis('equal'), hold on, axis off
    maxDisp = num2str(round(max(a)*1000)); %mm
    title(['Truss FEM Model (maximum displacement = ', maxDisp, 'mm), Scaling Factor of
        10'], 'FontSize', 10);
    %draw original and annotate with element numbers
    plotpar=[2 1 0];
    elnum=Edof(:,1);
    eldraw2(Ex,Ey,plotpar,elnum);

    %draw deformed
    plotpar=[1 2 1];
    eldisp2(Ex,Ey,Ed,plotpar, 10);

end;

%---------------------------- end -------------------------------
```

E.1.3 /includes/getCALFEMtruss_geometry.m

```
%% getCALFEMtruss_geometry: Static function providing Geometry for 10 element truss
     example in Ghasemi et al.
% Written by: Gavin Reynolds, 2008-2009
function [Edof, Ex, Ey]=getCALFEMtruss_geometry()
%This static data has been moved to an include to allow for access
%to geometry data without global variables or duplicating it in code

%----- Topology matrix Edof -------------------------------------

%----- Num A_Dof1 A_Dof2 B_Dof1 B_Dof2
Edof= [1   9  10  5  6;
    2   5  6  1  2;
    3   11 12 7  8;
    4   7  8  3  4;
    5   7  8  5  6;
    6   3  4  1  2;
    7   9  10 7  8;
    8   11 12 5  6;
    9   5  6  3  4;
    10   7  8  1  2];


%----- Element coordinates --------------------------------------

% Using a global coordinate matrix, a global topology matrix and coordxtr
% to get Ex and Ey

Coord= [18 9;
    18 0;
    9 9;
    9 0;
    0 9;
    0 0];

Dof=[1 2;
    3 4;
    5 6;
    7 8;
    9 10;
    11 12];

[Ex,Ey]=coordxtr(Edof,Coord,Dof,2);
```

E.1.4 /includes/getCALFEMtruss_sectionProperties.m

```
%% getCALFEMtrusssectionProperties: Provides section properties specified by the
     bitstring of the individual provided
% Written by: Gavin Reynolds, 2008-2009
function [elementArray] = getCALFEMtruss_sectionProperties(individual, options)

% Preallocating the elementArray, else it will grow incrementally...
elementArray=cell(10, 5);

for elementNum=1:10
    b = elementNum*2;
    a = b - (2-1);

    %Original method:
    %     Not a particularly elegant way of doing this...
    %     elementString = regexprep(num2str(individual(1, a:b)),'[^01]','');
    %     crossSectionNum = bin2dec(elementString) + 1;

    %better way of doing this...
    % on average more than halves the execution time, the num2str matlab function is
         incredibly
    % slow for that it has to do and was a major performance bottleneck. using sprintf
         instead.
    crossSectionNum = bin2dec(sprintf('%-1d',individual(a:b))) + 1;

    %% populate the array of cross-section areas via the structural lookup
    %All sections below are Corus Celsius 355 CHS, conforming to
    %(EN10210:2006, Part 1: S355J2H).
    % A: Area (cm^2)
    % I: Moment of Inertia/Second Moment of area (cm^4)
    % r: Radius of gyration (cm)
    % D: Diameter of the section (mm)
    switch crossSectionNum
        case 1
            %406.4 x 8.0 CHS
            A = 100.0;
            I = 19874;
            r = 14.1;
            D = 406.4;
            desc = '406.4 x 8.0 CHS';
            crossSectionNum = 1;
        case 2
            %355.6 x 8.0 CHS
            A = 87.4;
            I = 13201;
            r = 12.3;
            D = 355.6;
            desc = '355.6 x 8.0 CHS';
            crossSectionNum = 2;
       case 3
            %219.1 x 6.3 CHS
            A = 42.1;
            I = 2386;
            r = 7.53;
            D = 219.1;
            desc = '219.1 x 6.3 CHS';
            crossSectionNum = 3;
        case 4
            %114.3 x 3.2 CHS
            A = 11.2;
            I = 172.0;
            r = 3.93;
            D = 114.3;
            desc = '114.3 x 3.2 CHS';
            crossSectionNum = 4;
    end;

    %Unit conversions from cm based units to m
    I = I * 1e-8; % (cm^4 to m^4)
    A = A * 1e-4; % (cm^2 to m^2)
    r = r * 1e-2; % (cm to m)
    D = D * 1e-3; % (mm to m)

    elementArray(elementNum,1) = {A};
    elementArray(elementNum,2) = {I};
```

```
    elementArray(elementNum,3) = {r};
    elementArray(elementNum,4) = {D};
    elementArray(elementNum,5) = {desc};
    elementArray(elementNum,6) = {crossSectionNum};

end;
```

E.2 Simply Supported Beam

E.2.1 /fitness/fitness_Beam.m

```
%% fitness_Beam: Returns an objective fitness value for the supplied individual beam
     design
% Written by: Gavin Reynolds, 2008-2009
function [fitness] = fitness_Beam(individual, popNum, options)

[sectionProperties] = getCALFEMbeam_sectionProperties(individual, options);
b = sectionProperties(1,1);
h = sectionProperties(1,2);

if (b == 0 || h == 0)
    fitness = 0;
    return
end

[a, Q, es, W, D, L]=solver_CALFEMbeam(b,h,popNum, options);

%es is cell array of individual element [N V M];

elementNVMArray = cell2mat(es);
elementMArray = elementNVMArray(:,3);
maxM = max(abs(elementMArray));

%% Reinforcement

[rebarFitness, numBars, barDiameter, maxMomentCapacityFitness] =
     getCALFEMbeam_reinforcement(b,h,maxM);

if (rebarFitness == 0)
    %Reinforcement solution does not exist
    fitness = 0;
    return;
end;


%% Plastic Moment
%Mp  = (b*h^2)/4*yieldStress
%yield stress = say, 30MPa =  30e6 kN/m2
yieldStress = 30e6;
Mp = (b*h^2)/4*yieldStress; %kNm

plasticMomentFitness = 1 - (maxM/Mp);
if (plasticMomentFitness < 0)
    plasticMomentFitness = 0;
end


%% Displacement limit
%Say use two limits, approximately ULS and SLS?
%Displacements over ULS limit penalised severely.
%Displacements between SLS and ULS limits penalised.
%Displacements under SLS limit OK.

%using BS8110, limiting deflection to span/250 and span/500, C13.4.6.3

if (numel(options) == 0)
    L = 9;
else
    L = options(5);
end

%Deflection limits stated in BS8110 CL 4.3.6.3
ulsDisplacementLimit = L/250;
slsDisplacementLimit = L/500;

maxDisplacement = max(abs(a));

if (maxDisplacement <= slsDisplacementLimit)
    displacementFitness = 1.0;
elseif (maxDisplacement > slsDisplacementLimit && maxDisplacement < ulsDisplacementLimit
     )
    displacementFitness = (ulsDisplacementLimit - maxDisplacement)/slsDisplacementLimit;
```

```
else
     displacementFitness = 0;
end;

%% Weight Fitness

if (numel(options) == 0)
    %maxh = 1.5;
    %maxb = 1.5;

    %weightConstant = D * maxh * maxb; %W=DA = 2400 * 1.5 * 1.5 = 48600
    %weightConstant = ceil(weightConstant/1000)*1000; % rounded upwards to nearest
         1,000;
    weightConstant = 6000;
else
    maxh = options(2);
    maxb = options(4);
    weightConstant = ceil(D * maxh * maxb/1000)*1000; % W=DAL rounded upwards to nearest
         1,000;
end

weightFitness = weightConstant - W;

%% Fitness calculation
fitness = weightFitness * displacementFitness * plasticMomentFitness *
     maxMomentCapacityFitness * rebarFitness;

end
```

E.2.2 /solvers/solver_CALFEMbeam.m

```
%% CALFEM Beam solver, adapted from CALFEM example exs3
% Adapted and developed by: Gavin Reynolds, 2008-2009
function [a, Q, es, W, D, L]=solver_CALFEMbeam(b,h,popNum,options)
%Finite element model of concrete beam (model of 3 finite elements)
% return a - Displacements (m)
% return Q - Reactions (kN)
% return es - [N V M] array for each finite element
% return W - Weight per unit length (kg/m)
% return L - Length of beam (m)

%----- Setup ---------------------------------------------------

%Concrete Young's Modulus is 30GPa = 30e9 N/m2 = 30e6 kN/m2
E = 30e6;
% Concrete Density is 2400 kg/m3 => Weight = Density * Volume
D = 2400;

%Number of finite elements to use for the model
numElements = 10;

if (numel(options) == 0)
    L = 9;
else
    L = options(5);
end

elementSize = L/numElements;

%----- Topology ------------------------------------------------

% Edof=[1  1  2  3  4  5  6;
%     2  4  5  6  7  8  9;
%     3  7  8  9 10 11 12];

Edof = zeros(numElements,7);

for i = 1:numElements
    Edof(i,:) = [i (i-1)*3+1 (i-1)*3+2 (i-1)*3+3 (i-1)*3+4 (i-1)*3+5 (i-1)*3+6];
end;
numDofs = Edof(end:end);
%----- Stiffness matrix K and load vector f ---------------------

K=zeros(numDofs);       f=zeros(numDofs,1);


if (numel(options) == 0)
    %use default loads
    f(5)=-1000; f(8)=-1000; %kN

else
    %options provided therefore use specified loads
    %loads are specified in pairs of values, a location & a load.
    %locations are not necessarily on FEM nodal points
    numLoadPairs = (numel(options)-5)/2;

    for i = 1:numLoadPairs
        location = options(6+(i-1)*2);

        %sign convention on input is downward load is positive, therefore,
        %reverse sign for CALFEM
        loadAtLocation = options(7+(i-1)*2) * -1;

        %workout where on the beam the load is placed, in terms of finite
        %elements i.e. 3.25 indicates between element 3 and 4, 25% of
        %finite element length from node 3.
        nodalLocation = location/elementSize + 1;

        %convention for each element is working "left to right"
        %distributes load proportionally to LH & RHS nodes,
        % (or directly onto the appropriate node if load is exactly on an FEM node)
        proportionLeft = nodalLocation - floor(nodalLocation);
        proportionRight = 1 - proportionLeft;

        leftNode = floor(nodalLocation);
```

```
        verticalDofAtLeft = 2 + (leftNode-1)*3;
        verticalDofAtRight = verticalDofAtLeft + 3;

        f(verticalDofAtLeft) = f(verticalDofAtLeft) + loadAtLocation*proportionLeft;
        f(verticalDofAtRight) = f(verticalDofAtRight) + loadAtLocation*proportionRight;
    end

end


%----- Element stiffness matrices  -----------------------------

A=b*h;     I=(b*h^3)/12;      ep=[E A I];
ex=[0 elementSize];     ey=[0 0];

Ke=beam2e(ex,ey,ep);

%----- Assemble Ke into K ---------------------------------------

K=assem(Edof,K,Ke);

%----- Solve the system of equations and compute reactions ------

verticalDofAtRHSupport = numDofs-1;
bc=[1 0; 2 0; verticalDofAtRHSupport 0];

[a,Q]=solveq(K,f,bc);

%----- Section forces -------------------------------------------

Ed=extract(Edof,a);


for i = 1:numElements
    es{i,:}= beam2s(ex,ey,ep,Ed(i,:));
end

%es = [N V M];

%----- Weight- --------------------------------------------------

%Weight per unit length = Density * Volume = Density * Area
W = D.*A; %(kg)

%----- Graph the structure --------------------------------------
if (popNum == 1)
    figure(1), clf, axis('equal'), hold on, axis off
    maxDisp = num2str(round(max(a)*1000)); %mm
    title(['Beam FEM Model (maximum displacement = ', maxDisp, 'mm), Scaling Factor of
         10'], 'FontSize', 10);
    scalingFactor = 10;

    %need to get the verticaldisplacements corresponding
    %to the vertical displacements of the nodes.
    numNodes = numElements+1;
    for i=1:(numNodes)
        X(:,i) = elementSize * (i-1);

        verticalDOFForNode = 2+3*(i-1);
        Y(:,i) = a(verticalDOFForNode) * scalingFactor;

    end;

    line(X,Y);
end;
%------------------------ end ----------------------------------
```

E.2.3 /includes/getCALFEMbeam_sectionProperties.m

```
%% getCALFEMbeam_sectionProperties: Provides section size specified by the bitstring of
     the individual provided
% Written by: Gavin Reynolds, 2008-2009
function [sectionProperties] = getCALFEMbeam_sectionProperties(individual, options)
%2^8 possible breadths = 256
%2^8 possible depths = 256
%approximately to 5mm for the default max & min dimensions

if (numel(options) == 0)
    %default case
    minh = 0;
    maxh = 1.5;
    minb = 0;
    maxb = 1.5;

else
    %use specified max & min dimensions
    minh = options(1);
    maxh = options(2);
    minb = options(3);
    maxb = options(4);
end

heightInterval = (maxh-minh)/256;
breadthInterval = (maxb-minb)/256;

breadthNum = (bin2dec(sprintf('%-1d',individual(1:8))));
heightNum = (bin2dec(sprintf('%-1d',individual(9:16))));

%round upwards to nearest 10mm
b = ceil(breadthNum * breadthInterval/ 0.01)*0.01 + minb;
h = ceil(heightNum * heightInterval/ 0.01)*0.01 + minh;

%returning as an array of b & h to stay consistent with the number of variables
%returned by the truss section properties include, as the GA code is
%generic so don't want to get into handling special cases etc in the
%main GA code

sectionProperties = [b,h];
```

E.2.4 /includes/getCALFEMbeam_reinforcement.m

```
% getCALFEMbeam_reinforcement: Get CALFEM beam reinforcement properties
% Written by: Gavin Reynolds, 2008-2009
function [rebarFitness, numBars, barDiameter, maxMomentCapacityFitness, d,
     barCentertoCenterSpacing, linkSize, coverToLink] = getCALFEMbeam_reinforcement(b,h,
     maxM)

minSpacing = 25;
linkSize = 16;
coverToLink = 25;
avaliableBars = [16; 20; 25; 32; 40;]; %mm

%need to guess at bar diameter to get cover distance, taking mean of
%avaliable bars as a rough estimate?

coverToRebarGuess = ceil((coverToLink + linkSize + mean(avaliableBars)/2) / 10)*10; %mm
d = h - coverToRebarGuess/1000; %m

fcu = 40; %kN/m2
fy = 460;

%Mr = 0.156b*d^2*fcu
Mr = 0.156 * (b*1000) * (d*1000)^2 * fcu  / 1e6; %kNm

%initially assume reinforcement design to be ok, unless proved otherwise
reinforcementDesignCalcsOK = true;

maxMomentCapacityFitness = 1/(maxM/Mr);
if (maxMomentCapacityFitness > 1)
    maxMomentCapacityFitness = 1;
end;

%% Do reinforcement design
K = maxM*1e6/((b*1000)*(d*1000)^2*fcu);

%if coefficient K is greater than 0.156 then design is invalid
if (K >= 0.156)
    reinforcementDesignCalcsOK = false;
end;
z = (0.5+sqrt(0.25-K/0.9))*d; %m
%if lever arm exceeds limit of 0.95d, then design is invalid
if (z >= 0.95*d)
    reinforcementDesignCalcsOK = false;
end;
x = (d-z)/0.45; %m

%if neutral axis is outside these limits then design is invalid
if (x <= 0.1*d || x >= 0.5*d)
    reinforcementDesignCalcsOK = false;
end;

%if reinforcement design has failed for any reason, return variables are 0
if (reinforcementDesignCalcsOK == false)
    rebarFitness = 0; numBars = 0; barDiameter = 0; maxMomentCapacityFitness = 0; d = 0;
          barCentertoCenterSpacing =0;
    return;
end;

As = maxM*1e6/(0.95*fy*z*1000); %mm2
minAs = 0.13/100*(b*1000)*(h*1000);

%if the required area of steel is less than the minimum steel required for
%the section size, then use the minimum value
if (As < minAs)
    As = minAs;
end;

%% Get required num bars & diameters

loop = true;
j = 1;
numBars = 1;
sectionWidth = (b*1000);
numAvaliableBars = numel(avaliableBars);
while (loop == true)
```

```
    barDiameter = avaliableBars(j);
    barArea = pi()*(barDiameter/2)^2;
    numBars = ceil(As/barArea);

    requiredWidth = numBars * barDiameter + 2*linkSize + (numBars + 1) * minSpacing; %mm

    if (requiredWidth <= sectionWidth)
        %rebar fits the section
        loop = false;
        rebarSolutionFound = true;
    else
        %try the next size of bar up
        j = j + 1;
        if (j > numAvaliableBars)
            loop = false;
            rebarSolutionFound = false;
        end;
    end;
end;

%check coverToRebar was suitable?
coverToRebar = (coverToLink + linkSize + barDiameter/2);

%calculate the center to center spacing to get the rebar equi-distant get space
totalWidthofSteel = (b*1000 - (2*coverToLink + 2*linkSize + numBars*barDiameter));
barCentertoCenterSpacing = totalWidthofSteel/(numBars - 1) + barDiameter; %mm

rebarFitness = 0;

if (coverToRebar <= coverToRebarGuess)
    rebarFitness = rebarFitness + 0.25;
    disp('Rebar Cover Guess OK');
end;

if (barCentertoCenterSpacing >= minSpacing)
    rebarFitness = rebarFitness + 0.25;
    disp('Bar Spacing OK');
end;

if (rebarSolutionFound == true && rebarFitness > 0)
    %OK
    disp('Rebar Solution FOUND');
    rebarFitness = rebarFitness + 0.50;
end;

end
```

F Misc. MATLAB Code

- Figures
  - create_FitnessGraph.m (page 90): Graphs the best and average fitness of the population over generations.
  - create_TrussDiagram.m (page 92): Generates the visualisation of the fittest truss design.
  - create_BeamDiagram.m (page 94): Generates the visualisation of the fittest beam design.
- Search Space Mapping
  - fitnessMap.m (page 95): A wrapper for the fitness functions, used to generate fitness landscape plots and also time how long it takes to evaluate the fitness of every possible combination.

F.1 Figures

F.1.1 /figures/create_FitnessGraph.m

```
%% create_FitnessGraph: Creates a fitness over iterations graph
% Written by: Gavin Reynolds, 2008-2009
function create_FitnessGraph(iterationBestFitnessArray, iterationAverageFitnessArray,
    gaParameters, results, problemType)
%% Create the fitness graph
% Create figure
figure2 = figure(2);
clf;
% Create axes
axes1 = axes('Parent',figure2,'YGrid','on','XGrid','on','Position',[0.11 0.10 0.70
    0.80]);
box('on');
hold('all');

numIterations = results(1);
x = (1:numIterations)';
% Create plot
plot(x, iterationBestFitnessArray(:,1),'-b');
plot(x, iterationAverageFitnessArray(:,1),'--g');
titleText = ['Best & Average Fitness over Iterations: ', problemType];

% Create title
title(titleText,'FontSize',12);

% Create xlabel
xlabel('Iterations');

% Create ylabel
ylabel('Fitness Value');

% Create light
light('Parent',axes1,'Position',[-0.03 1.0 0.001]);

populationSize = gaParameters(1);
mutationPercent  = gaParameters(2)*100;
elitismPercent = gaParameters(3)*100;
mutateElite = gaParameters(4);


countSinceLastFitnessImprovement = results(2);
lastFitnessImprovement = numIterations - countSinceLastFitnessImprovement;

maxFitness = results(3);
```

```
executionTime = results(4);

% Create textbox
solutionDesc = {'Population:', populationSize, 'Mutation %:', mutationPercent, 'Mutate
    Elite:', mutateElite, 'Elitism %:', elitismPercent, '', 'Iterations:',
    numIterations, 'Last fitness improvement:', lastFitnessImprovement, 'Max Fitness:',
     maxFitness, '', 'Time Elapsed:', executionTime};
annotation(figure2,'textbox',[0.82 0.15 0.17 0.70],'String', solutionDesc, 'FitBoxToText
    ','off');

end
```

F.1.2 /figures/create_TrussDiagram.m

```
%% create_TrussDiagram
% Written by: Gavin Reynolds, 2008-2009
function create_TrussDiagram(fittestSolution, options)
[Edof, Ex, Ey]= getCALFEMtruss_geometry();

% Create figure
figure3 = figure(3); clf, axis('equal'), hold on, axis off

sectionDescs = cell2mat(fittestSolution(:,5));
sectionNums = cell2mat(fittestSolution(:,6));

% get weight
[a,Q,N,O,W]=solver_CALFEMtruss(fittestSolution, 0, options);

totalWeight = [num2str(round(sum(W))) ' kg'];

uniqueSectionNums = unique(sectionNums);
numSectionsUsed = numel(uniqueSectionNums);

for i=1:numSectionsUsed

    j = uniqueSectionNums(i);

    y1 = 0.15 + 0.1*(j-1)*2;
    y2 = y1 + 0.1;

    switch j
        case 1
            Colour = [1 0 0];
        case 2
            Colour = [0 1 0];
        case 3
            Colour = [0 0 1];
        case 4
            Colour = [0 0 0];
    end;

    % Create line
    annotation(figure3,'line',[y1 y2],[0.95 0.95],'Color',Colour);

end;


% Create Title textboxes
annotation(figure3,'textbox',[0 0.9 1.0 0.11],...
    'String',{'Fittest Design'},...
    'HorizontalAlignment','center',...
    'FontSize',12,...
    'FitBoxToText','off',...
    'LineStyle','none');

annotation(figure3,'textbox',[0 0.01 1.0 0.11],...
    'String',{'Total Weight:', totalWeight},...
    'HorizontalAlignment','center',...
    'FontSize',10,...
    'FitBoxToText','off',...
    'LineStyle','none');


for i=1:numSectionsUsed

    j = uniqueSectionNums(i);

    left = 0.135 + 0.1*(j-1)*2;

    sectionDescRow = find(sectionNums ==j, 1, 'first');
    sectionDesc = sectionDescs(sectionDescRow,:);

    % Create textbox
    annotation(figure3,'textbox',[left 0.88 0.13 0.07],...
        'String',{sectionDesc},...
        'HorizontalAlignment','center',...
        'FitBoxToText','off',...
        'LineStyle','none');
```

```
end;


elementDiameters = cell2mat(fittestSolution(:,4));
%Find the minimum element diameter
minElementDiameter = min(elementDiameters);

numElements=10;

x=Ex';
y=Ey';

for i=1:numElements
    %based the line width of the element
    lineWidth = elementDiameters(i)/minElementDiameter;
    elementNumber = cell2mat(fittestSolution(i,6));
    xe = x(:,i); ye = y(:,i);
    switch elementNumber
        case 1
            plotStyle = '-ro';
        case 2
            plotStyle = '-go';
        case 3
            plotStyle = '-bo';
        case 4
            plotStyle = '-ko';
    end;

    plot(xe,ye,plotStyle,'LineWidth',lineWidth)
end
```

F.1.3 /figures/create_BeamDiagram.m

```
%% create_BeamDiagram
% Written by: Gavin Reynolds, 2008-2009
function create_BeamDiagram(fittestSolution, options)

b = fittestSolution(1,1);
h = fittestSolution(1,2);

[a, Q, es, W]=solver_CALFEMbeam(b,h,0, options);

totalWeight = [num2str(round(W)) ' kg'];

%es is cell array of individual element [N V M];
elementNVMArray = cell2mat(es);
elementMArray = elementNVMArray(:,3);
maxM = max(abs(max(elementMArray)));

[rebarOK, numBars, barDiameter, maxMomentCapacityFitness, d,  barHorizontalSpacing,
     linkSize, coverToLink] = getCALFEMbeam_reinforcement(b,h,maxM);

figure3 = figure(3); clf, axis('equal'), hold on, axis off

% Create Title textbox
sectionDesc = ['Fittest Design: Section = ', num2str(b), 'x', num2str(h), 'm , Rebar = '
     , num2str(numBars),'T', num2str(barDiameter), ', Effective Depth = ', num2str(d), '
     m, Rebar c/c Spacing = ', num2str(floor(barHorizontalSpacing)), 'mm'];
annotation(figure3,'textbox',[0 0.9 1.0 0.11],...
    'String',{sectionDesc},...
    'HorizontalAlignment','center',...
    'FitBoxToText','off',...
    'LineStyle','none');

annotation(figure3,'textbox',[0 0.01 1.0 0.11],...
    'String',{'Weight per m:' totalWeight},...
    'HorizontalAlignment','center',...
    'FontSize',10,...
    'FitBoxToText','off',...
    'LineStyle','none');


rectangle('Position',[0,0,b,h],'LineWidth',2)

% there seriously isn't a circle plot function... using one from Mathworks File Exchange
%http://www.mathworks.com/matlabcentral/fileexchange/2876
if (numBars == 1)
    drawCircle([b/2,h-d],barDiameter/1000/2,1000);
else
    for i=1:numBars
        rebarHorizontalCenter = (coverToLink + linkSize + barHorizontalSpacing*(i-1) +
             barDiameter/2)/1000; %m
        drawCircle([rebarHorizontalCenter,h-d],barDiameter/1000/2,1000);
    end
end

end
```

F.2 Search Space Mapping

F.2.1 /test/fitnessMap.m

```
%% fitnessMap: A wrapper for the fitness functions, to generate fitness landscape plots.
% Also times how long it takes to evaluate the fitness of every possible combination.
% Written by: Gavin Reynolds, 2008-2009
function [fitnessArray, t] = fitnessMap(problemType)

tic;

%% Set fitness function based on problem type
switch problemType
    case 'Ten Bar Truss'
        fitnessFunction = @fitness_Truss;
        bitstringLength = 20;
        options = [];
    case 'Simply Supported Beam'
        fitnessFunction = @fitness_Beam;
        bitstringLength = 16;
        options = [];
    otherwise
        disp('Unknown problem type')
        return
end

%% Calculate fitness for generated individuals
numIndividuals = 2^bitstringLength;
fitnessArray = zeros(numIndividuals,1, 'double');

for i = 0:numIndividuals
    disp('============================')
    disp('Individual:')
    disp(i)
    disp('============================')
    %individual = dec2binvec(i, bitstringLength);

    binaryString = dec2bin(i,bitstringLength);
    %Convention, LSB is LHS, MSB is RHS.
    individual = logical(str2num(['fliplr(binaryString);blanks(length(binaryString))]')')
        ;

    fitnessArray(i+1) = fitnessFunction(individual, 0, options);
end

%% Save the results
t = toc;
assignin('base', 'fitnessTime', t);

assignin('base', 'fitnessArray', fitnessArray);
dlmwrite ([problemType '_fitness_map.csv'], fitnessArray);

sortedFitnessArray = sort(fitnessArray);
assignin('base', 'sortedFitnessArray',sortedFitnessArray);
```