THE UNIVERSITY OF QUEENSLAND

Bachelor of Engineering Thesis

Development of a New Bio-Inspired Optimisation Algorithm

Student Name: Timothy CASSELL

Course Code: MECH4500

Supervisor: Dr. Michael Heitzmann

Submission date: 28 October 2016

A thesis submitted in partial fulfilment of the requirements of the Bachelor of Engineering degree in Mechanical Engineering

UQ Engineering

Faculty of Engineering, Architecture and Information Technology

ACKNOWLEDGEMENTS

Firstly, I would like to thank my supervisor Dr Michael Heitzmann for giving me the opportunity to undertake this project. It has some of the most stimulating work I have done in my time at university and I am very grateful for his support and guidance.

I would also like to thank my family, and in particular my parents, for their constant love and support. They have always been understanding if I ever needed to prioritise university over other things.

Additionally, I would like to thank my friends. They always managed to make the long days and the late nights much more bearable. In particular, I would like to thank Emma for the idea of sugar gliders, which she came up with at about 3 o'clock one morning in early June.

Lastly, I am grateful to all those whom I have not mentioned, but have somehow helped me in the completion of this thesis.

ABSTRACT

Bio-Inspired Algorithms (BIAs) are a class of metaheuristic that have proven to be effective at optimising a vast range of complex, black box function types. A new BIA is proposed that is based on a small, nocturnal gliding possum; the native Australian sugar glider (*Petaurus breviceps*). Sugar Glider Algorithm (SGA) imitates the leadership hierarchy and foraging behaviour of a colony of gliders. Two co-dominant males lead a colony of five to seven gliders that forage for food by gliding between trees in search for insects or tree sap. The algorithm employs concurrent local exploitation (performed by the codominant males) and global exploration (performed by the remaining gliders). The performance of SGA has been quantitatively evaluated using five mathematical test functions, which are a mix of both unimodal and multimodal domains. The results are compared against Particle Swarm Optimisation, Differential Evolution and an Evolutionary Algorithm, with SGA performance amongst the best observed. Furthermore, SGA has been tested on three constrained engineering problems; coil spring design, welded beam design and pressure vessel design. SGA exhibited strong performance against seven existing algorithms, and found multiple new minimums than previously reported in literature. The results show that SGA is competitive against a wide range of existing algorithms in a variety of search domain topologies. These findings indicate that SGA is at the forefront of BIA performance and prove it is a superior candidate for the optimisation of engineering design problems.

CONTENTS

CHAPTER 1 – Introduction 1

1.1 – Introduction to Bio-Inspired Algorithms 1

1.2 – Motivation for Research 2

1.3 – Report Objectives and Scope 3

1.4 – Thesis Structure 4

CHAPTER 2 – Optimisation in Engineering 5

2.1 – Algorithm Terminology 5

2.2 – Engineering Optimisation Problems 7

CHAPTER 3 – Literature Review 11

3.1 – Algorithm Classification 11

3.2 – Evolutionary Algorithms 12

3.3 – Swarm Intelligence Algorithms 16

CHAPTER 4 – Algorithm Development Process 25

4.1 – Algorithm Goals 25

4.2 – Investigated Inspiration Sources 25

4.3 – Development Process 29

CHAPTER 5 – Sugar Glider Algorithm 33

5.1 – Algorithm Description 33

5.2 – Parameter Tuning 37

5.3 – Comparison to Existing Algorithms 43

5.4 – Python Code and User Recommendations 44

CHAPTER 6 – Performance Benchmarking 47

6.1 – Classical Mathematical Functions 47

6.2 – Constrained Engineering Design Problems 52

CHAPTER 7 – Case Study: Gearbox Design 62

7.1 – Introduction to the Case Study 62
7.2 – Methodology 63
7.3 – Results 68
7.4 – Case Study Outcomes 70
CHAPTER 8 – Further Work 71
8.1 – Implementation of SGA in ANSYS 71
8.2 – Algorithmic Improvements 71
8.3 – Gearbox Design Solver Improvements 73
8.4 – Implementation of SGA in Other Languages 74
Conclusion 75
References 76
Appendices 79

LIST OF TABLES

Table 1 – Structural optimisation problem description 8
Table 2 – Process optimisation problem description 9
Table 3 – Comparison of ABC to GA, PSO and DE (Karaboga & Akay, 2009) 20
Table 4 – Algorithm inspiration selection matrix 29
Table 5 – Tuning functions for the sight distance 37
Table 6 – Convergence power and colony size tuning functions 38
Table 7 – SGA Python input parameters 45
Table 8 – Mathematical benchmarking function definitions 47
Table 9- Algorithm parameter values 49
Table 10 – Performance benchmarking test settings 49
Table 11 – Mathematical benchmarking results (Krink et al., 2004) 50
Table 12 – Calculated feasibility percentages 52
Table 13 – Total function evaluations used for solving the design problems 56
Table 14 – Design problem test settings 57
Table 15 – Best minimum value results for the coil spring design problem 57
Table 16 – Statistical analysis of the results for the coil spring design problem 58
Table 17 – Best minimum value results for the welded beam design problem 58

Table 18 – Statistical analysis of the results for the welded beam design problem 59
Table 19 – Best minimum value results for the pressure vessel design problem 59
Table 20 – Statistical analysis of the results for the pressure vessel design problem 60
Table 21 – Cumulative rank results 60
Table 22 – Design parameter definitions 64
Table 23 – Overload factor K_o selection matrix 65
Table 24 – Mounting factor K_m selection matrix 65
Table 25 – Gearbox design equations 66
Table 26 – Gearbox design parameters (MECH3100 Group 9, 2015) 68
Table 27 – SGA parameters for gearbox optimisation 68
Table 28 – Gearbox optimisation results 69
Table 29 – Optimised gearbox design layout 69

LIST OF FIGURES

Figure 1 – Topology optimisation example (GS Engineering, 2014) 8
Figure 2 – Tool path optimisation example 9
Figure 3 – Typical 2D scramjet combustor (Lewis, 2012) 10
Figure 4 – Algorithm classification tree 11
Figure 5 – Example of genetic crossover 13
Figure 6 - Optimal path finding by ants 18
Figure 7 – Firefly Algorithm optimisation process (Yang, 2009) 22
Figure 8 – Grey Wolf Optimizer optimisation process (Mirjalili et al., 2014) 23
Figure 9 – Dolphin Echolocation Optimisation process (Kaveh & Farhoudi, 2013) 24
Figure 10 – Erratic search histories 31
Figure 11 – Delta functions (left) and Correct convergence behaviour (right) 32
Figure 12 – Sugar Glider Algorithm pseudocode 33
Figure 13 – Weightings for a CP value of 5 (the default) 35
Figure 14 – Effect of the CP parameter on the weightings 36
Figure 15 – Convergence behaviour for CP = 10 (a) and CP = 1 (b) 36
Figure 16 – Outcome f(x) values for differing SD 39
Figure 17 – Average fitness gain for differing SD 39
Figure 18 – Outcome f(x) for differing CP 40
Figure 19 – Outcome standard deviation for differing CP 41
Figure 20 – Outcome f(x) for differing colony size 42

Figure 21 – Outcome standard deviation for differing colony size 42
Figure 22 – Example implementation of SGA in Python 45
Figure 23 – Visual 2D representation of the benchmarking functions 48
Figure 24 – Optimum convergence behaviour (Krink et al., 2004) 51
Figure 25 – Physical coil spring dimensions 53
Figure 26 – Physical welded beam dimensions 54
Figure 27 – Physical pressure vessel dimensions 55
Figure 28 – Convergence behaviour analysis 61
Figure 29 – Previously designed heliostat (MECH3100 Group 9, 2015) 62
Figure 30 – Gearbox solver Python function concept... 67
Figure 31 – Attempted simulated breeding through mixing 72
Figure 32 – Attempted breeding through constrained random value selection 73

CHAPTER 1

INTRODUCTION

1.1 – INTRODUCTION TO BIO-INSPIRED ALGORITHMS

Optimisation problems are prevalent in all facets of engineering. A general optimisation task involves minimising/maximising an objective function via alteration of variable values, whilst accounting for variable constraints. Mathematical optimisation, through use of calculus, is often not a viable option for engineering problems. This is because the objective function often doesn't take a derivable algebraic form. It can be presented in a complex mathematical form, via computational simulations, or even in terms of measurements obtained from real objects. For example, the objective of a car exterior design may be to minimise drag, where the drag is calculated via implementation of the model in a CFD program. In such cases, the only information known is the variable values and the resulting 'fitness' of the solution (how minimal the drag is). Thus, methods that utilise only this information are required.

Methods for finding optimal solutions to problems where only the input/output information is known are classed as metaheuristic search methods. Metaheuristics are algorithms that are problem-independent, employ stochastic (random) methods, and make no assumptions about the space being searched. Thus, they are widely applicable to a large range of optimisation problems. Metaheuristics, however, do not guarantee that the global optimum will be found. Example metaheuristics include Simulated Annealing (Kirkpatrick, Gelatt Jr, & Vecchi, 1983) and the human-memory inspired Tabu Search (Glover & McMillan, 1986).

Bio-Inspired Algorithms (BIAs) are a class of metaheuristic algorithm that utilise methods inspired by biological processes to solve optimisation problems. BIAs gained popularity as research topics due to their combination of fascinating inspiration sources and promising performance outcomes. There are now algorithms inspired by a vast range of biological sources such as genetics, pack hunting of wolves, social behaviour of bees, and bird flocking. BIAs have been shown to be able offer a mix of both good performance and search space adaptability, meaning they can generally be effective at solving a broader range of problems. The adaptability is inherent in the design due to the way in which the biological organisms from which they are derived are able to adapt to their environment. For these reasons, BIAs are a promising area of research within the field of metaheuristics.

1.2 – MOTIVATION FOR RESEARCH

1.2.1 – Reasoning Behind Choosing Bio-Inspired Algorithms

Bio-inspired algorithms are of particular research interest due to their inherent robustness and efficiency. Over the course of millions of years, biological processes have themselves been able to 'evolve' such that they are continuously becoming more effective (whether the process is the human immune system or the pack hunting technique of wolves). This implies that nature is an abundant resource for processes that are operating in optimal ways. Therefore, algorithms based on these processes are often very effective at the optimisation of arbitrary objective functions. A particular strength of BIAs is that they are able to escape local optima effectively due to the 'judgement' exercised in nature, where a good solution is not always accepted due to the desire for a better one. An example being the pollination of flowers by bees; a bee may have an abundance of suitable flowers available, but may discard many options with a preference for searching of the best one available. This is one of the reasons that BIAs are regarded as powerful optimisation tools.

1.2.2 – The No Free Lunch Theorem

The No Free Lunch Theorem provides both a motivation and an inherent design guideline for developing algorithms. The No Free Lunch theorem states that the performance of all search algorithms is the same when averaged over all possible objective functions (Wolpert & Macready, 1997). That is, some algorithms perform exceedingly well with certain functions, but are inefficient with others. An implication of this theorem is that there is no single algorithm (existing or otherwise) that is the best for optimising all objective functions. Therefore, as the number of engineering applications that utilise optimisation increases, as does the demand for the creation of new algorithms. This is one of the main driving forces behind the continual development of new optimisation algorithms.

The No Free Lunch Theorem also implies the importance of designing the algorithm with a certain application area in mind. In accordance with the theorem, trying to design an algorithm that performs well in all areas is futile. Although metaheuristics are applicable to almost all optimisation tasks, performance is not guaranteed to be acceptable. Therefore, some prior knowledge about the search spaces of particular interest is beneficial when designing the algorithm processes. When an engineer is selecting an algorithm for their particular application, they do not look at algorithms with a broad-range of good performance. Rather, they seek the algorithm with best performance in their application. Thus, the theorem implies it makes most sense to take an application-based approach to designing the algorithm.

1.3 – Report Objectives and Scope

1.3.1 – Project Aim

"To produce a novel bio-inspired algorithm that is efficient and effective at solving a vast range of engineering optimisation problems, particularly those within the field of mechanical design"

1.3.2 – Project Objectives

The project objectives are the metrics that will be used to determine if the project was successful in achieving its intended purpose. The project objectives are to:

- produce a truly-novel bio-inspired algorithm,
- ensure that the algorithm performs competitively against existing BIAs on problems related to engineering design,
- ensure that the algorithm is easily implementable by engineers unfamiliar with metaheuristics, and
- in conjunction with work performed by Bryce (2015) , enhance the experience of FEA program users, removing the user-dependency of the current inbuilt optimisation methods.

1.3.3 – Project Scope

The following items were within the scope of the project:

- A comprehensive literature review that identifies existing BIAs and their optimisation mechanics,
- Development of new bio-inspired algorithm based on either:
  - animal/insect hunting, mating or social behaviours,
  - genetics, DNA/RNA, Proteins, Immune System, or
  - dynamics of cellular-level biological processes.
- A performance comparison of the algorithm through:
  - use of mathematical benchmark test functions, or
  - use of classical constrained engineering design problems.
- The undertaking of a case study that further highlights the value of the proposed algorithm.

The following items were not within the scope of the project:

- Investigation of algorithms based on other natural phenomena such as physics or chemistry (beyond verifying that the proposed algorithm is dissimilar in order to avoid plagiarism), or
- Performance testing in fields unrelated to engineering design where significant alteration of the algorithm would be required.

1.4 – Thesis Structure

In order for the proceeding information on optimisation to be well understood, Chapter 2 gives an introduction to the topic. Algorithm terminology is presented, along with typical applications of optimisation within engineering. Chapter 3 is a comprehensive literature review that analyses the existing bio-inspired algorithms. Then, the algorithm development process is outlined in Chapter 4, including the design goals of the final algorithm, investigated sources of inspiration and the early design iterations. Next, Chapter 5 provides the intricate details of Sugar Glider Algorithm including parameter tuning and selection, and a guide to using SGA in Python. Chapters 6 provides the results of benchmarking the algorithm against existing BIAs, on both mathematical functions and constrained engineering design problems. Chapter 7 details a case study that was performed to highlight the strength of SGA when applied to engineering problems (through design of a spur gearbox). Finally, Chapter 8 provides the intended future work to be performed with relation to SGA.

CHAPTER 2

OPTIMISATION IN ENGINEERING

2.1 – ALGORITHM TERMINOLOGY

There is a range of terms associated with optimisation that characterise both the type of problems being solved, and the methods used to do so. A working knowledge of these terms is thus required in order for the proceeding work to be understood.

2.1.1 – Unconstrained and Constrained Optimisation

Unconstrained optimisation occurs when the whole real-numbered domain is available to search. Furthermore, there are no supplementary constraint equations that must be satisfied whilst the optimisation process is occurring. It is a simpler case to solve, but the results may not be feasible if variables have not been appropriately constrained.

Engineering design problems are almost exclusively of the constrained type. This ensures that the result of the optimisation process is a feasible solution to the problem. Constraints are introduced in two ways, including:

- applying limits to the range of numbers that variable values are able to be selected from, and
- introducing inequality constraint equations that must be satisfied in order for a solution to be considered feasible.

Constraint equations facilitate the assurance that a result output from the optimisation process is able to be implemented appropriately. For example, in the design of an ultra-light aerofoil, a constraint may be that the total aerodynamic lift supplied be greater than a set value. Implementation of constraints in computational optimisation techniques is often not a difficult task, as the objective function is modified such that penalties are applied if constraints are violated.

2.1.2 – Discrete, Continuous and Mixed Variable Optimisation

Discrete optimisation occurs when the variables in a problem can only have values belonging to a particular set. Combinatorial optimisation is a subset belonging to discrete optimisation, which are common amongst planning problems. Combinatorial problems include the Travelling

Salesman Problem, where the objective is to select a combination of routes between cities that minimises the total travel time. The total number of possible combinations of routes between cities is a (large) discrete set, and thus optimisation routines applied to the problem must account for the discrete nature.

On the other hand, parameters of continuous optimisation problems can take any value within a set range. Many design optimisation problems are setup such that they can be solved using continuous techniques, particularly in shape, size and topology optimisation.

Mixed variable optimisation is a commonly occurring case in structural design problems. In mixed variable optimisation, the set of variables is a mix of both continuous and discrete. An example is a problem involving SHS beams, where the cross-section is to be a standard design. The cross-sectional dimensions would be a discrete variable, but others such as the length will be continuous.

2.1.3 – Single and Multiple Objective Problems

Single objective optimisation is the case where the objective function is singular in its dependant variable. An example would be optimising the shape of a structural component with the objective of minimising the weight. In this case, the material would be set prior to the optimisation process being carried out, such that the cost would be a simple function of the material volume.

On the contrary, multiple objective optimisation scenarios occur when the objective function is non-singular in its dependant variable (so the problem effectively has multiple objective functions, all to be optimised simultaneously). Building on the example above, the problem would become a multiple objective problem if the material was also considered variable. The objective may then be to minimise both the weight (a function of component volume and material density) and the cost (a function of component volume, material cost, and shape). Here, there is no clear relation between the two objectives to be minimised, and so the problem is classed as multiple objective. Multiple objective problems require special solvers that are able to handle the concurrent optimisation of more than the one objective function.

2.1.4 – Stochastic and Deterministic Algorithms

A stochastic algorithm is one which utilises randomness in order to search the domain. All bio-inspired algorithms are stochastic in design, with random operators often at the heart of their exploration routines. An implication of the random operators in stochastic algorithms is that they will never perform exactly the same over multiple runs, even when initialised in the same

configuration. The challenge is to still produce the same end result (the global optimum) whilst accounting for the different iteration patterns.

Deterministic algorithms are the logical opposite of stochastic. If initialised in a certain configuration, the algorithm will always iterate in the same pattern and produce the same end result. Deterministic algorithms are often ineffective unless the search space topology is well-defined and the user has a good idea of the location of the optimum (Kress & Keller, 2007).

2.1.5 – Static and Adaptive Algorithms

A static algorithm is one whose parameter definitions do not change as the iterations progress. Most algorithms are, in their basic form, static in design. Algorithm tuning is often performed to determine optimal definitions for the parameters (which may be numerical values or functions) and the definitions then do not change once the algorithm is initialised.

Research is often undertaken on an algorithm to determine if changing these definitions through the iterations is effective. The algorithm would then be considered adaptive. A feedback loop is established between the algorithm outcomes and its parameter set, with a user-defined alteration routine altering the parameter definitions whilst the algorithm progresses. The hope is that the performance can be improved through implementation of adaptive techniques.

2.2 – Engineering Optimisation Problems

There are a wide range of optimisation problems that are encountered by engineers. These range from structural design optimisation and process optimisation, to other applications such as control and manufacturing or mathematical modelling. New applications of optimisation within engineering are also being formulated regularly. Additionally, as the "computerisation" of engineering processes further increases, the opportunities for computational optimisation of such processes also further increases.

2.2.1 – Structural Optimisation

Described in Table 1, structural optimisation is one of the most common engineering optimisation problem types and involves the alteration of a physical component in order to satisfy a design goal. The most common of these goals is to minimise the volume of material used in the part. Optimising component mass is an ever increasingly important concept as manufacturers of both aerial and road-going vehicles look to provide maximum fuel efficiency, for performance, cost and environmental reasons. Furthermore, a lower mass of given material often implies a cost decrease also.

Table 1 – Structural optimisation problem description

Objective Function	Weight, cost, strength, drag coefficient
Variables	Dimensions
Constraints	Stress, deflection, manufacturability

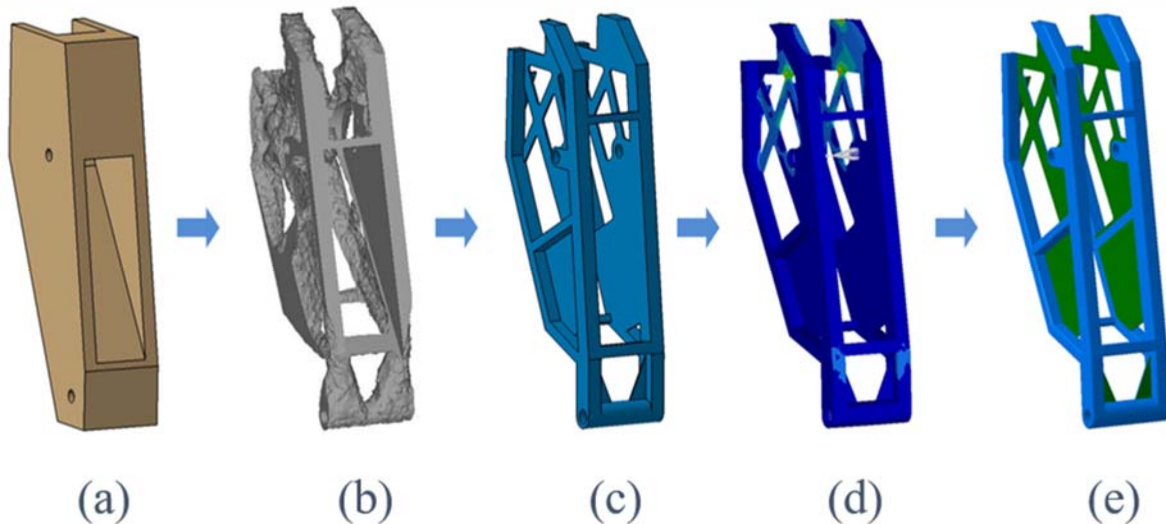


Figure 1 – Topology optimisation example with Allowable space claim (a), Topology optimisation output (b), Validation CAD model (c), Validation FEA model (d) and Final design (e) (GS Engineering, 2014)

Structural optimisation often involves the integration of a computer aided design (CAD) program to validate complex component shapes. Figure 1 gives an example of topology optimisation, a type of structural optimisation process. In this case, a CAD program assists to minimise the total material mass. Other examples of structural optimisation may not require the use of a CAD program, with their objective functions being mathematically formulated. This is applicable to structures with simple shapes, such as pressure vessels and springs.

2.2.2 – Process Optimisation

Described in Table 2, process optimisation within the context of engineering design is often associated with the manufacturing processes of components. For example, optimising the tool path of a CNC milling process will reduce the time required to produce the part. The optimisation task may account for CNC head speeds, tool rotational speeds and the order and direction of cutting passes. In this case, a computer aided manufacturing (CAM) program would be used to simulate the tool paths. If a part is to be mass produced, optimising the tool path will result in a lower manufacturing time that increases process efficiency.

Table 2 – Process optimisation problem description

Objective Function	Time or cost
Variables	Timings, process parameters or order of operations
Constraints	Feasibility, tolerances, finish quality

Figure 2 gives an example of tool path optimisation. Figure 2 (a) is the intuitive path, which is the path that would likely be taken when minimal thought is given to the problem. However, this path is actually 11% longer than the optimised path on of Figure 2 (b). This application is actually an example of the well-known Travelling Salesman Problem.

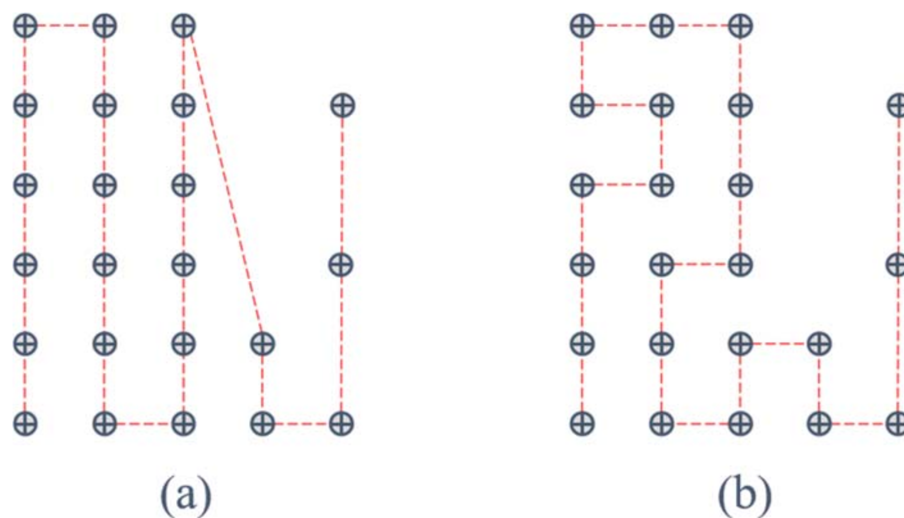


Figure 2 – Tool path optimisation example

2.2.3 – Other Applications

Curve fitting of experimental data is another application of optimisation within engineering. Experimentally measured points are able to be approximated with mathematical functions, with the optimisation process minimising the total error of the approximation. Applications of this method include UV spectroscopy, X-ray analysis, IR spectroscopy and chromatographic techniques, with bio-inspired algorithms being a popular choice of optimisation method (Polo-Corpa et al., 2009).

The identification of optimal control parameters is another application of optimisation within engineering. As a result of the emerging Internet of Things phenomenon, there is an ever

increasing amount of data flowing off machines. For example, GE Oil and Gas has predicted that if all of the data generated by industry turbomachinery, pipelines and artificial lift equipment was harnessed correctly, production can be improved by up to 8% (GE Oil and Gas, 2016). As the data becomes more accessible through the Internet of Things, identification of optimal control parameters will be another application of bio-inspired algorithms.

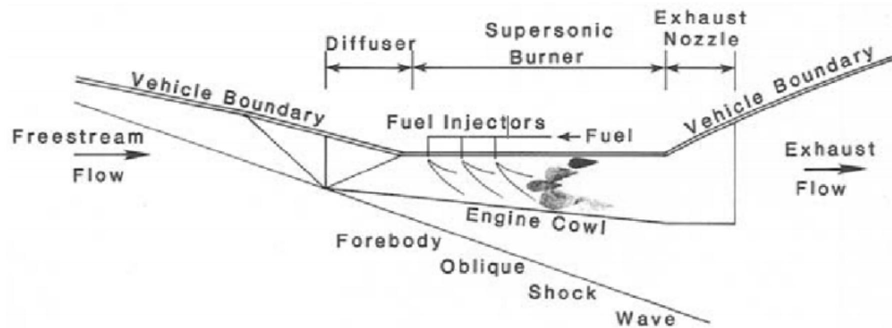


Figure 3 – Typical 2D scramjet combustor (Lewis, 2012)

Furthermore, work performed at The University of Queensland by Lewis (2012) used a bio-inspired algorithm to optimise the inlet of a scramjet combustor, as shown in Figure 3. The goal was to minimise the total pressure loss for an inviscid, two-dimensional, three ramp scramjet inlet and combustor. The process involved linking the optimisation code to a Barrine CPU cluster that performed computational fluid dynamics (CFD) simulations. The algorithm proved effective in optimising the geometry such that the design goal was achieved.

CHAPTER 3

LITERATURE REVIEW

3.1 – ALGORITHM CLASSIFICATION

Within the space of numerical optimisation lies a class of algorithms inspired by nature. This class can further be divided into those inspired by chemistry, physics or biology. Bio-inspired algorithms can then be separated into two sub-classes. The Evolution sub-class contains algorithms primarily inspired by both genetic operations and processes, and the Darwinian theory of Survival of the Fittest. The Swarm Intelligence sub-class is based on the hunting, movement and reproduction processes of living organisms. Figure 4 gives some examples of existing bio-inspired algorithms. The algorithms vary in many aspects, from their inspiration source, to the number of control parameters to their method complexity.

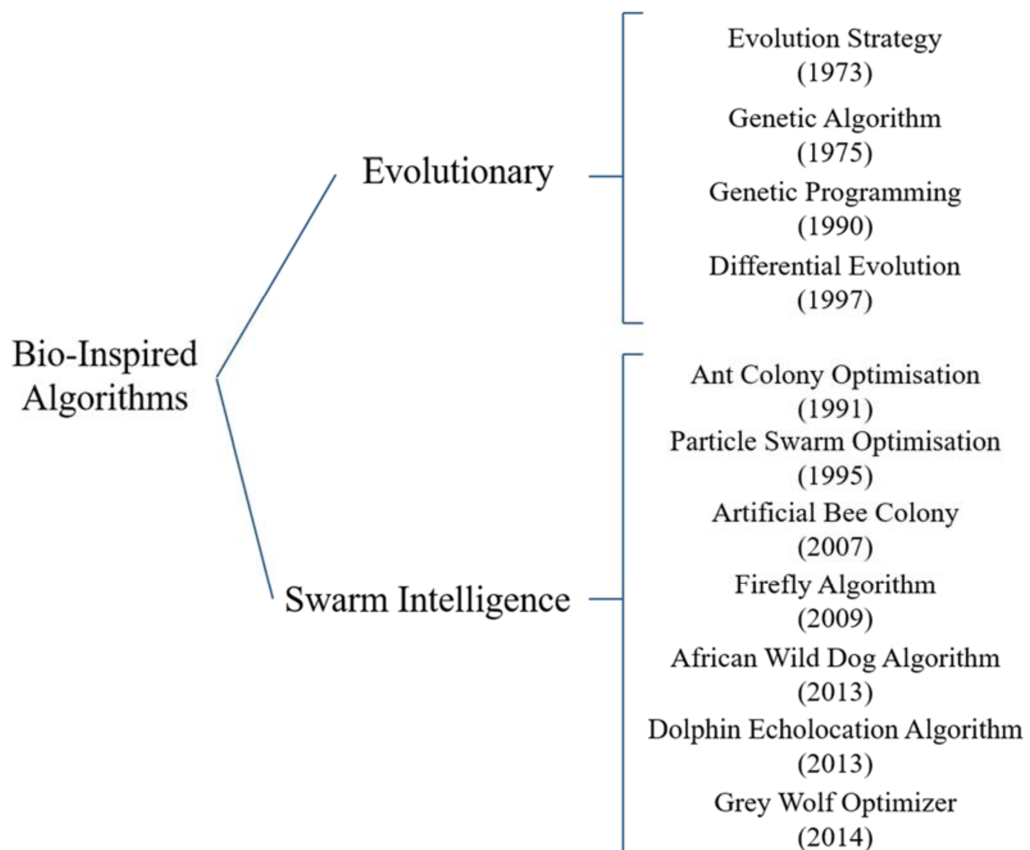


Figure 4 – Algorithm classification tree

3.2 – Evolutionary Algorithms

Existing evolutionary algorithms have been presented, along with their domain suitability and the number of control parameters (excluding population size and the number of iterations, as all algorithms require these selections).

3.2.1 – Genetic Algorithm (GA)

Domain: Continuous/Discrete **Control Parameters:** 3

GA is perhaps the most well-known of the Evolutionary Algorithms. Proposed by Holland (1975), GA takes inspiration from the micro-level biological processes that drive evolution, in accordance with the Charles Darwin Theory of Survival of the Fittest. Solutions are likened to chromosomes, with each having a number of genes representing the variables. The chromosomes have genetic operations performed upon them to produce new generations. The idea is that each generation will have generally better fitness values than the previous, meaning that the population will eventually reduce in genetic diversity and converge towards an optimal value. The process for traditional GA is:

1. Generate initial population of solutions
2. Evaluate initial population fitness values
3. Whilst there is sufficient diversity amongst population (loop):
    i. Select parents and perform crossover to produce new generation
    ii. Perform mutation to small percentage of population
    iii. Evaluate population fitness values
    iv. Repeat loop

The genes (solution components) are typically encoded in binary, so as to allow for genetic operations to be performed. The genetic operations were traditionally limited to selection, crossover and mutation; however recent variants of GA have also explored the use of regrouping, colonization-extinction and migration with promising results (Akbari, 2010). Crossover is the method through which new generations of solutions are generated and is analogous to the biological process from which it takes its name. An example of crossover can be seen in Figure 5.

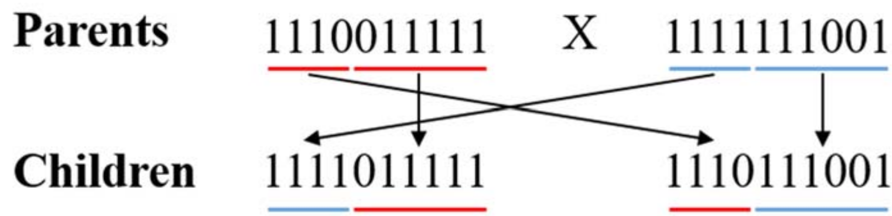


Figure 5 – Example of genetic crossover

The crossover point/s are chosen at random and, depending on the parents, may result in children that are identical (particularly in later generations). Converting from binary to decimal, the two parents have parameter values of 927 and 1017. The resulting children have values of 991 and 953 respectively. These numbers may represent the variable values in a one-dimensional function, and their fitness may be the function value. When averaged over the whole group of solutions, the fitness of the children will generally be higher than that of the parents due to biased selection. As the solutions converge, genetic diversity between chromosomes decreases, meaning the generations become more alike. The algorithm typically terminates when the deviation across the population is smaller than a tolerance.

The solutions (chromosomes) are ranked based on their fitness values (determined by the objective function). The higher the ranking, the greater the chance of being selected for reproduction via crossover. Zitzler, Deb, and Thiele (1999) note that it is important to not limit the parents to only the best-performing solutions or else premature convergence will occur. Lower fitness parents aid in the explorative characteristics of the algorithm, increasing the chances of escaping local optima.

As a modification to the traditional procedure, Elitist Genetic Algorithm variants allow for the best few solutions (or best single solution) to carry over to the next generation without alteration, guaranteeing best-solution preservation throughout the iterations (Baluja & Caruana, 1995). This modification is sometimes beneficial, but may cause the population to get stuck in local optima, depending on the search space topology. Zitzler et al. (1999) have claimed that adding elitism to GA improved the efficiency of the algorithm, which has been reaffirmed by Rudolph (1999) and Deb (2002).

Binitha and Siva Sathya (2012) state that GA may have a tendency to converge towards local optima rather than the global optimum if the fitness function is not defined properly. Another disadvantage of GA is that it is not directly accommodating of constrained optimisation (where variables must stay within certain ranges). To handle constraints, penalty functions must be used that assign very unfit values to solutions where a variable is out of the acceptable range.

3.2.2 – Differential Evolution (DE)

Domain: Continuous/Discrete **Control Parameters:** up to 6

Differential Evolution is an evolutionary algorithm proposed by Storn and Price (1997). It is similar to Genetic Algorithm in that it takes inspiration from the genetic operators of crossover and mutation. However, the generation of new solutions takes a slightly different method, with new solutions being a combination of three others, rather than two. Differential Evolution also differs from GA in that the newly generated solutions are only accepted if they are of a higher fitness. Although this guarantees best-solution preservation, it may also limit the explorative characteristics of the algorithm in multimodal domains. The process of DE is:

1. Set parameters of $CR \epsilon [0, 1]$, $F \epsilon [0, 2]$
2. Generate initial population of solutions called agents ($NP \geq 4$)
3. Evaluate initial population fitness values
4. Until a termination criterion is met (loop):

   For each agent x in the population:

   i. Select 3 other (random) agents a, b, c
   ii. Select random index $R \epsilon [1, \dots, n]$ where n is the dimension of the agent vector
   iii. For each dimension i, pick a random number $r_i = (0, 1)$
   iv. For each dimension i, if $r_i < CR$ or $i = R$ then set $y_i = a_i + F * (b_i - c_i)$, else set $y_i = x_i$
   v. If the fitness of the new solution y is better than the fitness of the old solution x, replace the agent with the improved candidate solution
   vi. Repeat loop

As well as three algorithm-defining parameters, DE has another three tuneable parameters that must be selected by the user. Increasing the population size, NP, increases the explorative capability of the algorithm, but with added computational cost. The differential weight, F, and the crossover probability, CR, also increase the explorative capability by increasing the mutation magnitude and probability respectively. However, this also increases the time taken for the algorithm to converge to a final solution. Zielinski and Laur (2006) have proposed an adaptive DE algorithm that changes the F and CR parameters as the iterations progress. Results show that adaptive control of the parameters increases the performance of the algorithm as opposed to tuned, fixed values. The average number of function evaluations is higher for the adaptive approach; however, the authors note that tuning the parameters also requires preliminary computational effort.

3.2.3 – Evolution Strategy (ES)

Domain: Continuous/Discrete **Control Parameters:** 3

Evolution Strategy was developed in its basic form at the Technical University of Berlin by Rechenberg (1973). ES is actually a group of closely-related algorithms, each differing slightly in recombination technique, but all with the same general process. Similar to DE, ES uses the micro-level process of random recombination to from new generations of individuals, with the highest individuals being selected to survive. GA on the other hand takes inspiration from the macro-level Survival of the Fittest theory. The population fitness is first evaluated and the best individuals are selected to become the parents that produce the next generation. The process for general ES is:

1. Generate initial population of solutions
2. While not termination criterion (loop):
   i. Perform random crossover using all population to produce new generation (that is larger than the population beforehand)
   ii. Perform mutation through perturbation using random vector with a zero mean
   iii. Evaluate new generation fitness values
   iv. Select best individuals for survival using multivariate normal distribution
   v. Repeat loop

Differences in the number of parents used in each crossover and the number of offspring produced for the new generation give rise to the different branches of ES. Mutation operators are also varied to produce other ES variants. Yao and Liu (1997) have used Cauchy mutation operators to derive a Fast Evolutionary Strategy (FES) that was shown to consistently outperform ES in multi-modal test functions due to its ability to escape local optima. The authors state that ES wasn't able to escape local optima effectively due to a local-search like Gaussian mutation operator. Another, state-of-the-art, ES variant was proposed by Hansen and Ostermeier (1996) and adapts the covariance matrix of the normal distribution as iterations are performed. The result is that the relationships between variables are learned by the algorithm, with performance increases as a result.

3.2.4 – Genetic Programming (GP)

Domain: Discrete **Control Parameters:** Depends on algorithm used

Genetic Programming was initially proposed by Koza (1990) for use in the context of artificial intelligence. GP is used to create computer programs that are able to perform a singular task optimally. GP is actually an application of genetic algorithms, rather than a standalone

algorithm itself. GP utilises any of the previously mentioned evolutionary algorithms (typically GA, however not exclusively) to find the best computer program to perform the specified function. GP encodes the tree-like structured programs into solutions and represents them as a set of genes (so that one solution is like a chromosome in GA). The genes are then mutated according to the methods of the evolutionary algorithm employed. Evaluating the fitness of a solution involves running the prospective program and evaluating its performance (often its run time and outcome accuracy). GP doesn't seem to have found popularity in solving practical engineering optimisation problems. This is likely because it is relatively computationally intensive and limited in engineering applications which are able to be represented in the required tree-like structure.

3.3 – Swarm Intelligence Algorithms

Existing swarm intelligence algorithms have been presented, along with their domain suitability and the number of control parameters (excluding population size and the number of iterations, as all algorithms require these selections).

3.3.1 – Particle Swarm Optimisation (PSO)

Domain: Continuous/Discrete **Control Parameters:** 3

Particle Swarm Optimisation was first proposed by Kennedy and Eberhart (1995) and was initially intended to simulate the social behaviour of animal swarms, particularly fish schooling and bird flocking. The algorithm was the first to utilise swarm intelligence, in contrast to the previously proposed Evolutionary Algorithms. Each solution is equated to a particle within a swarm of other particles. The particles can be likened to any swarming/schooling creature, including insects, fish and birds. The algorithm process is based on the behaviour of these swarms when searching for a goal (be that a food source or habitable environment). The intelligence component of the algorithm refers to the nature in which particles 'communicate' their positions to each other, allowing other particles to use this information when making movement decisions. The process for traditional PSO is:

1. Generate initial population of solutions
2. Evaluate initial population fitness values
3. Whilst there is sufficient spread in swarm location (loop):
    i. Generate all particles' velocity
    ii. Generate new positions based on velocities
    iii. Evaluate swarm fitness values
    iv. Update individual and global best position values

v. Repeat loop

The velocity is dependent on two factors:

- The position of the particle relative to its own best known position (allowing sufficient search space exploration)
- The position of the particle relative to the swarm's best known position (causing eventual convergence)

Each of these positions has a weighting applied and the velocity and new position are calculated. As iterations take place and the global best-known value comes to be constant, the swarm will eventually converge.

Unlike the non-elitist Genetic Algorithm, PSO guarantees that the best solution is always carried over throughout iterations (as the particle with the global best solution has zero velocity). This inherent best-solution preservation is a strength of the PSO algorithm. Unlike GA, if PSO reaches a local optimum and has decided that it is not good enough to accept as the final solution, it is able to re-disperse the particles and continue searching (keeping the most-fit particle in place).

A popular modification to PSO is the implementation of multiple swarms. Information is only shared amongst each independent swarm for a majority of the iterations, with the outcome of finishing with multiple optima. Hendtlass (2005) cites this as being particularly useful for searching amongst multi-modal functions with good results obtained using an altered PSO that uses multiple waves of swarms. Another modification is that of introducing random perturbations of particle velocities, increasing the explorative capabilities of the algorithm. Lovbjerg and Krink (2002), Xie, Zhang, and Yang (2002) and Xinchao (2010) all report performance increases with different methods of velocity perturbation. The algorithm tends to explore the domain more exhaustively, leading to better solutions in difficult search spaces.

3.3.2 – Ant Colony Optimisation (ACO)

Domain: Discrete (combinatorial) **Control Parameters:** 4

Proposed by Dorigo, Maniezzo, and Colorni (1991), Ant Colony Optimisation is based on the foraging behaviour of ants. ACO is able to solve problems that can be reduced to finding good paths through nodes in space. ACO is based on the concept of stigmergy, a term accredited to Grasse (1959). Stigmergy refers to the phenomenon of indirect information sharing amongst a group via individuals modifying the local environment. Ants release pheromones as they travel between their nest and food sources, as shown in Figure 6. Therefore, the trails to the richer

sources of food have more pheromone laid down upon them. Ants tend to travel along paths that have a greater amount of pheromone, which results in indirect communication about locations of better food sources, as shown in Figure 6. Dorigo, Maniezzo and Colorni successfully applied the concept of stigmergy in ant colonies in order to provide biasing to the random operators used in the algorithm.

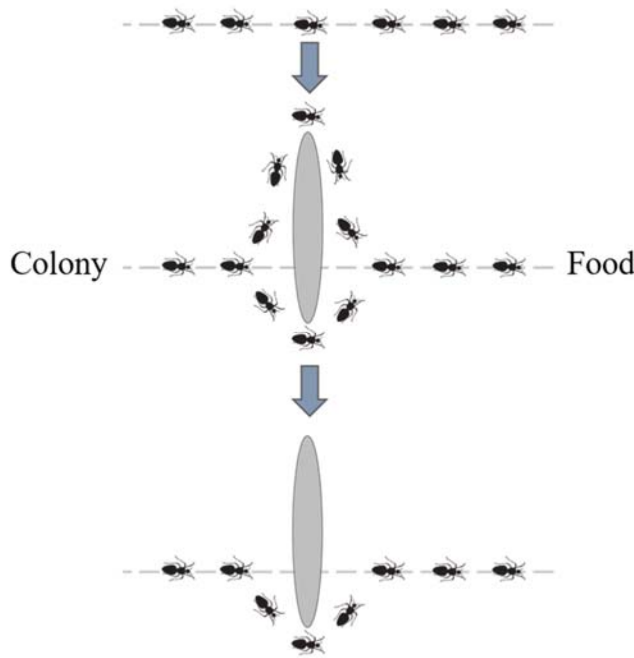


Figure 6 - Optimal path finding by ants

ACO is a solver limited to combinatorial optimisation problems (including path planning, job scheduling and assignment problems). Algorithms for solving discrete, combinatorial problems are not generally applicable to the field of mechanical design (especially not in the cases for which FEA is used). Shape, size and topology optimisation all have continuous variable domains, meaning that ACO (or a similar algorithm) would not be of use. However, the concept of stigmergy is stimulating and highlights the various ways in which nature provides inspiration for computational algorithms.

3.3.3 – Artificial Bee Colony (ABC)

Domain: Continuous/Discrete **Control Parameters:** 3

Proposed by Karaboga and Basturk (2007), Artificial Bee Colony Algorithm takes inspiration from the foraging behaviour of a honey bee swarm. The algorithm has four main components:

- Food sources (solutions) with a certain level of nectar (fitness value)
- Employed bees, tasked with finding new food sources in the local neighbourhood of their current best known food source

- Onlooker bees, which receive information provided by employed bees, before probabilistically selecting a food source and becoming employed themselves
- Scout bees, who were once employed bees whose food source could not be improved after a certain search time, choose another food source at random, increasing the algorithm's explorative characteristics (and preventing local optimum acceptance)

The algorithm is both simple in design and strongly interlinked with the inspiration behind it. The general process for ABC is:

1. Generate initial population of employed bees and their respective food sources (solutions)
2. Evaluate initial population nectar levels (fitness values)
3. Until termination criterion met (loop):
    i. Set employed bees to search for new food sources in their neighbourhood
    ii. Assign onlooker bees to a food source based on probabilistic decision
    iii. Send scout bees on random search
    iv. Memorise swarm single best position
    v. Repeat loop

Algorithms that only accept fitter solutions can have a tendency to become stuck in local optima of multimodal functions. ABC is clever in that it detects stagnation of a bee and randomises its position, ensuring that the algorithm is searching the domain for all of the iterations.

The user defined parameters for ABC are:

- The population size, which is equal to the number of food sources
- The limit for search attempts before an employed bee converts into a scout bee
- The termination criterion

ABC has been widely implemented in a number of applications. Research by Omkar et al. (2011) has applied a modified ABC for multi-modal objective functions through design optimisation of a composite structure. The performance of ABC was found to be at least on-par with that of GA and PSO. Another large performance comparison was performed by Karaboga and Akay (2009) using 50 multi-dimensional benchmark functions. Results showed that ABC performed markedly higher than GA and PSO, whilst the difference over DE is marginal. Table 3 gives a comparison of the performance of ABC to the other three algorithms.

Table 3 – Comparison of ABC to GA, PSO and DE (Karaboga & Akay, 2009)

Algorithm	GA	PSO	DE
ABC Performed Better	28	24	8
Equal Performance	20	22	37
ABC Performed Worse	2	4	5

3.3.4 – African Wild Dog Algorithm (AWDA)

Domain: Continuous/Discrete **Control Parameters:** 0

AWDA is based on the communal hunting behaviour of African Wild Dogs (sometimes called African Hunting Dogs). The algorithm was proposed by Subramanian et al (2013) and is an SI based meta-heuristic. The authors say the algorithm is simple to implement with minimal parameters to be user-specified, however this is perhaps more due to a lack of technical endeavour rather than innovative algorithm processes. The process for AWDA is:

1. Generate initial population of dogs (solutions)
2. Evaluate initial population fitness values
3. Whilst there is sufficient spread in dog pack location (loop):
    i. Update each individual dog position
    ii. Evaluate population fitness values
    iii. Repeat loop

Each dog is moved toward a random dog that has a higher fitness value. Updating position is a simple method subject to Euclidian distances between dogs, with the outcome being that the pack will always converge between iterations. This is in contrast to Grey Wolf Optimizer (Section 3.3.6) where wolves are able to also diverge, increasing the explorative characteristics of the algorithm. Due to the simple movement definition of AWDA it would likely not deal well with largely multimodal functions with many local optima. The authors have only verified the algorithm on one benchmark function: the Goldstein-Price function. Although the function converged to the global optimum, it took 1000 function evaluations (Subramanian et al., 2013). The SHERPA algorithm was able to converge to the optimum in 500 evaluations (Red Cedar Technology, 2014).

The mechanics of AWDA are simple and their relation to African Wild Dogs appears rudimentary. The position updating method is a general characteristic of all pack hunting animals – that the pack will move towards the best-performing members (the leaders, or the members closest to a prey). Beyond this mechanic, there seems to be no other relation to African

Wild Dogs, which are some of the best pack hunting animals on earth. They display hierarchical social structures and strong inter-pack breeding whereby all females find new packs once they reach maturity. None of these behavioural characteristics have been explored for use in the algorithm. Implementing some of these processes would likely lead to better performance. For example, the breeding technique could be combined with genetic operations to introduce greater solution variability (helping search space exploration).

3.3.5 – Firefly Algorithm (FA)

Domain: Continuous/Discrete **Control Parameters:** 2

A relatively recent addition to the swarm intelligence family, Firefly Algorithm is based on the flashing light behaviour of its namesake. Fireflies use this flashing mechanism for many reasons, including communication and for attracting both potential mates and prey. Yang (2009) simplified the behaviour of fireflies into three rules for implementation in the algorithm:

1. All fireflies are unisexual and attracted to all other fireflies.
2. The strength of attraction is both proportional to the brightness of, and inversely proportional to the distance between, two fireflies.
3. If there are no brighter fireflies than a given firefly, then it will move randomly.

Each firefly is a representation of a potential solution. The brightness value is a result of the fitness function, such that a higher fitness yields a higher brightness. The rules are simple in nature but give yield to a relatively intricate process of searching the solution space. Unlike other algorithms such as AWDA, the fireflies are never in a forced-convergence phase (such that agents must move towards a fitter agent). Although fireflies with greater intensity may exist, from the perspective of a given fly there may be no visible fitter solutions (due to distance decreasing the relative brightness's). If a firefly can see no better solutions it simply moves randomly, which increases the explorative characteristics of the algorithm. The general process for FA is shown in Figure 7.

Objective function $f(\mathbf{x})$, $\mathbf{x} = (x_1, ..., x_d)^T$
Generate initial population of fireflies $\mathbf{x}_i$ $(i = 1, 2, ..., n)$
Light intensity I_i *at* $\mathbf{x}_i$ *is determined by* $f(\mathbf{x}_i)$
Define light absorption coefficient γ
while *(t <MaxGeneration)*
for $i = 1 : n$ *all n fireflies*
  for $j = 1 : i$ *all n fireflies*
    if $(I_j > I_i)$, *Move firefly i towards j in d-dimension;* **end if**
    Attractiveness varies with distance r via $\exp[-\gamma r]$
    Evaluate new solutions and update light intensity
  end for *j*
end for *i*
Rank the fireflies and find the current best
end while
Postprocess results and visualization

Figure 7 – Firefly Algorithm optimisation process (Yang, 2009)

The three user-defined parameters of FA are:

- γ, determining how quickly the distance between fireflies reduces the relative intensity
- α, a parameter controlling the maximum step size
- *n*, the population size of the swarm, which is usually set between 15 and 40 (Yang, 2009)

FA has been shown to perform well over a wide variety of objective functions. Gandomi, Yang, and Alavi (2011) have had success in implementing a mixed variable form of FA, applied to civil-structural optimisation problems. Performance testing on the continuous-domain welded beam problem often found in literature showed that MV-FA converged to the global optimum with less function evaluations than both GA and DE (amongst other non-bio based). When applied to a mixed variable problem of a reinforced concrete beam, MV-FA again outperformed other algorithms found in literature (Gandomi et al., 2011).

3.3.6 – Grey Wolf Optimizer (GWO)

Domain: Continuous/Discrete **Control Parameters:** 2

As the name suggests, GWO is based on the structured hunting techniques of grey wolves. Proposed by Mirjalili, Mirjalili, and Lewis (2014), GWO is one of the newest bio-inspired algorithms presented in literature and was partly developed by a team at Griffith University in Brisbane. The algorithm is based on the observed pack hunting hierarchy that the wolves employ when searching for prey. Each wolf represents a solution with the positions updated according to rules adapted from observations of the animals in the wild. The alpha, beta and

delta wolves represent the best three current positions. The general process for GWO is shown in Figure 8.

```
Initialize the grey wolf population X_i (i = 1, 2, ..., n)
Initialize a, A, and C
Calculate the fitness of each search agent
X_α=the best search agent
X_β=the second best search agent
X_δ=the third best search agent
while (t < Max number of iterations)
    for each search agent
            Update the position of the current search agent
    end for
    Update a, A, and C
    Calculate the fitness of all search agents
    Update X_α, X_β, and X_δ
    t=t+1
end while
return X_α
```

Figure 8 – Grey Wolf Optimizer optimisation process (Mirjalili et al., 2014)

The position updating accounts for the position of the current wolf in relation to the three pack leaders. The movement occurs in spherical manner (n-spheres to be precise), mimicking the encirclement of prey by wolves. The A parameter is a multiplier that is decreased over the iterations, shifting the algorithm from exploration to exploitation by decreasing the relative distances moved between wolves. C is a random perturbation vector that assists to prevent premature convergence.

The performance of GWO was verified by Mirjalili et al. (2014) through testing on both benchmark test functions and classical engineering design problems. The algorithm outperformed PSO, GA and DE in 3 out of 7 unimodal test functions. For multimodal functions, GWO provided competitive performance on many of the functions, often outperforming PSO and GA. In the classical welded beam problem, GWO equalled the outcome of Firefly Algorithm in finding the global optimum.

3.3.7 – Dolphin Echolocation Optimisation (DEO)

Domain: Discrete (combinatorial) **Control Parameters:** 1

Proposed by Kaveh and Farhoudi (2013), Dolphin Echolocation Optimisation is a combinatorial problem solver based on the methods used by dolphins to locate prey. The dolphins generate high-frequency clicks, the sound waves of which strike objects and are reflected, allowing the dolphin to identify the location of prey when visibility is poor. Dolphins search large spaces until a suitable prey is found, analogous to the way in which DEO perform

a global search in an effort to find the optimum. Kaveh and Farhoudi simulated dolphin echolocation by limiting an agent's exploration in proportion to the distance from the target. The process for DEO is outlined in Figure 9.

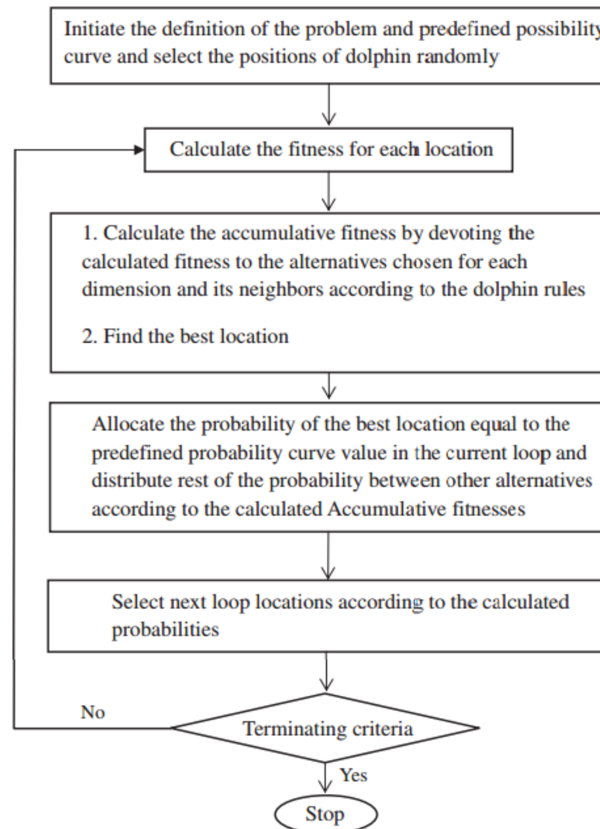


Figure 9 – Dolphin Echolocation Optimisation process (Kaveh & Farhoudi, 2013)

The user is required to define a curve that describes the convergence profile of the algorithm, setting the proportion of computational effort devoted to exploration and exploitation respectively. Although this makes the algorithm less parameter dependant, it puts responsibility onto the user to have an idea about the space being searched. Furthermore, the parameter sets from which variables must take their values are required to be ordered in ascending or descending order prior to running the algorithm.

The authors have published further papers on DEO, applying it to civil structural problems involving steel frame structures (Kaveh & Farhoudi, 2015) and cantilever retaining walls (Kaveh & Farhoudi, 2016). In both instances, DEO was found to be on-par with Differential Evolution in algorithm outcomes. Performance measurements outside of the original authors are limited, with only one seemingly independent source found. Gholizadeh and Poorhoseini (2015) developed a modified DEO by changing the exploration-exploitation curve definitions and found that it outperformed the standard definition of the algorithm. However, no comparisons were made to other algorithms.

CHAPTER 4

ALGORITHM DEVELOPMENT PROCESS

4.1 – ALGORITHM GOALS

After performing the literature review, there were a set of goals that were identified as desirable to be achieved by the new algorithm. These goals aimed to maximise the value of the algorithm to the engineering community.

The first goal was that the algorithm be readily applicable to, and a strong performer on, a broad range of engineering problems. This implied that the algorithm needed to make no assumptions about the search domain. Therefore, algorithm methods were to be as general as possible. This included not directly using the fitness values within the algorithm (such as in Firefly Algorithm), as this may have caused problems when using a penalty function constraint handling approach. The fitness values were to be used for ranking of the swarm only.

The second goal was efficiency in performance. This meant producing results comparable with existing algorithms in a fewer number of total function evaluations. For some particular engineering problems, a function evaluation may take a relatively long time (such as a CFD computation). Therefore, the algorithm was to be as efficient as possible and minimise the required number of iterations to produce fit and trustworthy results.

The third and final goal was simplicity. When performing the literature review, it was found that some particular algorithms were quite mathematically complex, making them hard to comprehend. It was hoped that the new algorithm be easily useable by anyone with good, general scientific or mathematical knowledge, rather than just by experts in the metaheuristic field. Therefore, the algorithm was to be kept as simple as possible in terms of the mathematical methods implemented. This also meant minimising the number of user-selected control parameters.

4.2 – INVESTIGATED INSPIRATION SOURCES

A number of inspiration sources were investigated as being the possible basis for the formulation of the new algorithm. The majority of these sources were animal-based, that would lead to swarm intelligence algorithms. Evolutionary inspiration sources were largely avoided due to Evolutionary Algorithms being markedly more complicated to understand and

implement. For an algorithm that was to be easily understood by engineers not familiar with the BIA space, it was decided that a swarm based inspiration would be more appropriate.

4.2.1 – African Wild Dogs

African Wild Dogs are a canine native to Sub-Saharan Africa. They are characterised by their dark, mottled coats and large ears. They are regarded as one of the most efficient hunters with a kill rate of up to 80%, compared to a lion's 10% (National Geographic, 2014). The African Wild Dog Algorithm of Subramanian et al. (2013) is based upon the dog. However, as discussed in Section 3.3.4, the authors of the algorithm haven't utilised any of the interesting behaviour that the dogs display. This left an opportunity to develop a new algorithm based on the animal that utilises a more advanced method.

The possible behavioural characteristics upon which the algorithm processes could have been based included:

- The dogs are unique amongst social carnivores in that it is the females that scatter from the natal pack once mature, with all females going on to find new packs.
- The males and females have different social hierarchies.
- The young are the first that are allowed to feed on carcasses.
- The dogs chase their prey to exhaustion before encircling and attacking.
- The leaders of the chase change periodically to share the load.

Dr Michael Somers (2016) has described as the dog's hunting behaviour as context dependent and able to be altered to suit the local condition. If this was able to be translated into the code, it would mean the algorithm would be able to adapt to the topology of the local domain, increasing search efficiency. The way in which the females scatter to find new groups also allows for the possible implementation of genetic operators in the algorithm, increasing the explorative capabilities (but perhaps adding some complexity).

4.2.2 – Bottle Nose Dolphins

The Dolphin Echolocation algorithm of Kaveh and Farhoudi (2013) was exclusively for the optimisation of combinatorial (discrete) problems. It was demonstrated through the optimisation of steel truss structures where cross-sections were required to come from a set of standard types. The combinatorial nature of the solver is hard-coded, meaning the algorithm can't be adapted for continuous domains without significant alteration. Therefore, an opportunity existed to develop a dolphin-based algorithm that is able to handle continuous

optimisation for implementation in continuous domain design problems. The possible behavioural characteristics upon which the algorithm processes could have been based include:

- Dolphins' use of echolocation to source their prey.
- Dolphins' tendency to migrate to warmer (more habitable) waters when conditions deteriorate at their current location.
- The dolphin pod technique of herding their target fish into a tight group before feeding on them.
- Communication amongst a dolphin pod based on the dolphins whistling.

Kaveh and Farhoudi didn't utilise any communication between dolphins, beyond the use of probability distributions based around the position of the best dolphin (imposing an inherent communication about the dolphin's current position relative to the best position). Direct communication between dolphins could have been a possible algorithm process to implement.

In nature, if dolphins find an adequate food source and the water temperature is satisfactory they will tend not to migrate. In the algorithm, if a dolphin finds an adequate food source (local optimum) it could be forced to escape and continue searching by "decreasing the water temperature". In code form, this could have been implemented as an escape if a dolphin had been stagnant for a number of iterations.

4.2.3 – Sugar Gliders

Sugar Gliders are a native Australian animal characterised by their compact size and ability to glide between trees. No current optimisation method exists based on the animal, however they displayed some promising behavioural and social characteristics that made them a promising candidate for the algorithm basis. These included:

- The animals can glide for up to 50 metres in a single go, with an average of 20 metres.
- The females produce 1 or 2 offspring per pregnancy.
- Two codominant males lead a colony of 5 to 7 adults and additional young (with the other males being suppressed).
- One of the codominant males are the most likely father of the young.
- The gliders breed more often when a sufficient diet is available and breeding is not restricted to a season.

Solutions would be represented as trees with food sources, with glider agents gliding from tree-to-tree, searching for better solutions. The large glide distance of the glider would allow for sufficient domain exploration, with the glide distance decreasing as the iterations progress and

the colony converges. Also, if the codominant males were taken to be the gliders with the best solutions, genetic operations could be introduced to produce offspring which would likely contain the best information from the previous generation, further increasing the explorative power of the algorithm. The tendency to breed more often when food is abundant can help escape local optima (diverge through producing many offspring when a good food source is found).

4.2.4 – Selection of an Inspiration Source

All three sources were thoroughly investigated, and all seemed promising in the potential to develop a new bio-inspired algorithm. Therefore, a selection matrix was used to distinguish the animals based on three key criteria:

- Interestingness; a measure of how interesting the animal is in general. This accounts for intriguing behavioural characteristics and how unique the animals are.
- Depth of available literature; a measure of how much information is available about the animal, which would serve as sources for inspiration for the algorithm methods.
- Originality; a measure of how novel an algorithm based on this animal would be. Accounts for how many existing algorithms there are for the animal, and how closely they relate to it.

Table 4 gives the selection matrix that assisted in choosing the most promising inspiration source for the algorithm. Each criterion for each animal was given a score from 1-5. Table 4 shows that sugar gliders stood out as the most favourable option. There was no existing BIA based on the animal, meaning the new algorithm would be easily distinguished from the existing literature. Sugar gliders were also a very interesting choice, with their gliding behaviour relatively unique amongst animals. Furthermore, sugar gliders are native to Australia which would came an added bonus; Australian researchers basing their work on an Australian animal.

Table 4 – Algorithm inspiration selection matrix

Criteria	African Wild Dogs	Dolphins	Sugar Gliders
Interestingness	**2** Relatively unknown creatures, and very similar to wolves, hyenas and lemurs.	**3** Known to be very intelligent creatures, meaning that there are many stimulating behavioural characteristics.	**4** Their gliding behaviour is relatively unique amongst animals and may be unknown to many people.
Depth of Available Literature	**2** Not much available literature on the animals due to low numbers in the wild.	**5** Well studied animals mean that there is a wealth of available information about their behaviour.	**3** Relatively lower amount of information about them, but still some qualified sources.
Originality	**3** One existing AWDA algorithm, however it doesn't capture the animal's unique behavioural traits.	**1** There are already two dolphin-based algorithms (DPO and DEO).	**5** No algorithms based on sugar gliders, or similar gliding creatures, currently exist.
Total	**7**	**9**	**12**

4.3 – Development Process

Once the inspiration source of sugar gliders had been selected, work began on constructing the mathematical formulation of the algorithm. The first task was to identify how to link the behaviour of the gliders to the processes used in the optimisation routine.

4.3.1 – Linking Glider Behaviour to the Algorithm Methods

One of the interesting behavioural characteristics of sugar gliders was their hierarchical social structure. Klettenheimer, Temple-Smith, and Sofronidis (1997) observed that there are two codominant males that lead the colony, with other males being suppressed. These two males cooperated with each other in activities such as grooming and fighting, but never cooperated with any subordinate males. From this behaviour, it was decided that the colony of gliders (the algorithm search agents) should be divided into two groups – the codominant gliders and the

subordinate gliders. The codominant gliders would lead the search of the domain, with the subordinates updating their position based on the codominants' position.

Sugar gliders feed on insects, as well as supplementary nectars such as acacia gum and eucalyptus sap when the bugs are scarce. All the food sources for a glider are contained in the trees that they glide between. Therefore, it was decided that a glider's fitness would be represented as the available food as its location. Gliders were then gliding from tree-to-tree in search of the most-abundant food source. The glide distance would decrease as the iterations progressed, as gliders would be continuously finding better food sources and thus would not need to fly as far.

Another interesting characteristic of sugar gliders is that they often occupy more than one den at once. Lindenmayer (2002) found that some gliders simultaneously inhabit up to 13 dens. Gliders search for food in the areas surrounding their den. This implies that the location of their current den impacts and directs their search for food. To simulate this behaviour, it was decided that glider agents would also use a randomly-generated home position to influence their search, with a single home position for each colony. The random generation would occur at each iteration of the loop, introducing greater search space exploration. This was not implemented until the second design iteration, in Section 4.3.3.

4.3.2 – Design Iteration #1

Initial efforts were not directed towards gaining exceptional performance, but rather toward ensuring that the simulated colony was behaving in a way that somewhat emulated real glider behaviour. This primarily entailed observing convergence of the colony, somewhat toward the optimum, as the iterations progressed. Convergence would imply that the gliders are moving toward the colony's best known food source, which is an intuitive behaviour. Convergence of the colony was identified to be the first step toward strong performance.

The colony was divided into the two codominants and the remaining subordinate gliders. However, all gliders used the same position updating method described in Equations 1 to 3.

$$\Delta = 2 * \left(1 - \frac{iteration_{current}}{iteration_{maximum}}\right) \quad (1)$$

$$dx = \Delta * \big((codom_{random} - current) * rand(0,1) + (historicbest - current) * (1 - rand)\big) \quad (2)$$

$$new = current + dx \quad (3)$$

The positions were updated based on the relative distances between a random one of the two codominant gliders, and the colony's best found position. Because the codominant gliders updated the same as the subordinates, there was a chance that the best position would not be carried through the iterations. In hindsight, this was not an ideal outcome.

This iteration of SGA was tested on simple functions throughout the design process. However, even on simple functions (such as a three-dimensional x^2) the results were poor, let alone comparable to existing algorithms. The colony often converged to a random point in space, rather than toward the optimum.

This design iteration also produced some very spurious results. Figure 10 gives examples of colony search histories that resulted in star-like patterns, with gliders mostly moving along straight lines. Behaviour like this occurred randomly, and it was unpredictable as to when it would happen. It was found that this was likely due to the Δ operator linearly decreasing the move distance, without any introduced randomness.

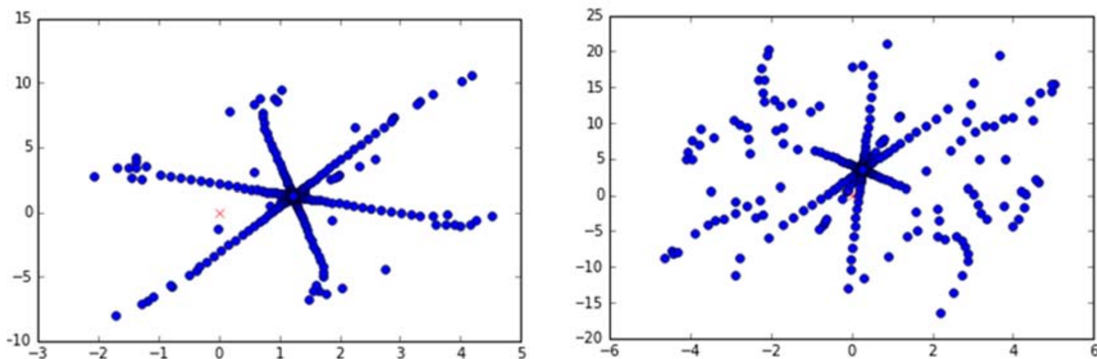


Figure 10 – Erratic search histories

This sort of behaviour was not desired, as the domain was clearly not being searched comprehensively. Therefore, it was decided that the methods used were to be reworked. In order to introduce some more variability in results, the home position was introduced in the second major design iteration.

4.3.3 – Design Iteration #2

The introduction of the home position marked the second major iteration of the algorithm. The home operator aimed to introduce a greater amount of stochastic operation in the algorithm (meaning increased randomness). This was aimed at helping explore the domain more comprehensively.

Updating the codominant gliders in the same way as the subordinate gliders reduced the differentiation between the two groups. As such, it was decided that the two codominant gliders

would not update their positions at all. They would stay stagnant for as long as they were one of the codominants. This would ensure that the best solution would be carried over throughout the generations. The subordinate gliders would base their movements off a random one of the codominant gliders, and the home position (as shown in Equations 4 to 6).

$$\Delta = 0.5 * 2^{1-5\left(\frac{iteration_{current}}{iteration_{maximum}}\right)^4} \tag{4}$$

$$dx = \Delta * \big((codom_{random} - current) * rand(0,1) + (home - current) * (1 - rand)\big) \tag{5}$$

$$new = current + dx \tag{6}$$

The delta operator was now non-linear, and attempted to emphasise exploration by decreasing to zero at a slower rate. Figure 11 shows both the linear and non-linear delta functions (Equations 1 and 4).

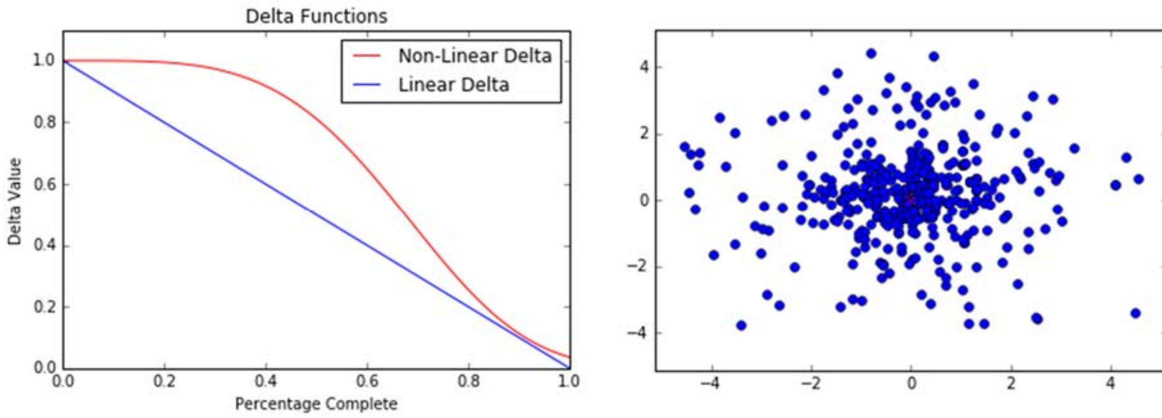


Figure 11 – Delta functions (left) and Correct convergence behaviour (right)

Figure 11 also shows the algorithm working toward finding an optimum at the point $(0, 0)$. As the figure depicts, the domain was now being explored much more thoroughly compared to the first design iteration. The swarm eventually converged towards the optimum. However, the final value produced, whilst generally good, was not comparable with existing algorithms. It was clear that the exploration power of the algorithm was now much better, but the exploitation power needed to be improved.

To maximise the algorithm performance, methods needed to be introduced that increased the ability of the algorithm to find the exact position of the optimum. Section 5.1 shows that the final algorithm increased the exploitation power through the codominant gliders updating their positions based on a local search. Furthermore, the introduction of using both codominant positions for updating the subordinate gliders increased the exploitation power of the algorithm also.

CHAPTER 5

SUGAR GLIDER ALGORITHM

5.1 – ALGORITHM DESCRIPTION

5.1.1 – SGA Pseudocode

The previous chapter has outlined the design process that was undertaken in order to formulate the eventual final version of Sugar Glider Algorithm. Throughout this process the general format of the algorithm only changed slightly. The final outline of the algorithm is expressed in pseudocode in Figure 12.

```
Objective function f(x), x = (x_1, ..., x_d)
Generate initial population of gliders
Evaluate fitness of each glider
WHILE i < iteration max:
    Rank colony and set two codominant gliders
    FOR each codominant:
        Perform local search
        IF fitness new < fitness current:
            Move to new position
    FOR each subordinate glider:
        Update position based on codominants and home den
    i = i + 1
RETURN best fitness, best position
```

Figure 12 – Sugar Glider Algorithm pseudocode

5.1.2 – Method Description

The algorithm starts by initialising the glider colony through assigning variable values, which are randomly selected from the user-defined ranges. The values are stored in a position matrix $\hat{P}$, with n rows (for n gliders) and d columns (for d dimensions).

$$\hat{P} = \begin{pmatrix} x_{1,1} & \cdots & x_{1,d} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \cdots & x_{n,d} \end{pmatrix} \tag{7}$$

The fitness of the colony is then evaluated through calculation of the objective function value for each glider. These values are stored in a fitness vector $\vec{F}$.

$$\vec{F} = \begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix} \tag{8}$$

The algorithm then enters the main loop that iterates until the maximum number of iterations has been reached. First, the colony is ranked in order of best fitness, meaning that the first two rows of the position matrix become the two codominant gliders.

$$\hat{P} = \begin{pmatrix} x_{c1,1} & \cdots & x_{c1,d} \\ x_{c2,2} & \ddots & x_{c2,d} \\ x_{n,1} & \cdots & x_{n,d} \end{pmatrix} \tag{9}$$

The codominant gliders then search for a move by observing a sighted position, sp, through use of the sight distance (SD) parameter, which is default at 0.1:

$$SD = (0, 1) \tag{10}$$

$$rand = random\big(1 - SD(1 + t), 1 + SD(1 - t)\big) \tag{11}$$

$$sp = current * rand \tag{12}$$

The variable t is the time factor, which linearly increases $[0 \rightarrow 1]$, and is given through:

$$t = \frac{iteration_{current}}{iteration_{max}} \tag{13}$$

The objective function is then evaluated for the sighted position. If the fitness at the new location is better, a move is performed. Otherwise, the codominant stays in its current position.

Next, the subordinate gliders' positions are updated. Three random vectors assist to increase the variability and prolong the convergence:

$$r1 = random(t, 2 - t) \tag{14}$$

$$r2 = random(t, 2 - t) \tag{15}$$

$$r3 = random(-2, 2) \tag{16}$$

The distance to move is then calculated by the addition of distances to the two codominants and the home den position:

$$CP = [1, 10] \tag{17}$$

$$\begin{aligned} dx = r1 * (codom_1 - current) + r2 * (codom_2 - current) * t^{\frac{1}{CP}} \\ + r3 * (home - current) * (1 - t)^{CP} \end{aligned} \tag{18}$$

The time factor (t) and a convergence power parameter (CP) act on both the home and second codominant distances such that a weighting toward the codominant distance increases throughout the iterations. The weightings are the $t^{\frac{1}{CP}}$ and the $(1 - t)^{CP}$ factors of Equation 18. Figure 13 gives the time plot for the default CP value of five over the iterations.

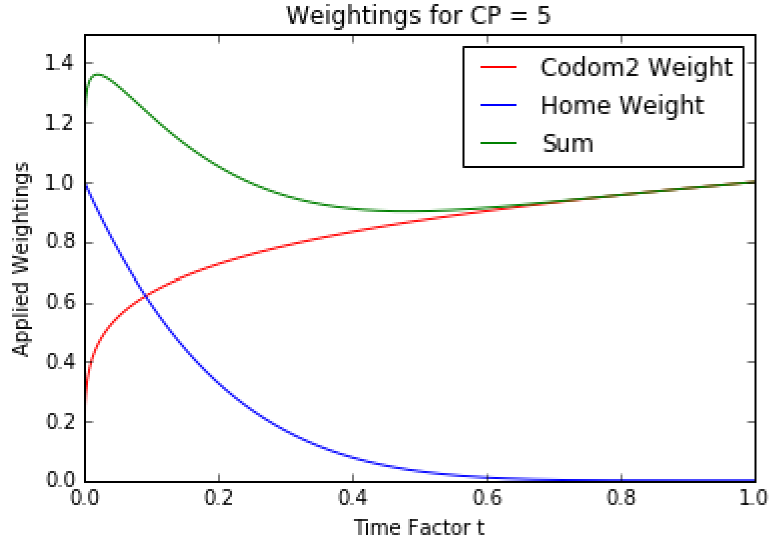


Figure 13 – Weightings for a CP value of 5 (the default)

Towards the later stages, dx will simply direct the gliders towards the codominants. The codominants will ideally, by this stage, be positioned in the same local area. Therefore, it is clear that:

$$\lim_{t \to 1} dx = 2 * (codom - current) \tag{19}$$

Therefore, the move distance must be halved in order to ensure proper convergence of the swarm:

$$new = current + 0.5 * dx \tag{20}$$

The $\hat{P}$ matrix is then updated with the new glider positions. The fitness values for the new subordinate glider positions are then evaluated and the $\vec{F}$ matrix is updated. The codominant and subordinate gliders then continuously update their positons until the maximum number of iterations has been reached.

The algorithm is easily extended to multiple colonies via adding an extra dimension to each matrix, representing the colony number. The supplied code has this built in.

5.1.3 – Convergence Power Parameter

The convergence power is a parameter that acts to alter the weightings applied to the distances moved towards the second codominant and the colony home. Figure 14 gives plots for the weightings for different values of CP, with the valid range defined in Equation 17. It can be seen that as CP is increased, the gliders move more toward the second codominant and less toward the random home position.

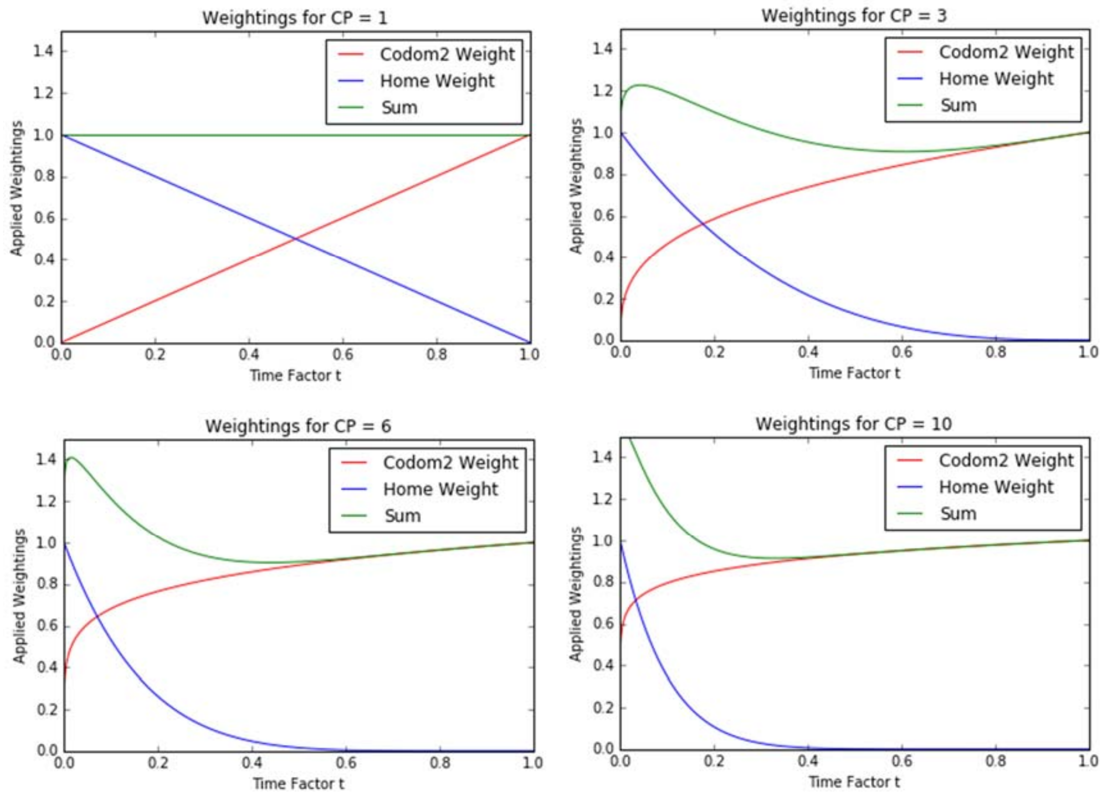


Figure 14 – Effect of the CP parameter on the weightings

The convergence power acts to increase the convergence rate of the swarm. Figure 15 depicts this effect on a test run of a unimodal test function shown in Equation 21.

$$f(\vec{x}) = x^2 + y^2 \tag{21}$$

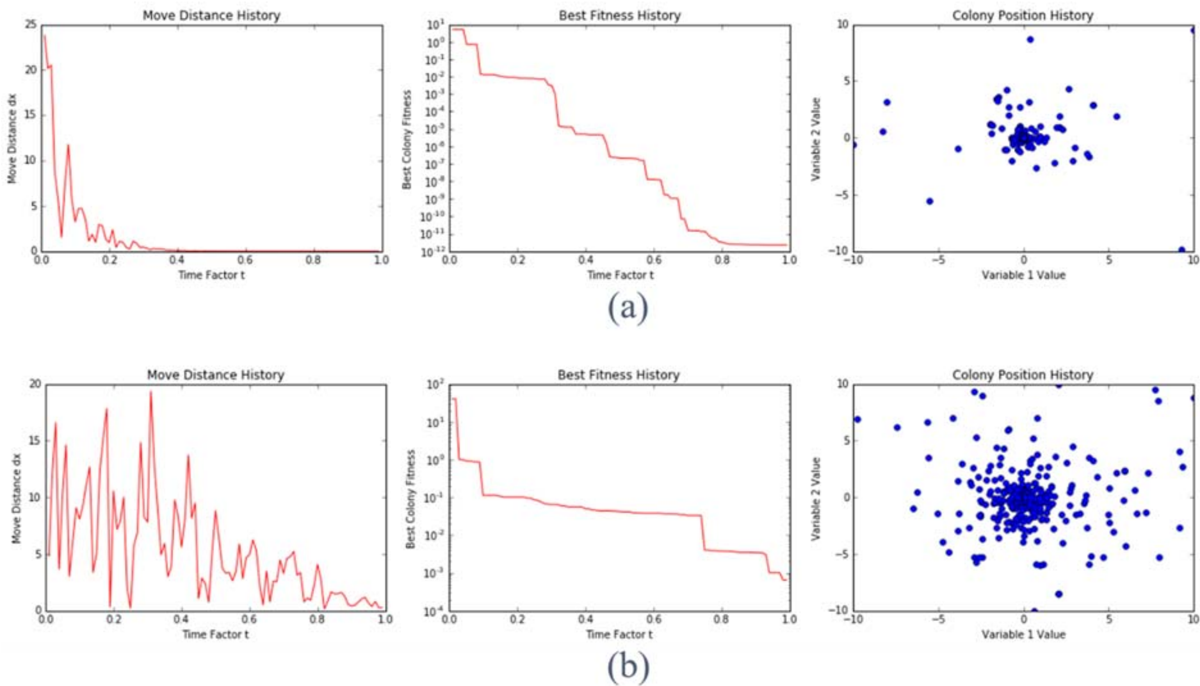


Figure 15 – Convergence behaviour for CP = 10 (a) and CP = 1 (b)

Because the function is highly unimodal, and of a low dimension, the run with a CP of 10 outperforms the run with a CP of 1. However, the search space exploration is much lower for the high convergence power. This is expected, and is exactly the desired effect of the CP parameter. The parameter has been tuned in Section 5.2.3.

5.2 – Parameter Tuning

The algorithm has three parameters that must be tuned to ensure optimal performance of the algorithm. These parameters are the sight distance, the convergence power and the colony size.

5.2.1 – Tuning Functions

For tuning the sight distance, a set of 5 unimodal functions were chosen. This is because the sight distance is an exploitive operator that works to find the local optimum in the current location. Unimodal functions are best used to test the exploitation characteristics of algorithms as there is a single optimum to converge towards. The functions used to tune the sight distance are listed in Table 5.

Table 5 – Tuning functions for the sight distance

Name	Formula		Type	Dim	Min
Sphere	$f(\vec{x}) = \sum x^2$	(22)	US	30	0
SumSquares	$f(\vec{x}) = \sum ix^2$	(23)	US	30	0
Matyas	$f(\vec{x}) = 0.26(x^2 + y^2) - 0.48xy$	(24)	UN	2	0
Schwefel 2.22	$f(\vec{x}) = \sum \lvert x \rvert + \prod \lvert x \rvert$	(25)	US	30	0
Dixon-Price	$f(\vec{x}) = (x_1 - 1)^2 + \sum 2x_i^2 - x_{i-1}$	(26)	US	30	0

For tuning the convergence power and population size, a mixed set of functions was chosen that were a combination of unimodal/multimodal and separable/non-separable. This was done to ensure that the parameter values chosen provided acceptable performance across a range of objective function types. This means the algorithm can be applied to many functions without needing modification. The functions used to tune the convergence power and population size are listed in Table 6.

Table 6 – Convergence power and colony size tuning functions

Name	Formula		Type	Dim	Min
Sphere	$f(\vec{x}) = \sum x^2$	(27)	US	30	0
SumSquares	$f(\vec{x}) = \sum ix^2$	(28)	US	30	0
Matyas	$f(\vec{x}) = 0.26(x^2 + y^2) - 0.48xy$	(29)	UN	2	0
Schaffer	$f(\vec{x}) = 0.5 + \frac{\sin(x^2 + y^2)^2 - 0.5}{1 + 0.001(x^2 + y^2)^2}$	(30)	MN	2	0
Schwefel	$f(\vec{x}) = \sum -x \sin\left(\sqrt{\lvert x \rvert}\right)$	(31)	MS	30	-12569.5

30 tests were run for testing each parameter value. Each test was for 1000 iterations of a colony of five gliders. The numerical results obtained have not been reported; the important outcome is the result relative to the others obtained for different parameter values.

5.2.2 – Sight Distance

The sight distance parameter designates by what percentage a codominant glider can fluctuate its values by. This is analogous to how far the glider can search, in its local domain, for a tree with a greater food source. The sight distance SD can take any value in the range designated in Equation 32.

$$SD = (0, 1) \tag{32}$$

In order for the search be considered local, the chosen value would be towards the lower end of the range. Furthermore, the codominant glider is already in a quasi-optimal location, and so to improve its position, only a small change would be required. This would help ensure sufficient exploration of the local domain. Therefore, the range of sight distance values tested were those given by Equation 33.

$$SD = [0.01, 0.05, 0.1, 0.2, 0.3] \tag{33}$$

In order to observe the effects of altering the sight distance, a count was made of how many times a particular sight distance resulted in a better solution. Furthermore, the average percentage change in the fitness was also recorded. Along with the accuracy of the final solutions obtained, this allowed for a full evaluation for the performance implications of changing the sight distance.

Figure 16 shows the average final result of implementing SGA with the given sight distance. The results have been normalised using feature scaling, such that the scores are all then in the

range of [0, 1]. A lower score implies greater performance. Figure 16 shows that a sight distance of 0.1 results in the lowest average function outcomes. It also has the lowest maximum value and the smallest spread across the data points. This is important as it implies good all-around performance.

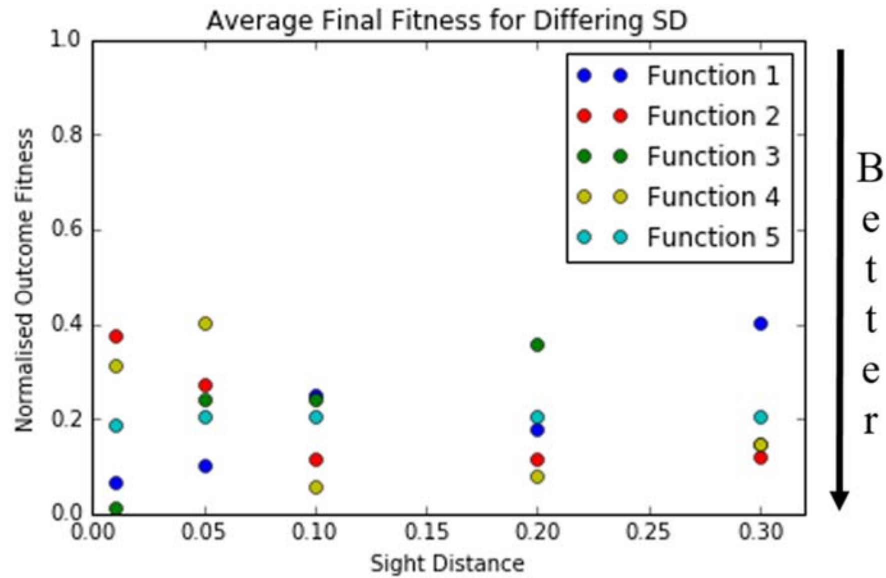


Figure 16 – Outcome f(x) values for differing SD

Figure 17 is a plot of the total fitness increases gained by the codominant gliders. It is the multiple of the number of times the position was updated with the fitness increase each time. Here, a larger value is more desirable, as it implies the position updating was more effective. In general, a smaller sight distance implied a larger number of fitness increases, but for a smaller gain each time. The opposite was true for a larger sight distance. The data has again been normalised using feature scaling.

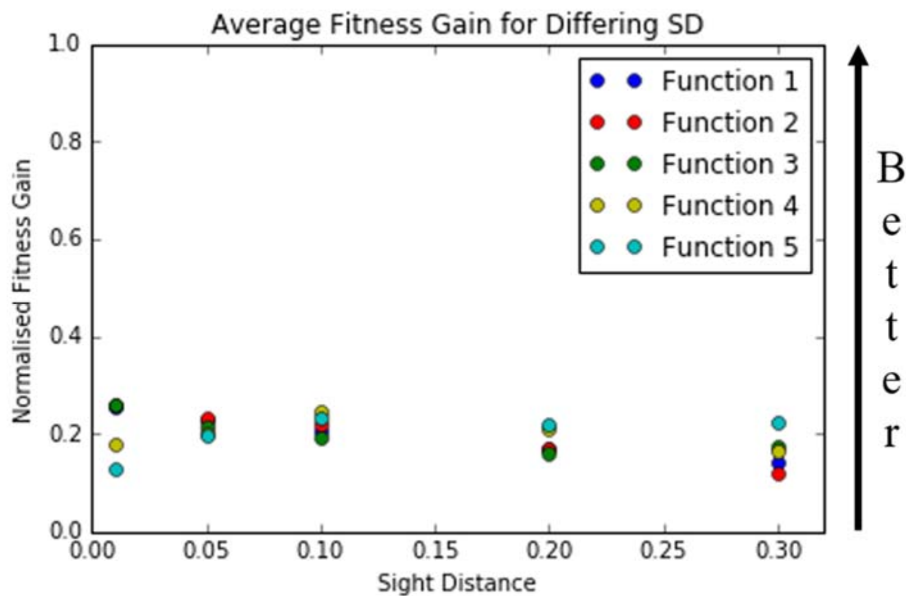


Figure 17 – Average fitness gain for differing SD

Figure 17 shows that the sight distances of 0.05 and 0.1 seemed to be equally effective in the average fitness gain. The average gain for 0.1 was very slightly higher, with 0.05 giving a smaller spread. The values towards the end of the range experienced a higher spread, meaning their performance was inconsistent. This is not desirable, and as such, they were not selected.

Based on the results, a distance of 0.1 seems to be the value for an optimal balance between the two trade-offs. As such, a sight distance SD of 0.1 has been selected as a default value.

5.2.3 – Convergence Power

The convergence power CP acts to alter the relative weightings of the distances moved towards the home and second codominant positions. It is recommended that the user set the convergence power to an appropriate value for the problem at hand. However, a default value for CP still needed to be set; one that gave good all-round performance. The convergence power range tested was the set $[1, 10]$, in integer increments.

CP values of 1 and 2 did not converge the colony until the very late stages, meaning the optimum was not exploited fully. This caused exceedingly sub-par performance on the first two US-type objective functions, meaning that these values have been omitted (such that the feature scaling normalisation was interpretable).

Figure 18 is a plot of the average final optimisation result against the convergence power used. A smaller value is more desirable, as the problems are all minimisation type. Figure 18 shows that values on the extreme ends of the CP range gave mixed performance across the five test functions. A CP value of 5 gave the best performance overall, with the smallest spread and the lowest maximum value.

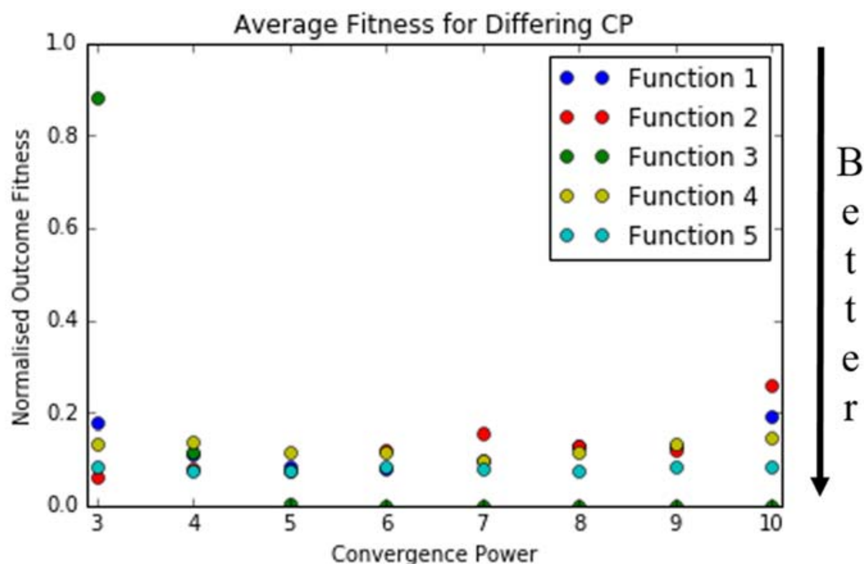


Figure 18 – Outcome f(x) for differing CP

Figure 19 is a plot of the standard deviations across the data sets for each function (with feature scaling applied). A low standard deviation is important to ensure that the result of a single optimisation run can be trusted with confidence. Again, a CP value of 5 gave the best overall performance amongst the range with the smallest maximum standard deviation and equal spread of standard deviations.

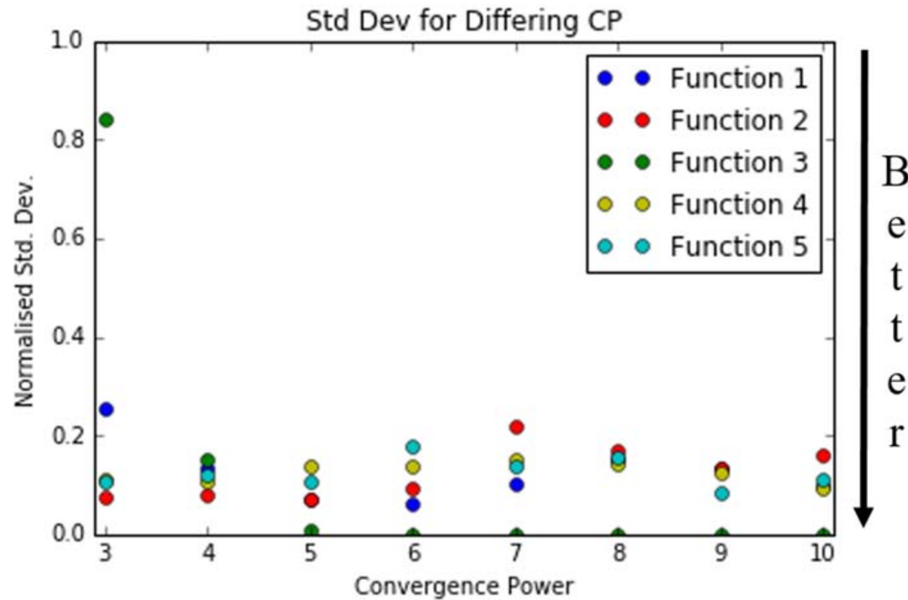


Figure 19 – Outcome standard deviation for differing CP

Due to a CP value of 5 giving the best performance in both aspects, it was selected as the default value for the convergence power. Section 5.4.2 gives advice on how to select the ideal convergence power for a given optimisation problem.

5.2.4 – Colony Size

The colony size required tuning in order to find a value that gave good all-around performance on a range of functions. The colony size parameter was tuned using an equal-NFE basis. Therefore, a larger colony size meant a lower number of algorithm iterations (for the same number of total function evaluations). This ensured a fair comparison between the values tested. The minimum number of gliders is three; two codominants and one subordinate. Gliders in the wild typically live in colonies of five to seven, however the maximum value tested was ten to ensure the optimal value was selected. There is no maximum limit to the number of gliders the user can opt to use.

Figure 20 is a plot of the final function values over 30 repeated runs. The figure depicts that values from five onwards show minimal difference in the results obtained. Values below five show significantly worse performance.

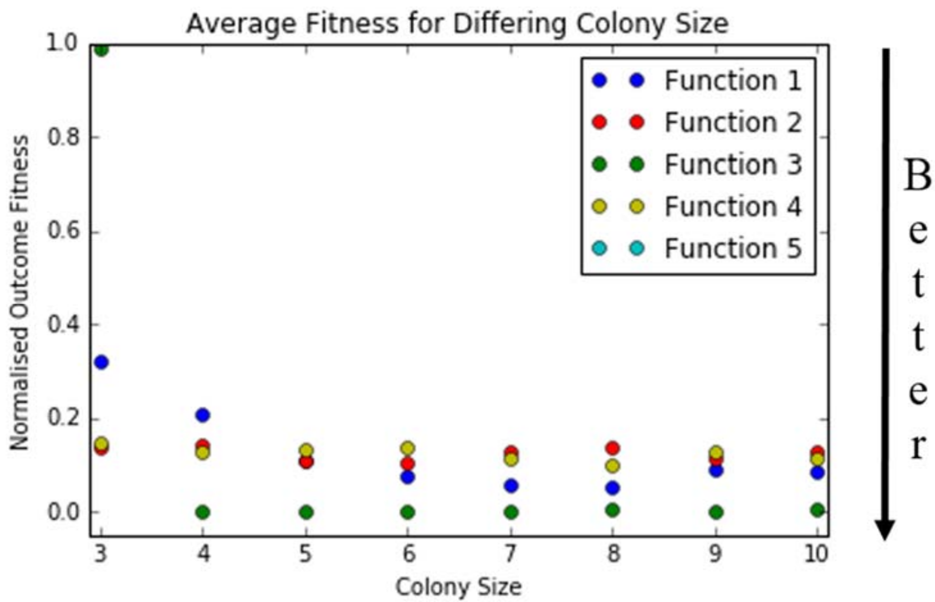


Figure 20 – Outcome f(x) for differing colony size

Figure 21 is a plot of the standard deviations between the results obtained over the 30 repeated runs. The figure depicts that a colony size of five gave the minimum values for standard deviation, however hot much of an increase is observed for values greater than five. Again, a colony of three or four gliders gave sub-par performance compared to the rest of the values.

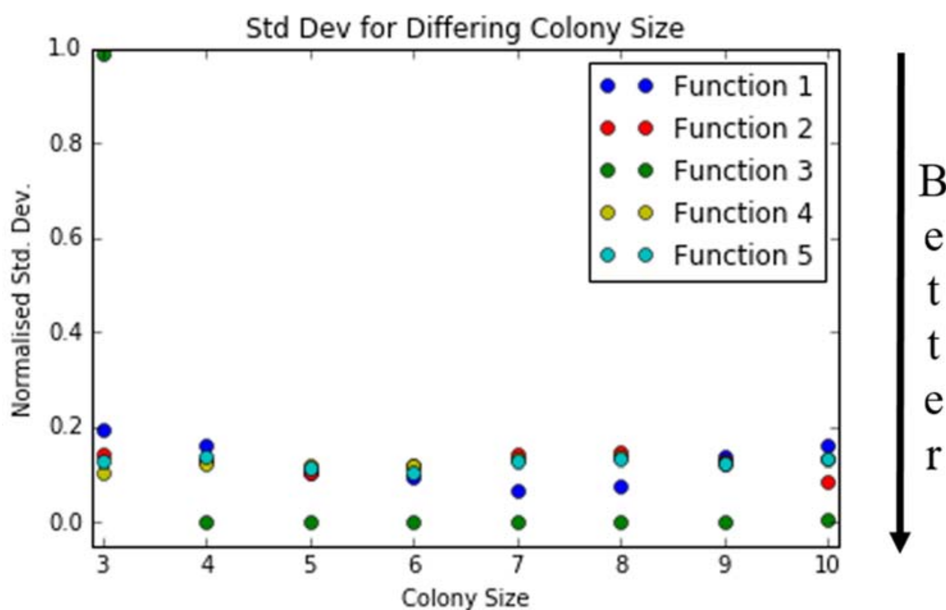


Figure 21 – Outcome standard deviation for differing colony size

Based on the results of the two metrics, a colony size of five seemed to give the best balance of codominant and subordinate gliders, and was selected as the default value. A colony of five gliders showed good performance across both the minimum values and the standard deviations across the 30 runs. This also tied in well with the fact that gliders typically form colonies of five to seven in the wild.

5.3 – Comparison to Existing Algorithms

5.3.1 – General Method Comparison

Forms of swarm intelligence algorithms use a position updating procedure that often utilises a combination of the following methods:

- Relative position between the search agent and the swarm's best known position.
- Relative position between the search agent and its own best known position.
- Relative position between the search agent and an agent with a better current position.
- Random vector operators to vary the distance moved.

SGA utilises methods derived from the first and fourth points. The codominant search method of fluctuating positional values is understood to be original within the field. SGA is further differentiated from other swarm intelligence algorithms in the following aspects:

- The codominant gliders update their positions in a separate manner to the rest of the gliders. This separates the colony into two groups and allows the domain to be concurrently explored and exploited.
- The introduction of a convergence power that is applied to the time factor to facilitate convergence behaviour changes, allowing the user to customise the process to suit the particular domain.
- The introduction of a random home position which increases variability in early iterations of the optimisation process. This increases the strength of the algorithm in multimodal domains.

The combination of these points ensures that the proposed algorithm is in fact novel amongst the existing literature.

5.3.2 – Algorithm Parameter Comparison

It can be argued that the parameters for SGA are significantly more intuitive to select than those of the other algorithms. The sight distance SD and population size have been given recommended values in Section 5.2, therefore it is unlikely that users will decide to alter them. Beyond the number of iterations, the convergence power is the only parameter that is recommended to be altered by the user. However, the convergence power has also been studied in Section 5.2.3, with a simple guide for selection of its value in Section 5.4.2.

In contrast, many other algorithms have parameters whose effect on results is not as instinctual. These parameters may be well understood by researchers in the field, but to typical engineers they are likely hard to comprehend without extensive study of the topic. For example, Particle

Swarm Optimisation has four control parameters. One of which is the population size, which is a straightforward selection. But the inertia weight and random φ operator ranges are less intuitive. Although there has been extensive work on the topic of tuning PSO parameters, it is still an added step to the optimisation process that SGA doesn't require.

To an extent, evolutionary inspired algorithms suffer from this problem even more than their swarm intelligence counterparts. Evolution Strategy, Genetic Algorithm and Differential Evolution all contain selectable parameters that are hard to conceptually understand. This is mostly because the parameters are drawn from the sometimes-complex genetic process occurring at a microscopic level. For example, the EA variant tested by Krink, Filipic, and Fogel (2004) had five control parameters. Again, one was the population size. However, the remaining four related to crossover, selection, mutation and tournament selection methods, all of which are not often understood by those outside the field. The fact that the parameters of SGA are easy to select further strengthens its presence amongst the available bio-inspired algorithms.

5.4 – Python Code and User Recommendations

The Sugar Glider Algorithm has so far been actualised in Python code. The code formulations can be found in Appendices 2 through 9. Some recommendations have also been formulated that should assist in helping end users to understand how to use the algorithm.

5.4.1 – SGA in Python

Python was chosen as the first implementation platform due to both familiarity and popularity within the engineering community. This should mean that the developed algorithm is able to be employed by a large user base, immediately. The provided Python code currently includes:

- SGA for continuous domains (Appendix 2)
- SGA for integer-valued discrete domains (Appendix 3)
- SGA with built in plotting functions showing swarm behaviour (Appendix 4)
- SGA for the mixed-variable pressure vessel problem in Section 6.2.3 (Appendix 5)
- The benchmarking function test scripts (Appendices 6 and 7)

A barebones Python script has also been included in Appendix 9. The script simply requires the user to fill in the function definition for which they want to test (including any constraints), and define variable ranges. When the script is run, SGA will optimise the function and return the resulting values f_{opt} and x_{opt}.

When using SGA in Python, it is accessed through a function call. There are three necessary parameters that must be passed to the SGA function at runtime. The first is the Python function object to be optimised, which takes a single list of input variables. The second and third arguments are the lower and upper variable ranges respectively. The lengths of the variable bounds define the dimension of the problem, and the algorithm uses this information in its routines. Figure 22 gives a simple example of implementing SGA in code form.

```
from SGA import sga

def obj_func(var_list):
    x, y, z = var_list
    return x**2 + y**4 + z**6

lower_bound = [-10, -10, -10]
upper_bound = [10, 10, 10]

result = sga(obj_func, lower_bound, upper_bound)

print 'Minimum is', result[0]
```

Figure 22 – Example implementation of SGA in Python

When wanting to use SGA to optimise functions that are not represented by mathematical formulas, the objective function must be modified such that it can connect to the external "fitness-generating" mechanism (be it another computer program or physical measurements).

There are a number of other options that can be set when running the Python command. Table 7 lists these options and their valid value/s. These options aim to help customise the optimisation process such that the user gains maximum accuracy and efficiency.

Table 7 – SGA Python input parameters

Parameter	Description	Valid Value/s
`itermax`	The maximum number of optimisation iterations, which is default at 1000.	$[1, \infty]$
`colonies`	The number of colonies of gliders, which is default at 1.	$[1, \infty]$
`gliders`	The number of gliders per colony, which is default at 5.	$[3, \infty]$
`cp`	The convergence power (CP) value, which is default at 5.	$[1, 10]$
`sd`	The sight distance (SD) value, which is default at 0.1.	$[0, 1]$

`guess`	An initial guess of the optimum location. Only recommended to be used if there is significant confidence in the guess.	$[x_1, \cdots, x_d]$ ($Python\ list$)

5.4.2 – Parameter Value Recommendations

The number of gliders within a colony is recommended to be set at five. For more accurate results, instead of adding more search agents, it is recommended that the user simply increase the number of iterations. Section 5.2.4 shows that no performance increases were observed for colonies with greater than five gliders. Therefore, the easiest way to guarantee a performance increase is to simply increase the number of iterations performed by SGA.

It is recommended that the sight distance (SD) parameter not be altered by the user, unless they are prepared to undertake problem-specific tuning to determine its ideal value. The tuning in Section 5.2.2 showed that larger sight distances decreased the chance of finding a better solution. However, when a better solution was found, its magnitude increase in fitness was larger. The resulting combination of these factors was optimal at a value of 0.1, across a range of problems. Therefore, the sight distance has already been tuned to what is understood to be a generally good value. No easy parallel can be drawn between objective function domain topology and a suitable sight distance value, meaning problem-specific tuning should be performed if the user wants to alter the value. However, it is recommended that this time should rather be devoted to extra algorithm iterations, as this would likely increase the performance by a larger amount than any tuning of the SD parameter.

Other than the number of iterations, the convergence power (CP) is the only control parameter that the user needs to set before running the algorithm. As previously outlined, the parameter is recommended to take a value in the range $[1, 10]$, where the default value is 5. The ideal CP value is dependent on the modality of the function and the size of the search domain. For highly unimodal functions, the convergence power may be increased towards the higher end of the range, with 10 being the recommended limit to try and ensure adequate exploration. For highly multimodal functions, especially those including discontinuities from constraints, it is recommended that the convergence power be set toward the lower end of the range. Furthermore, the domain size must be considered when setting the parameter value. A smaller domain size means the convergence power can be increased due to a lower exploration requirement. CP values in the range $[0, 1]$ are valid, however they severely reduce the convergence rate meaning that effective exploitation of the optimum will likely not occur.

CHAPTER 6

Performance Benchmarking

6.1 – Classical Mathematical Functions

Testing on unconstrained mathematical functions has been performed as it allowed for a generic performance comparison to be made against existing algorithms.

6.1.1 – Test Functions

The test functions that have been used are commonplace amongst researchers in the field. They can be classified in two ways:

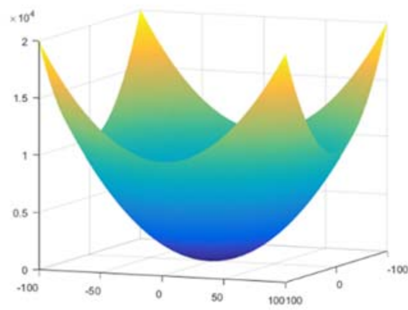
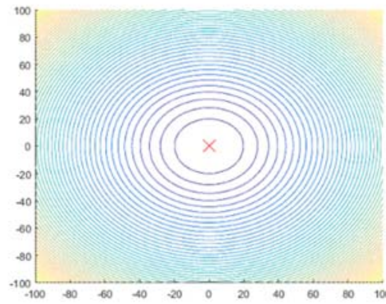
- Unimodal (U) and multimodal (M), which refers to the number of optima in the domain (with multimodal functions regarded as harder to solve), and
- Separable (S) and non-separable (N), with separable functions generally easier to solve as the variables are independent from each other (meaning the problem can be likened to simultaneously optimising a larger number of simpler functions).

Table 8 outlines the definitions and characteristics of the test functions used. The functions are a good mix of type, ensuring acceptable broad-ranging performance.

Table 8 – Mathematical benchmarking function definitions

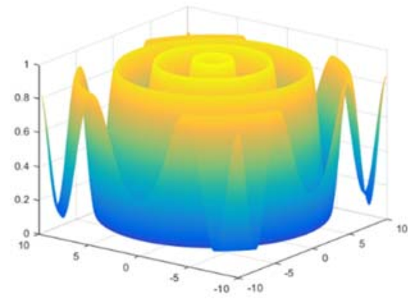
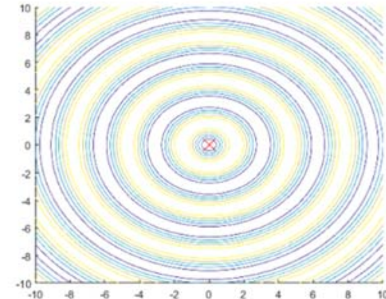
Name	**Definition**	**Type**	**Range**	**Dim**
Sphere (F1, 34)	$\sum x_i^2$	US	[-100, 100]	5
Schaffer (F2, 35)	$0.5 + \frac{\sin^2\left(\sqrt{x_1^2 + x_2^2}\right) - 0.5}{\left(1 + 0.001(x_1^2 + x_2^2)\right)^2}$	MN	[-100, 100]	2
Griewank (F3, 36)	$1 + \frac{1}{4000}\sum (x_i - 100)^2 - \prod \cos\left(\frac{x_i - 100}{\sqrt{i}}\right)$	MN	[-600, 600]	50
Rastrigin (F4, 37)	$\sum x_i - 10\cos(2\pi x_i) + 10$	MS	[-5.12, 5.12]	50
Rosenbrock (F5, 38)	$\sum 100\left(x_{i+1} - x_i^2\right)^2 + (x_i + 1)^2$	UN	[-50, 50]	50

Figure 23 visually depicts the 2D versions of the test functions, as well as their contour maps.

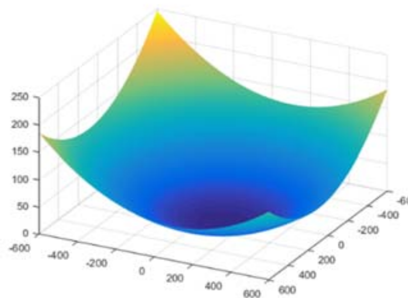
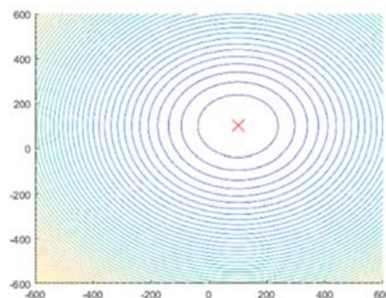
Sphere

Eq. 34

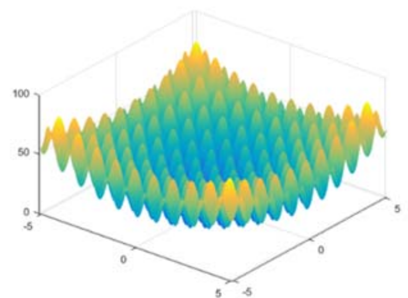
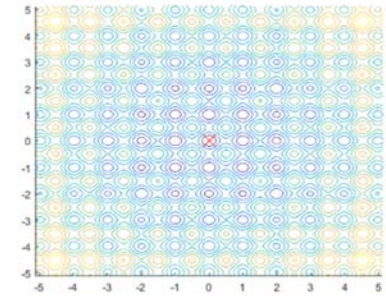
Schaffer

Eq. 35

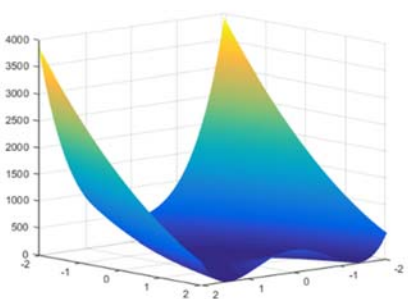
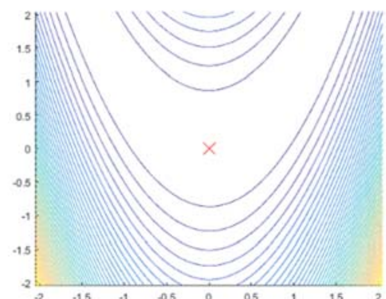
Griewank

Eq. 36

Rastrigin

Eq. 37

Rosenbrock

Eq. 38

Figure 23 – Visual 2D representation of the benchmarking functions

As can be seen, some of the functions are highly multimodal in nature. Griewank is classed as multimodal, however its 2D representation doesn't depict its multimodal nature very well (which is more prevalent in higher dimensions). The multimodal functions test the explorative characteristics of the algorithm, ensuring it can avoid local minima and find the area of the global optimum effectively. On the other hand, unimodal functions test the exploitation power, ensuring it can find the exact value of the function minimum.

6.1.2 – Benchmark Settings

The algorithm's performance has been compared to that of three existing bio-inspired algorithms. The results, and therefore the algorithmic settings, for DE, PSO and an EA have been taken from Krink et al. (2004). As explained in Section 5.4.2, there is only one control parameter for SGA that is recommended to be altered by the user; the convergence power CP. As the set of test functions is a mix of both unimodal and multimodal, it was decided that the convergence power be left to its default value of 5. This allowed for a demonstration of the general performance of the default algorithm. Table 9 lists the control parameter values for all the algorithms.

Table 9- Algorithm parameter values

DE		**PSO**		**EA**		**SGA**	
$PopSize$	50	$PopSize$	20	$PopSize$	100	$Colonies$	1
CF	0.8	w	1 → 0.7	p_c	1.0	$Gliders$	5
f	0.5	φ_{min}	0.0	p_m	0.3	CP	5
		φ_{max}	2.0	σ_m	0.01	SD	0.1
				n	10		

To ensure a fair comparison, the same runtime setting as Krink et al. (2004) have been used. These parameters are listed in Table 10 below.

Table 10 – Performance benchmarking test settings

Parameter	**Value**
NFE (F1, F2)	100,000
NFE (F3, F4, F5)	500,000
No. of Runs	30

6.1.3 – Results

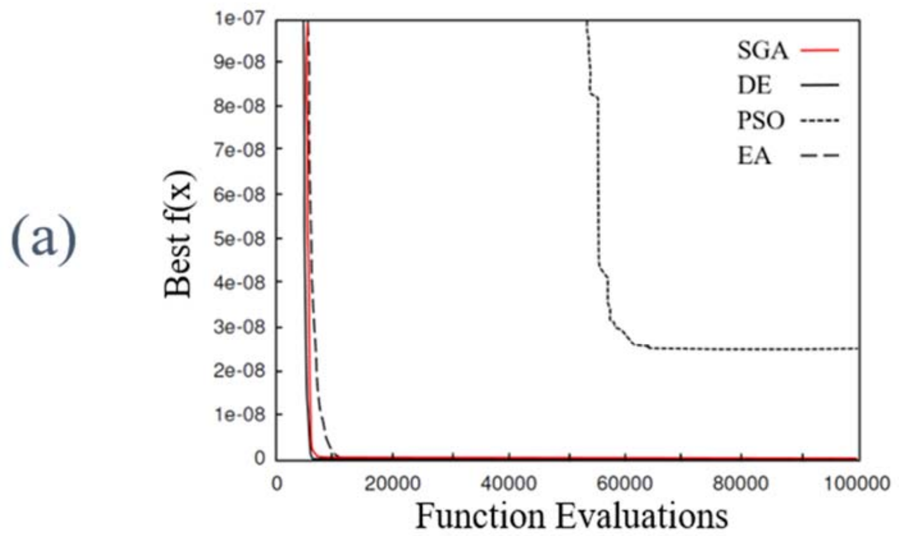
Table 11 shows the results for the test functions used. The average function value and the standard deviation across 30 runs are presented. All functions are minimisation problems with a global optimum value of zero. Values below E-12 have been presented as zero.

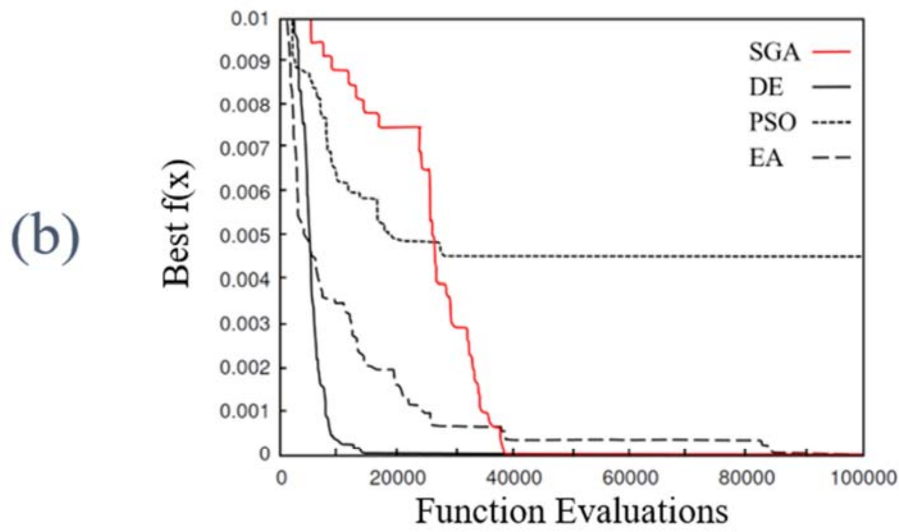
Table 11 – Mathematical benchmarking results (Krink et al., 2004)

Function		DE	PSO	EA	SGA
Sphere	Mean	0	2.51E-08	0	0
	Std Dev	0	0	0	0
Schaffer	Mean	0	0.00453	0	0
	Std Dev	0	0.00090	0	0
Griewank	Mean	0	1.549	0.00624	0
	Std Dev	0	0.06695	0.00138	0
Rastrigin	Mean	0	13.1162	32.6679	261.842
	Std Dev	0	1.44815	1.94017	32.114
Rosenbrock	Mean	35.3176	5142.45	79.818	39.1265
	Std Dev	0.2744	2929.47	10.4477	0.19824

The results show that SGA is competitive with the existing BIAs. It achieved the global minimum in 3 of 5 functions, whereas EA achieved 2 of 5 and PSO didn't find the global optimum for any function. DE outperforms SGA on the last two functions, however the difference in the Rosenbrock function is only small. SGA outperformed PSO and EA on all functions apart from Rastrigin. The result for the Rastrigin function is the worst compared to the other three algorithms. This is likely due to the highly multimodal nature of the function. As such, performance would be expected to improve if a lower convergence power was used.

Figure 24 (over the page) shows the averaged best fitness history curves for the 30 runs of each function. Figure 24 (a) shows that SGA and DE give almost identical performance for the Sphere function. EA converges to zero slightly slower, whilst PSO only reaches 2.5E-08 as an average minimum. Figure 24 (b) shows that SGA initially converges slower than the other algorithms for the Schaffer function, but then converges to the optimum faster than EA. PSO again only reaches 0.00453 as the average minimum. Figure 24 (c) shows that SGA outperforms the other three algorithms in terms of finding the Griewank function optimum earlier in the optimisation process. PSO again fails to converge to the optimum.





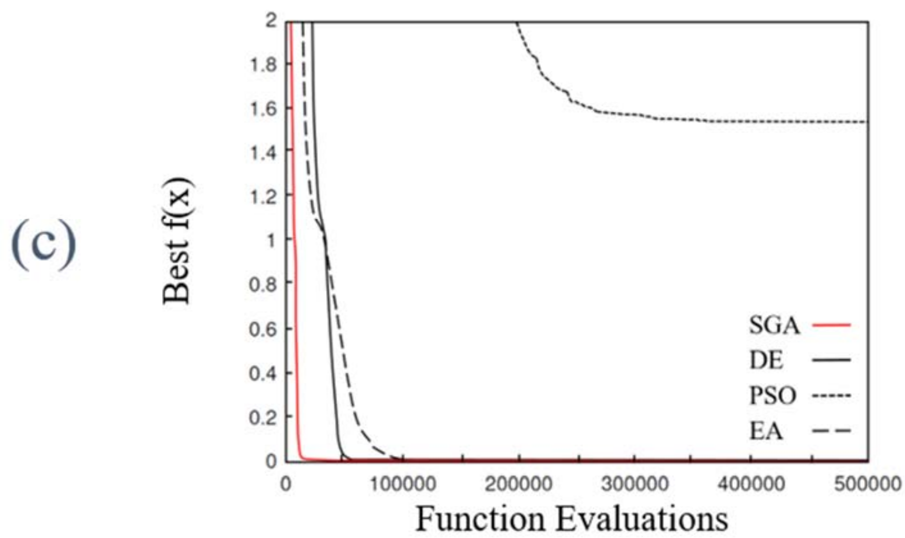


Figure 24 – Optimum convergence behaviour for the Sphere (a), Schaffer (b) and Griewank (c) functions (Krink et al., 2004)

6.2 – Constrained Engineering Design Problems

Three engineering problems have been selected to demonstrate the applicability of the algorithm to mechanical design, taken from Kannan and Kramer (1994). These constrained optimisation problems have been well studied in the literature, and solved by various different numerical optimisation methods. This has allowed for a performance comparison to be made against several existing bio-inspired algorithms.

The constraints in the problems mean that many of the solutions in the search space are infeasible. A measure of the feasible search space, suggested by Michalewicz (1996), was to take a large sample of random points and find the ratio of feasible solutions to the total number of solutions, as shown in Equation 39.

$$\rho = \frac{N_{feasible}}{N_{total}} \tag{39}$$

Mezura-Montes and Coello (2008) used 1,000,000 random samples to calculate the percentage of feasible search space for the three constrained design problems, with the results in Table 12.

Table 12 – Calculated feasibility percentages

Design Problem	**ρ (% Feasible)**
Coil Spring	0.7537
Welded Beam	39.6762
Pressure Vessel	2.6859

A lower ρ measure indicates that it is much harder to generate feasible solutions. This tests the ability of the algorithms to navigate toward the feasible region, so as to not waste function evaluations in the infeasible region.

6.2.1 – Coil Spring Design

The objective of the coil spring design problem is to minimise the total mass via alteration of the spring dimensions. The design is subject to constraints on deflection, shear stress and surge frequency that limit the feasible space. The variables also have limits on their range of valid values. Figure 25 presents the physical design of the spring.

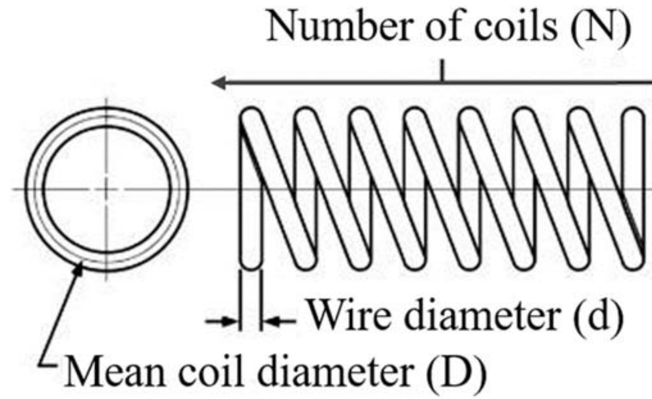


Figure 25 – Physical coil spring dimensions

The coil spring design problem has the smallest feasible solution area, with a ρ measure of just 0.7537%. Solutions in the other 99.2463% of the search space fail at least one of the constraints. This is an extremely small region, and truly tests the ability of the algorithm to navigate towards the feasible space.

The problem is mathematically formulated in Equations 40 to 48:

With $$\vec{x} = [x_1, x_2, x_3] = [d, D, N] \quad (40)$$

Minimise $$f(\vec{x}) = (x_3 + 2)x_2 {x_1}^2 \quad (41)$$

Subject to $$g_1(\vec{x}) = 1 - \frac{{x_2}^3 x_3}{71785 {x_1}^4} \leq 0 \quad (42)$$

$$g_2(\vec{x}) = \frac{4{x_2}^2 - x_1 x_2}{12566(x_2 {x_1}^3 - {x_1}^4)} + \frac{1}{5108 {x_1}^2} - 1 \leq 0 \quad (43)$$

$$g_3(\vec{x}) = 1 - \frac{140.45 x_1}{{x_2}^2 x_3} \leq 0 \quad (44)$$

$$g_4(\vec{x}) = \frac{x_1 + x_2}{1.5} - 1 \leq 0 \quad (45)$$

With variable ranges $$0.05 \leq x_1 \leq 2 \quad (46)$$

$$0.25 \leq x_2 \leq 1.3 \quad (47)$$

$$2 \leq x_3 \leq 15 \quad (48)$$

6.2.2 – Welded Beam Design

The welded beam design problem attempts to minimize the total material and fabrication cost of a beam that is loaded in bending. Beam dimensions are varied to reduce the total mass (thus reducing material cost). However, the cost of welding is also considered, introducing more complexity to the problem. The objective function is the total cost, and is minimised subject to constraints on shear and bending stresses, buckling loads and end deflection. The variables also have limits on their range of valid values. Figure 26 depicts the problem.

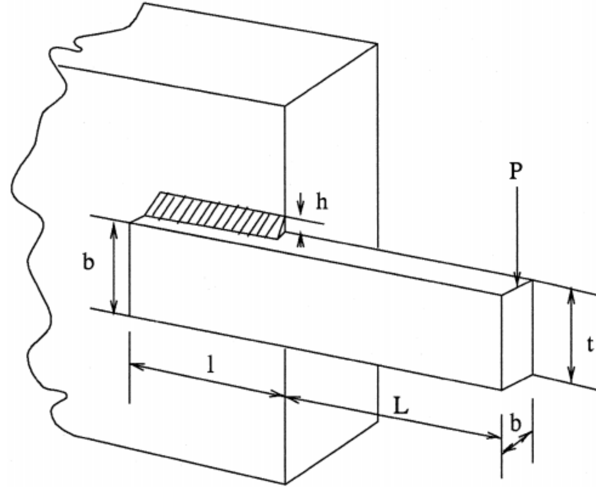


Figure 26 – Physical welded beam dimensions

The problem is mathematically formulated in Equations 49 to 67:

With $\vec{x} = [x_1, x_2, x_3, x_4] = [h, l, t, b]$ (49)

Minimise $f(\vec{x}) = 1.10471{x_1}^2 x_2 + 0.04811 x_3 x_4 (14 + x_2)$ (50)

Subject to $g_1(\vec{x}) = \tau(\vec{x}) - \tau_{max} \leq 0$ (51)

$g_2(\vec{x}) = \sigma(\vec{x}) - \sigma_{max} \leq 0$ (52)

$g_3(\vec{x}) = x_1 - x_4 \leq 0$ (53)

$g_4(\vec{x}) = 0.10471{x_1}^2 + 0.04811 x_3 x_4 (14 + x_2) - 5 \leq 0$ (54)

$g_5(\vec{x}) = 0.125 - x_1 \leq 0$ (55)

$g_6(\vec{x}) = \delta(\vec{x}) - \delta_{max} \leq 0$ (56)

$g_7(\vec{x}) = P - P_c(\vec{x}) \leq 0$ (57)

Where $\tau(\vec{x}) = \sqrt{(\tau')^2 + 2\tau'\tau''\frac{x_2}{2R} + (\tau'')^2}$ (58)

$\tau' = \frac{P}{\sqrt{2}x_2 x_2}, \tau'' = \frac{MR}{J}, M = P(L + \frac{x_2}{2})$ (59)

$R = \sqrt{\frac{{x_2}^2}{4} + \left(\frac{x_1 + x_3}{2}\right)^2}$ (60)

$J = 2\left(\sqrt{2}x_1 x_2 \left[\frac{{x_2}^2}{12} + \left(\frac{x_1 + x_3}{2}\right)^2\right]\right)$ (61)

$\sigma(\vec{x}) = \frac{6PL}{x_4 {x_3}^2}, \delta(\vec{x}) = \frac{4PL^3}{E x_4 {x_3}^3}$ (62)

$P_c = \frac{4.013E\sqrt{\frac{{x_3}^2 {x_4}^6}{36}}}{L^2}\left(1 - \frac{x_3}{2L}\sqrt{\frac{E}{4G}}\right)$ (63)

For $P = 6000\, lb, L = 14\, in, E = 30 \times 10^{-6}\, psi,$

$G = 12 \times 10^{-6}\, psi, \tau_{max} = 13600\, psi,$

$\sigma_{max} = 30000\, psi, \delta_{max} = 0.25\, in$

With variable ranges	$0.1 \le x_1 \le 2$	(64)
	$0.1 \le x_2 \le 10$	(65)
	$0.1 \le x_3 \le 10$	(66)
	$0.1 \le x_4 \le 2$	(67)

6.2.3 – Pressure Vessel Design

The pressure vessel design problem again aims to minimise the total manufacturing cost, including material, welding and forming costs. The problem is based on a pressure vessel with internal pressure capacity and volume requirements. Dimensions are again the variables, and the objective function is the total cost, which is subject to various constraints. The variables also have limits on their range of valid values. Figure 27 depicts the problem.

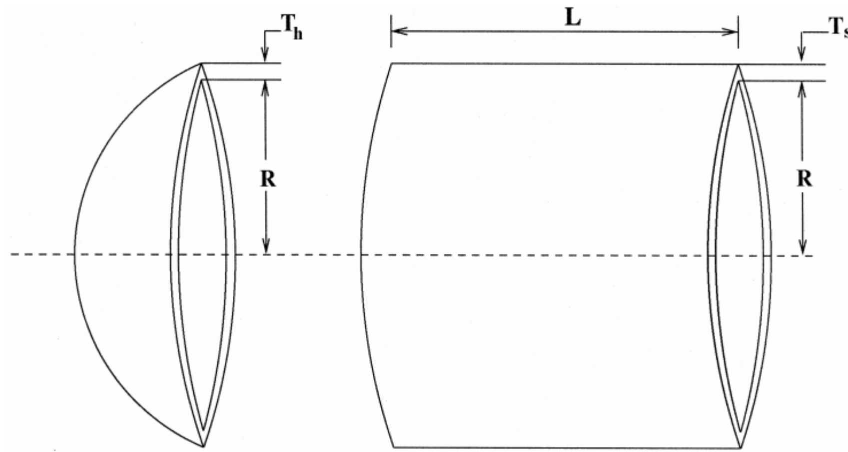


Figure 27 – Physical pressure vessel dimensions

The problem is mathematically formulated in Equations 68 to 77:

With	$\vec{x} = [x_1, x_2, x_3, x_4] = [T_s, T_h, R, L]$	(68)
Minimise	$f(\vec{x}) = 0.6224x_1x_3x_4 + 1.7781x_2x_3{}^2 + 3.1661x_1{}^2x_4 + 19.84x_1{}^2x_3$	(69)
Subject to	$g_1(\vec{x}) = -x_1 + 0.0193x_3 \le 0$	(70)
	$g_2(\vec{x}) = -x_2 + 0.00954x_3 \le 0$	(71)
	$g_3(\vec{x}) = -\pi x_3{}^2x_4 - \frac{4}{3}\pi x_3{}^3 + 1296000 \le 0$	(72)
	$g_4(\vec{x}) = x_4 - 240 \le 0$	(73)
With variable ranges	$1 \le x_1 \le 99$	(74)
	$1 \le x_2 \le 99$	(75)
	$10 \le x_3 \le 200$	(76)
	$10 \le x_4 \le 200$	(77)

6.2.4 – Constraint Handling Approach

Handling of the design problem constraints was necessary to ensure that the final solutions were valid. A penalty function approach was taken in order to effectively handle the constraint equations. This involved assigning a large multiplier to the amount of which a constraint was violated, and adding this to the objective function value. Equation 78 shows the approach used to handle a constraint function $g(\vec{x})$, where Δg is the numerical violation amount.

$$f(\vec{x}) = f(\vec{x}) + 10^{15} * (\Delta g)^2 \quad (78)$$

Therefore, any constraint violation, however minor, will make the solution worse than any other valid solution. This allows the swarm to move away from the invalid area and towards the valid domain space.

6.2.5 – Engineering Design Problem Settings

The engineering design problems have previously been solved by a number of researchers through different methods, including:

- Particle Swarm Optimization (He & Wang, 2007)
- Genetic Algorithm (Coello, 2000)
- Evolution Strategy (Mezura-Montes & Coello, 2008)
- Differential Evolution (Huang, Wang, & He, 2007)
- Harmony Search (Mahdavi, Fesanghary, & Damangir, 2007)
- African Wild Dog Algorithm (Subramanian et al., 2013)
- Grey Wolf Optimizer (Mirjalili et al., 2014)

As well as the accuracy of the final results, the efficiency of the algorithms was also important. Therefore, it was imperative to note the number of function evaluations (NFEs) used to obtain the reported minimums. Table 13 lists the total number of function evaluations used by researchers for the various algorithms and problems.

Table 13 – Total function evaluations used for solving the design problems

Algorithm	Spring Design	Beam Design	Pressure Vessel
PSO	200,000	200,000	200,000
GA	900,000	900,000	900,000
ES	25,000	25,000	25,000
DE	240,000	240,000	240,000
HS	50,000	300,000	200,000
AWDA	30,000	150,000	25,000
GWO	-	-	-

As Table 13 shows, there is a large variance amongst the total number of function evaluations used. In order to truly test the efficiency of SGA, it has been tested using the minimum number of function evaluations reported in the literature. This value is 25,000, as used by Mezura-Montes and Coello for testing Evolution Strategy. The authors of GWO negated to define the number of function evaluations used in their testing, making it hard to draw a comparison in efficiency.

The remaining parameter values are given in Table 14. The standard amongst researchers is for 30 runs to be undertaken to obtain the statistical results.

Table 14 – Design problem test settings

Parameter	Value
NFEs	25,000
$Colonies$	1
$Gliders$	5
SD	0.1
CP	5

6.2.6 – Engineering Design Problem Results

After running the algorithm, the results were analysed and compared to the existing literature. The raw data for the proof of results can be found in Appendix 10.

Table 15 and Table 16 outline the results obtained for the coil spring design problem.

Table 15 – Best minimum value results for the coil spring design problem

Rank	Algorithm	Optimum Variables			Optimum Weight
		d	D	N	
1	**SGA**	0.051659	0.356002	11.33104	0.0126652
2	AWDA	0.051655	0.355918	11.33603	0.0126653
3	GWO	0.051690	0.356737	11.28885	0.0126662
4	DE	0.051609	0.354714	11.41083	0.0126702
5	HS	0.051154	0.349871	12.07643	0.0126706
6	PSO	0.051728	0.357644	11.24454	0.0126747
7	ES	0.051643	0.355360	11.39792	0.0126980
8	GA	0.051480	0.351661	11.63220	0.0127047

Table 15 shows that SGA produced a better result than previously reported in the literature. The decrease in cost is small compared to the previously reported best value, however it does highlight the strength of SGA compared to the existing literature.

Table 16 – Statistical analysis of the results obtained for the coil spring design problem

Algorithm	Best	Mean	Worst	Std Dev
SGA	0.0126652	0.012898	0.015269	4.7E-05
AWDA	0.0126653	-	-	-
GWO	0.0126660	-	-	-
DE	0.0126702	0.012703	0.012790	2.7E-05
HS	0.0126706	-	-	-
PSO	0.0126747	0.012730	0.012924	5.2E-05
ES	0.0126980	0.013461	0.016485	9.7E-04
GA	0.0127047	0.012769	0.012822	3.9E-05

Table 16 shows the statistical analysis of the results for the first problem. It is important to note that all algorithms except ES used a greater number of function evaluations, meaning they should have a smaller spread across the best-to-worst range, and a smaller standard deviation. SGA bettered ES in every criterion, meaning that at an equal number of function evaluations, SGA outperforms it on this function. In particular, the standard deviation of SGA is a factor of 20 less than that of ES for the same number of function evaluations. DE used almost 10 times as many function evaluations and has a standard deviation less than a factor of 2 better than SGA.

Table 17 and Table 18 outline the results obtained for the welded beam design problem.

Table 17 – Best minimum value results for the welded beam design problem

Rank	Algorithm	Optimum Variables				Optimum Cost
		h	l	t	b	
1	HS	0.205730	3.47049	9.03662	0.205730	1.72480
2	AWDA	0.205729	3.47048	9.03662	0.205729	1.72485
3	**SGA**	0.205727	3.47054	9.03662	0.205729	1.72486
4	GWO	0.205676	3.47837	9.03681	0.205778	1.72624
5	PSO	0.202369	3.54421	9.04821	0.205723	1.72802
6	DE	0.203137	3.54299	9.03349	0.206179	1.73346
7	ES	0.199742	3.61206	9.03750	0.206082	1.73730
8	GA	0.208800	3.42050	8.99750	0.210000	1.74830

Table 17 shows that SGA ranks third amongst the existing literature for the minimum reported values. However, SGA used 92% fewer function evaluations than HS and produced a value that was only 0.0035% more costly. This highlights the efficiency of the algorithm at producing accurate results in a much lower number of function evaluations.

Table 18 – Statistical analysis of the results obtained for the welded beam design problem

Algorithm	Best	Mean	Worst	Std Dev
HS	1.72480	-	-	-
AWDA	1.72485	-	-	-
SGA	1.72486	1.729997	1.77763	0.01227
GWO	1.72624	-	-	-
PSO	1.72802	1.748831	1.782143	0.01292
DE	1.73346	1.768158	1.824105	0.02219
ES	1.73730	1.813290	1.994651	0.07050
GA	1.74830	1.771973	1.785835	0.01122

Table 18 shows the statistical analysis of the results for the second problem. Of the four other algorithms that have reported their statistical values (rather than just the minimum), SGA outperforms all algorithms at all criterion. The one exception is that GA has a slightly better standard deviation. However, this is expected as GA used 900,000 NFEs compared to just 25,000 of SGA. Again, an equal-NFE comparison to ES shows that SGA outperforms it by a fair margin.

Table 19 and Table 20 outline the results obtained for the pressure vessel design problem. The result obtained by GWO did not satisfy the requirement of T_s and T_h being integer multiples of 0.0625 inches, and as such, it has been omitted. Furthermore, HS breached the valid range of values for the length, and AWDA used different variable ranges.

Table 19 – Best minimum value results for the pressure vessel design problem

Rank	Algorithm	Optimum Variables				Optimum Cost
		T_s	T_h	R	L	
1	**SGA**	0.8125	0.4375	42.09844	176.6365	6059.7143
2	DE	0.8125	0.4375	42.09841	176.6376	6059.7340
3	ES	0.8125	0.4375	42.09808	176.6405	6059.7456
4	PSO	0.8125	0.4375	42.09126	176.7465	6061.0777
5	GA	0.8125	0.4375	40.32390	200.0000	6288.7445

Table 19 shows that, again, SGA has found a better value than previously reported in the literature.

Table 20 – Statistical analysis of the results obtained for the pressure vessel design problem

Algorithm	Best	Mean	Worst	Std Dev
SGA	6059.7143	6231.6808	7381.7174	282.37
DE	6059.7340	6085.2303	6371.0455	43.01
ES	6059.7456	6850.0049	7332.8799	426.00
PSO	6061.0777	6147.1332	6363.8041	86.45
GA	6288.7445	6293.8432	6308.1497	7.41

Table 20 shows the statistical analysis of the results for the third problem. An equal NFE comparison between SGA and ES shows that SGA has a much better mean value and a smaller standard deviation. It is observed that SGA has a higher standard deviation than most of the other algorithms. However, this is due to the much larger NFEs used by the algorithms that reduce the variability in the final result.

Along with the demonstrated efficiency that SGA has already displayed, it was also important that it provided good performance across a range of functions. Table 21 compares the cumulative ranks of the five algorithms that solved all three design tasks (in terms of the best reported values). Thus, the cumulative rank ignores the results of AWDA, HS and GWO (such that each algorithm was assigned a value between one and five).

Table 21 – Cumulative rank results

Algorithm	Cumulative Rank
SGA	3
DE	7
PSO	9
ES	11
GA	15

The cumulative rank gives an indication of each algorithms performance across the three functions. Table 21 shows that SGA gave the best overall performance amongst the five algorithms, in terms of the best reported values. This indicates the SGA displays strong performance across a range of objective functions; a key design goal.

6.2.7 – Convergence Analysis

Figure 28 gives the search history of the swarm for the welded beam design problem. Figure 28 (b) shows the move distance decreasing as the swarm moves from exploration to exploitation. This decreases the mean colony $f(x)$ value as the gliders converge towards the codominants. Because of the penalty function constraint handling approach, even a small constraint violation results in a severe increase in the objective function value. Each of the downward spikes in Figure 28 (c) is where no search agent is violating a constraint. It is observed that this becomes more prevalent in the later stages of the iteration history, meaning the swarm is converging toward an optimum in a valid domain space. This behaviour is ideal and indicates that the algorithm is performing to a high standard.

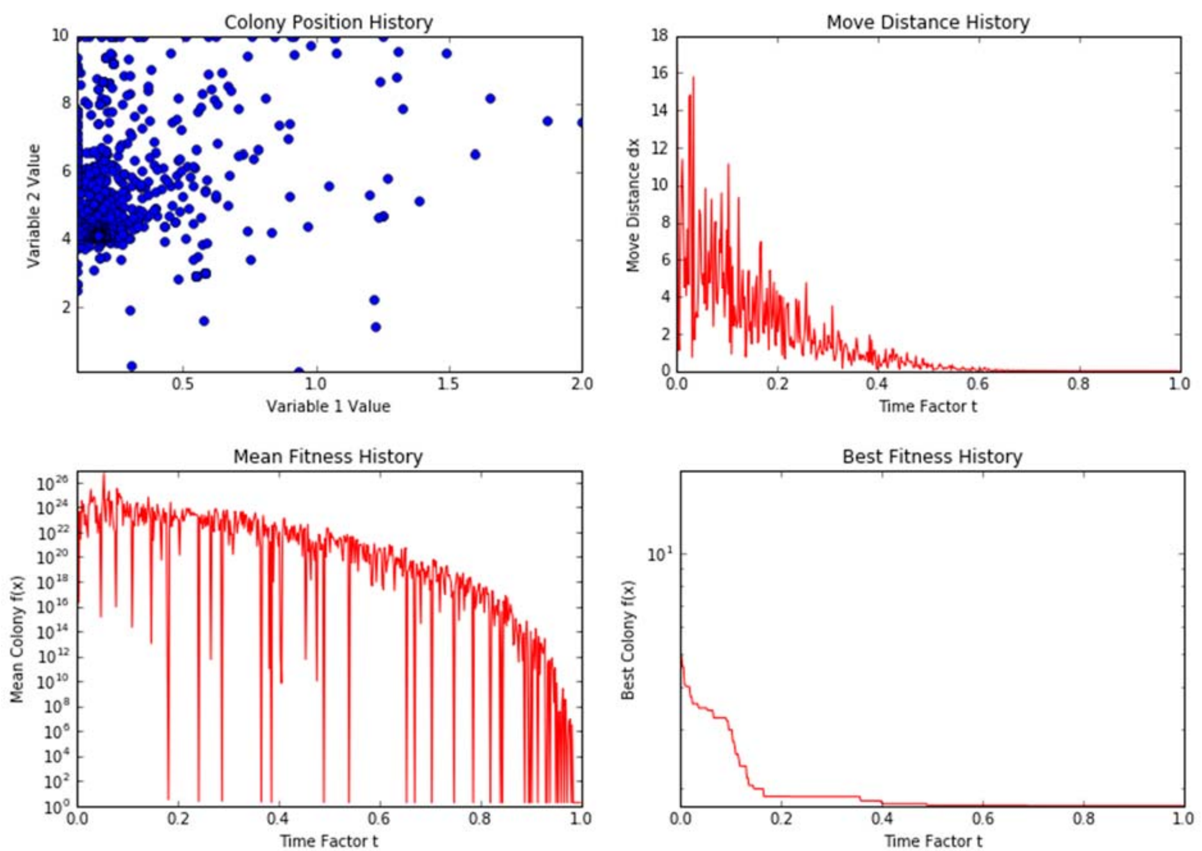


Figure 28 – Convergence behaviour analysis

CHAPTER 7

CASE STUDY: GEARBOX DESIGN

7.1 – INTRODUCTION TO THE CASE STUDY

In order to test the feasibility of using Sugar Glider Algorithm in real-world engineering design problems, a case study has been performed that was focused on designing a gearbox for a heliostat. Heliostats are large solar mirrors that reflect the sun's radiation towards a large central tower that is filled with molten salt. This molten salt is used to power a turbine that generates energy for homes and businesses. Concentrating Solar Thermal (CST) farms are a popular choice for renewable energy due to the fact that they are still able to generate electricity at night with the stored molten salt.

A design project has previously been completed that focused on designing a heliostat for prospective CST farms in Australia (MECH3100 Project Description, 2015). The project had two primary goals:

- To design the elevation and azimuthal gearboxes that controlled the movement of the heliostat, and
- To design the overall structure including tower, torque-tubes and the mirror-supporting frame.

The resulting heliostat design is shown in Figure 29.

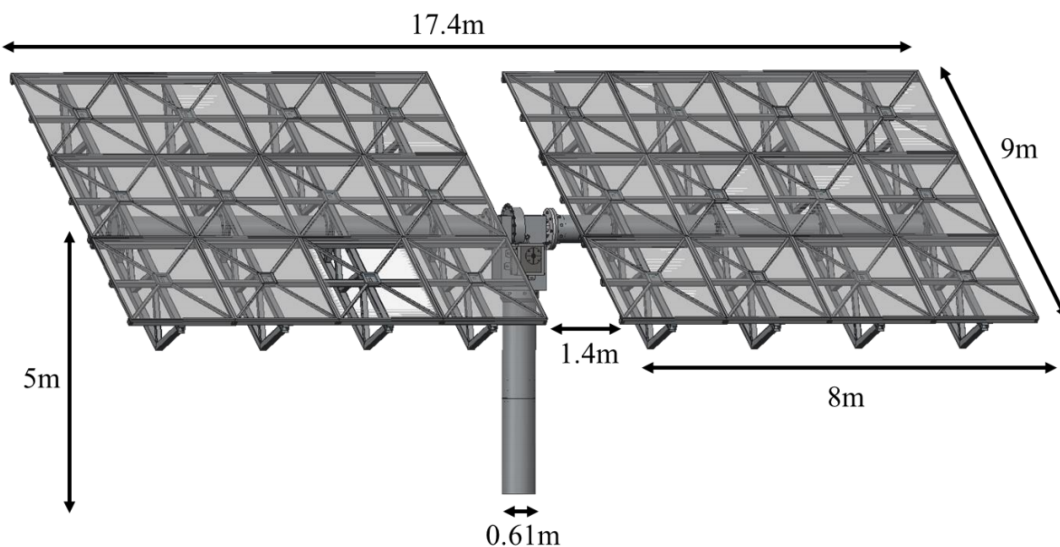


Figure 29 – Previously designed heliostat (MECH3100 Group 9, 2015)

As Figure 29 shows, heliostats are large structures that require equally-large torques to operate, especially in high winds. The design team calculated that the 60W motor needed to supply up to 45kNm of torque when performing the stowing operation in high wind. As such, a gearbox with a reduction factor of 9,570 was required.

The gearbox was a spur design that included 6 reductions, with a supplementary external worm gear reduction. The spur gearbox had a total reduction factor of 531, with the worm gear reducing by a further factor of 18. When designing the gearbox, close attention was paid to ensuring that the mirror pointing accuracy was not compromised by gear backlash or shaft deflections. However, there was no significant effort exerted in the optimisation of the gear train design.

In an effort to keep manufacturing cost and effort to a minimum, the same spur gears of 13 and 37 teeth were used throughout. All gears also featured the same module and face width (except the last reduction, which had a wider face). This resulted in most gears being well below their endurance limits with regards to the bending and contact stresses experienced. In summary, the design was safe and functional, yet heavy and wasted material. As such, it was decided that the gear train should be optimised using Sugar Glider Algorithm. This will further prove the applicability of SGA to practical engineering problems.

7.2 – METHODOLOGY

7.2.1 – Design Concept

In a typical gear train design, the known variables include:

- the input motor torque and speed,
- the output torque and speed (and thus, the total reduction),
- the desired type and number of reductions, and
- the gear material and its properties.

However, this still leaves a wealth of unknown information to contend with when beginning the design process. The main variables to select are the number of teeth per gear, module of each gear, and the face with of each gear (with the module and face width being the same through a reduction). As such, the number of variables to select is:

$$N_{variables} = 4 \times N_{reductions} \quad (79)$$

Even for smaller gear trains of few reductions, this number can become large and make the design task complex. Typical design methodology involves the use of a spreadsheet, where a single number is changed and its effect is observed. This is repeated until the total reduction

converges to the desired number, with all other constraints satisfied. This process is tedious and it is often hard to decide how to alter the input variables to achieve the desired outcome.

The concept behind using a bio-inspired algorithm to design the gearbox is that it will search the multitude of possible combinations and select the best one for the application. The objective in this case study was to minimise the total mass of the gear set, whilst still providing identical performance to the previously designed gearbox.

The optimisation process was applied only to the gear set itself, and not to the supplementary shafts or bearings. The shafts and bearings can be selected independently of the gears, and as such, they haven't been included in the optimisation task. They could, however, be added to the optimisation routine in future applications.

7.2.2 – Design Equations

There are a large number of equations that are used in order to design a functional gearbox. The equations listed in this section are applicable to spur gearboxes, with modifications required for gearboxes with helical, bevel, worm or planetary reductions. To facilitate the use of the algorithm, the design equations were converted into Python code form. All equations henceforth mentioned have been taken from Machine Component Design (Juvinall and Marshek, 2005). Table 22 lists the gearbox design parameters used and their definitions.

Table 22 – Design parameter definitions

Symbol	Definition	Units
m	Module of the gear	mm
b	Face width of the gear	mm
N	Number of teeth of the gear	-
n	Rotating speed of the gear	RPM
ϕ	Pressure angle of the gear, set to 20° for this case study	$degrees$
J	Geometry factor dependent on the number of teeth in contact at any time	-
K_v	Velocity factor, obtained from Equation 86	-
K_o	Overload factor, set to 1.5 in accordance with Table 23	-
K_m	Mounting factor, obtained from Table 24	-

The design equations are used to take information about the geometry and running conditions of the gear set, and produce the associated working stresses in the gears. Table 25 (located over the page) lists the equations utilised, as well as their use within the design process.

Table 23 provides the overload factor values. These factors reflect the degree of non-uniformity of the driving forces of the gearbox. The previous design team assumed that the source of power may experience light shock, and the driven machinery would likely experience moderate shock from winds.

Table 23 – Overload factor K_o selection matrix

Source of Power	**Driven Machinery**		
	Uniform	**Moderate Shock**	**Heavy Shock**
Uniform	1.00	1.25	1.75
Light Shock	1.25	1.50	2.00
Heavy Shock	1.50	1.75	2.25

The mounting factor in the stress equations reflects the accuracy of the gear alignment and varies with the face width of the mating gears. For the purpose of the case study, the support characteristics were assumed to be of the highest mounting accuracy category, as defined in Table 24. This table was linearly interpolated in the Python script to calculate the mounting factor value as the face widths changed.

Table 24 – Mounting factor K_m selection matrix

	Face Width (in.)			
Support Characteristics	**0 to 2**	**6**	**9**	**16 +**
Accurate mountings, small bearing clearances, minimum deflection, precision gears.	1.3	1.4	1.5	1.8
Less rigid mountings, less accurate gears, contact across the full face.	1.6	1.7	1.8	2.2
Accuracy and mounting such that less than full-face contact exists.	Over 2.2			

Table 25 – Gearbox design equations

Property	Equation		Units	Use
Gear Ratio	$R = \frac{d_{gear}}{d_{pinion}}$	(80)	-	Gives the reduction factor of a single gear mesh.
Pitch Circle Diameter	$d_p = mN$	(81)	mm	Computes the pitch circle diameter of the gear, used in other equations.
Pitch Line Velocity	$V = \frac{\pi}{12} d_p n$	(82)	ft/min	Finds the velocity at the pitch line (note that d_p must be in inches for this equation).
Face Width	$9m < b < 14m$	(83)	mm	Guideline for the face width as a function of module.
Diametral Pitch	$P = \frac{N}{d_p}$	(84)	$teeth$ $/inch$	Used to find the tangential force (note that d_p must be in inches for this equation).
Tangential Force	$F_t = \frac{P}{0.00508V}$	(85)	N	Computes the tangential force on a gear tooth. Used in the bending and contact stress equations.
Velocity Factor	$K_v = \frac{50 + \sqrt{V}}{50}$	(86)	-	Indicates the severity of impacts between successive pairs of mating teeth.
Lewis Bending Stress Equation	$\sigma_b = \frac{F_t}{mbJ} K_v K_o K_m$	(87)	MPa	Finds the maximum bending stress in the gear, to be compared with the fatigue limit.
Fatigue Geometry Factor	$I = \frac{\sin\phi\cos\phi}{2} \frac{R}{R+1}$	(88)	-	Geometry factor based on the tooth shape.
Hertzian Contact Stress Equation	$\sigma_H = C_p \sqrt{\frac{F_t}{bd_p I} K_v K_o K_m}$	(89)	MPa	Find the maximum contact stress on the gear surface, to be compared with the fatigue limit.
Contact Stress Fatigue Limit	$S_{fe} = 28 \times BHN - 69$	(90)	MPa	Gives the limit for the contact stress, as a function of the Brinell hardness.

7.2.3 – Optimisation Process

The gearbox design equations listed in the previous section were coded into Python and were used to create a function called `gearbox_eval`. This function takes the inputs of numbers of gear teeth, gear modules and face widths. The function then returns the weight of the gearbox plus any constraint violations as an added penalty. Figure 30 demonstrates the concept of the function.

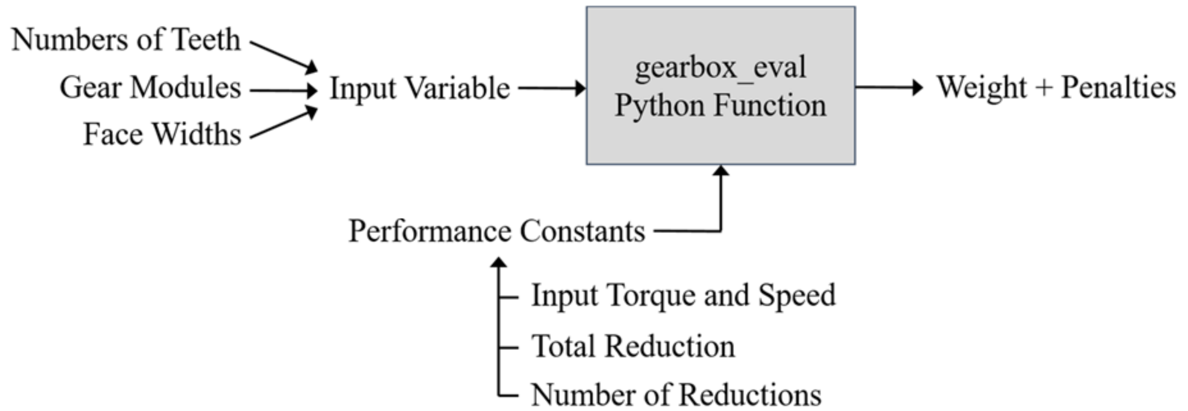


Figure 30 – Gearbox solver Python function concept

Since the objective of the optimisation exercise was to minimise total weight, the objective function of `gearbox_eval` was based on the total gear volume and the material density. Equation 91 details the objective function for the problem.

$$f(\vec{x}) = \rho \sum_{i=1}^{N} \frac{\pi}{4} d_{p,i} b_i \ [kg] \tag{91}$$

The constraint functions for `gearbox_eval` were based on the bending stress, contact stress and overall reduction ratio. The bending stresses in all gears was required to be less than the corrected bending fatigue limit, as detailed in Equation 92. Similarly, the contact stresses also needed to be less than the contact stress fatigue limit, which is a function of the material hardness as detailed in Equation 93. The final constraint was that the overall reduction ratio was required to be within 1% of the value reported by the initial design team. The motor input could then be adjusted very slightly to give identical gearbox outputs. The constraints are shown in Equations 92 to 94 below.

$$g_1(\vec{x}) = \sum_{i=1}^{N} \sigma_{b,i} - \sigma_{b,max} \tag{92}$$

$$g_2(\vec{x}) = \sum_{i=1}^{N} \sigma_{H,i} - \sigma_{H,max} \tag{93}$$

$$g_3(\vec{x}) = \left|\frac{R_{Total} - 531}{531}\right| - 0.01 \quad (94)$$

To produce a gearbox that functioned in an identical way to that which was previously designed, the same input conditions and total reduction were used. Furthermore, in order to give a fair comparison, the same number of reductions was also used. Table 26 outlines these values, as well as the material properties of Ferrium C61; the gear material previously selected by the design team (MECH3100 Group 9, 2015). Ferrium C61 has an extremely high hardness, meaning the contact stress fatigue limit was large by Equation 90 in Table 25.

Table 26 – Gearbox design parameters (MECH3100 Group 9, 2015)

Parameter	Value Used
Input Speed	60 RPM
Input Torque	9.41 Nm
Total Reduction	531
Number of Reductions	6
Bending Fatigue Limit	1156 MPa
Material Hardness	680 BHN

In order to increase simplicity in the calculation, gear losses have not been considered in the problem formulation, and have been removed from the calculations produced by the initial design team to ensure a fair comparison. Furthermore, in typical gearbox designs, the module and face width must be integer multiples. This standard was followed in the initial design, and as such was also implemented in the optimisation process. The Python script, named `gearbox_casestudy`, with the code formulation of the problem can be found in Appendix 8. The accuracy of this Python script was verified by inputting the values from the previous design and observing identical outcomes.

7.3 – Results

In accordance with the previous constrained engineering design problems, the algorithm parameters of Table 27 were used when running the optimisation tests.

Table 27 – SGA parameters for gearbox optimisation

Parameter	Value Used
CP	5
SD	0.1
Colony Size	5
Iterations Per Run	5000
No. of Runs	30

As stated previously, the design team who undertook the project produced a gear set that weighed 688kg. Performing the optimisation using Sugar Glider Algorithm yielded a minimum gear set weight of 189kg. Table 28 shows the full results of the optimisation problem.

Table 28 – Gearbox optimisation results

Performance Metric	Result (kg)
Old Design	688
SGA Minimum	189
SGA Average	221
SGA Maximum	277
SGA Std Dev	22

Table 28 indicates that the worst result output from SGA is still 60% lighter than the previously designed gear set, whilst the best result is 72% lighter. The standard deviation is also of a reasonable value, and this deviation would reduce for a greater number of iterations.

Table 29 – Optimised gearbox design layout

Gear Number	Number of Teeth	Module (mm)	Face Width Multiplier
Pinion #1	13	1	10
Gear #1	37		
Pinion #2	25	1	9
Gear #2	50		
Pinion #3	12	2	9
Gear #3	43		
Pinion #4	12	3	10
Gear #4	26		
Pinion #5	12	4	9
Gear #5	46		
Pinion #6	17	5	9
Gear #6	53		

The previous design team kept a constant module throughout their gearbox, resulting in unnecessarily large gears. Table 29 shows that the gearbox produced by the optimisation process followed a much more logical layout in that it increased the module as the reductions

progress. The stress in a gear is much more dependent on the torque rather than the speed, meaning that the initial reductions do not require large modules. Furthermore, it can be observed that the largest reductions in the gearbox are in the latter two stages. Leaving the larger reductions to the end reduces the stress in the preceding stages, reducing the volume of material required.

It can be observed that the new design is much more logical in its layout. However, it would have been a considerably harder task for the previous project team to be able to produce a design that is as efficient in its material usage. This displays the power of bio-inspired algorithms, and is a primary reason for their use. The results further prove the both the applicability and performance strength of Sugar Glider Algorithm with regard to engineering design problems.

7.4 – Case Study Outcomes

This case study has investigated the processes involved with applying bio-inspired algorithms to real-world engineering design problems. The case study was based on the design of a spur reduction gearbox; a project previously undertaken by a team of engineering students. The project was identified as a prime candidate for the application of bio-inspired algorithms to assist in the design process.

This case study has proven the application of Sugar Glider Algorithm to practical problems encountered by engineering designers. The gearbox that has been designed is of a more intuitive configuration with gear size that increased as the experienced stresses increased. The results obtained by the optimisation process saved a significant amount of weight compared to the previous design.

Had the previous design team had access to the code formulated in this case study, they would have benefitted greatly. Their design would be much more material efficient and the design time would have reduced significantly. Further checks would still need to be performed to ensure the proposed gearbox is feasible, however the optimisation does highlight the strength which BIAs have in solving engineering design tasks.

CHAPTER 8

FURTHER WORK

8.1 – IMPLEMENTATION OF SGA IN ANSYS

An opportunity exists for mechanical design to benefit greatly from the implementation of optimisation routines in computational Finite Element Analysis (FEA) programs. The potential benefits of utilising optimisation in FEA include:

- Reduced time for engineers to find the optimal design
- Designs that are safer without compromising in weight or cost
- Designs that are more efficient in their use of material (which is both weight and cost effective, and environmentally friendly)

ANSYS is an FEA program that enables users to solve complex structural engineering problems and make better, faster design decisions (ANSYS, 2016). ANSYS has inbuilt optimisation methods that aim to assist in designing mechanical components, however they are controlled largely by the user in their parameterisation of the design, and choice of optimisation methods (Bryce, 2015). This can often lead to inefficient optimisation processes or outcomes that are not actually optimal. As such, a method was developed by Bryce (2015) that allows interfacing between ANSYS and Python. This facilitates the implementation of automated, user-defined optimisation routines.

Implementation of SGA in ANSYS, through the use of the interface developed by Bryce (2015), will further increase the algorithm's value to the engineering community. It was initially planned that the interface would be used to further test the algorithm in the current work, however there were software compatibility issues encountered due to NumPy for IronPython no longer being supported. Time constraints meant that the interface issues were unable to be resolved, and as such, it is recommended that future work be undertaken to fix the issues and implement SGA in the interface.

8.2 – ALGORITHMIC IMPROVEMENTS

8.2.1 – Self-Adapting Improvements

An opportunity exists for a dynamic sight distance to be implemented in future versions of the algorithm. This would likely involve the distance decreasing as the iterations progressed, such

that as the gliders continuously found better positions, the distance they would attempt to move would decrease. This should further improve the results obtained from the codominant position updating method.

Furthermore, a dynamic convergence power should also improve the performance of the algorithm. Increasing the convergence power throughout the iterations would enhance the exploitation characteristics of the algorithm, at the cost of extensive exploration. However, if the algorithm was able to detect that it has already found the likely location of the global optimum, forcing convergence will increase the exploitation, leading to more accurate outcomes.

Implementing both of these improvements would mean that the algorithm would be self-adapting. Self-adapting algorithms change their parameters based on the results they observe throughout the iterations. For example, if the algorithm detects that it has become trapped in a local optimum the parameters can be altered in order to attempt to find a better solution somewhere else in the domain. Altering the algorithm to be self-adapting is not a trivial task, and would take significant work. However, it would almost guarantee a performance increase across all objective functions.

8.2.2 – Breeding Between Gliders

During the algorithm design process, some experimentation was undertaken in an attempt to introduce breeding between gliders. The idea was that breeding between two strong gliders (those with good fitness values) would result in a strong child. This is the basis behind all evolutionary algorithms.

The first experimentation involved taking two gliders and producing a third through mixing. This meant that some variable values were selected from one parent and the rest from the other (as shown in Figure 31). However, this was found to be very ineffective in low-dimension problems as there were a minimal number of different children that could be produced.

Parents	[25, 32, 12, 14]	[5, 58, 20, 10]
Possible Children	[25, 58, 20, 14]	[25, 32, 20, 10]

Figure 31 – Attempted simulated breeding through mixing

The second type of simulated breeding that was trialled involved taking two parents using their variable values to define ranges for selection. The child was then formed by choosing random

values from within the ranges (as shown in Figure 32). This method sometimes resulted in more fit children early in the iteration count, but rarely did so in the later stages. Therefore, it was not seen as a viable option for implementation.

Parents	[25, 32, 12, 14]	[5, 58, 20, 10]
Possible Children	[18, 40, 13, 11]	[20, 37, 18, 12]

Figure 32 – Attempted breeding through constrained random value selection

It was found that to make breeding a truly beneficial position updating method, some significant thought would need to be put into the idea. It would likely result in something similar to an evolutionary algorithm within the overall SGA algorithm. This would also likely mean the introduction of more control parameters, which could complicate the process of implementing the algorithm. One of the algorithm goals was simplicity, and for this reason, breeding between gliders didn't extend beyond some out-of-interest experimentation. However, this leaves the potential for future work to be undertaken in an attempt to implement breeding between gliders.

8.3 – Gearbox Design Solver Improvements

The gearbox design tool developed for the case study in Chapter 7 has been identified to be of a great potential value to the wider engineering community. As such, a more general version is planned to be developed in the future. This will allow engineers to input the operating conditions and gear material and set the BIA to design the gearbox on their behalf. The features of the general solver would include:

- A generalised format of the input variable to account for a different number of reductions,
- Addition of other supplementary gearbox design equations such as contact ratio and pressure angle,
- Inputs for general material properties, with calculations then performed to find the corrected endurance limits,
- An option for the BIA to also vary the number of reductions rather than have it constant,
- Potentially, support for the solving of gearboxes including planetary or helical, bevel and worm gear reductions.

8.4 – Implementation of SGA in Other Languages

Another improvement intended to be made for SGA is the translation of the Python code into other languages. Researchers implement metaheuristic algorithms in a variety of programming languages in order to suit the specific application at hand. Python is one of the most common choices, and as such it was selected for the current work. However, in order to ensure that SGA is available for a broad range of applications, it would be beneficial to have the source code available in a greater number of programming languages. The identified candidate languages include:

- Matlab
- Java
- Ruby
- Visual Basic

CONCLUSION

There are an ever-expanding range of applications for optimisation within engineering. Metaheuristics are a useful tool for solving optimisation tasks as they make no assumptions about the search space and can be used with 'black box' function types. Bio-inspired algorithms often offer a mix of both good performance and search space adaptability, meaning they can generally be effective at solving a broader range of problems. The adaptability is inherent in the design due to the way in which the biological organisms from which they are derived are able to adapt to their environment.

A novel bio-inspired algorithm has been presented that was based on the native Australian sugar glider, named Sugar Glider Algorithm (SGA). The algorithm has taken inspiration from the gliding and foraging behaviours, and the social hierarchy adopted by the animals. The algorithm has displayed strong performance in its preliminary testing stage. It set two new benchmark minimums for common engineering design problems, as well as being comparable in the statistical analyses performed. It has proven to be effective at finding optimal solutions at relatively low total function evaluation numbers. This enables shorter runtimes, compared to its competitors, in order to achieve equivalent outcomes. The combination of the algorithm simplicity, robustness, and strong performance ensures that it is to be a valued addition to the existing literature.

Once software compatibility issues are solved, it is expected that the algorithm will be implemented in the ANSYS interface developed by Bryce (2015). This will further increase its value to the engineering community and validate its performance. It is also intended that the work conducted will be published in the form of an article submitted to a respected journal within the field of metaheuristics. Suitable journals have been identified to include:

- Swarm Intelligence (Springer)
- Advances in Engineering Software (Elsevier)
- International Journal of Soft Computing (Medwell)

REFERENCES

Akbari, R. (2010). A multilevel evolutionary algorithm for optimizing numerical functions. International Journal of Industrial Engineering Computations, 2, 419-430.

ANSYS. (2016). Structural Analysis with ANSYS. Retrieved from http://www.ansys.com/Products/Structures

Baluja, S., & Caruana, R. (1995). Removing Genetics from the Standard Genetic Algorithm. Pittsburgh, PA.

Binitha, S., & Siva Sathya, S. (2012). A Survey of Bio-Inspired Optimisation Algorithms. International Journal of Soft Computing and Engineering, 2(2), 137-151.

Bryce, H. (2015). Finite Element Based Structural Optimization Techniques. (Bachelor of Engineering), The University of Queensland.

Deb, K., Amrit, P., Agarwal, S., & Meyarivan, T. (2002). A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. IEEE Transactions On Evolutionary Computation, 6(2), 182-197.

Dorigo, M., Maniezzo, V., & Colorni, A. (1991). Distributed Optimization by Ant Colonies. Paper presented at the European Conference on Artificial Life, Paris, France.

Gandomi, A. H., Yang, X.-S., & Alavi, A. H. (2011). Mixed variable structural optimization using Firefly Algorithm. Computers & Structures, 89(23–24), 2325-2336.

Gholizadeh, S., & Poorhoseini, H. (2015). Optimum design of steel frame structures by a modified Dolphin echolocation algorithm. Structural Engineering and Mechanics, 55(3), 535-554.

Glover, F., & McMillan, C. (1986). Applications of Integer ProgrammingThe general employee scheduling problem. An integration of MS and AI. Computers & Operations Research, 13(5), 563-573.

Hansen, N., & Ostermeier, A. (1996). Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. Paper presented at the Proceedings of the IEEE Conference on Evolutionary Computation.

Hendtlass, T. (2005). WoSP: A Multi-Optima Particle Swarm Algorithm. Paper presented at the IEEE Congress on Evolutionary Computation, Edinburgh, Scotland.

Holland, J. (1975). Genetic Algorithms and the Optimal Allocation of Trials. Ann Arbor.

Kannan, B. K., & Kramer, S. N. (1994). An Augmented Lagrange Multiplier Based Method for Mixed Integer Discrete Continuous Optimization and Its Applications to Mechanical Design. Journal of Mechanical Design, 116(2), 405-411.

Karaboga, D., & Akay, B. (2009). A comparative study of Artificial Bee Colony algorithm. Applied Mathematics and Computation, 214(1), 108-132.

Karaboga, D., & Basturk, B. (2007). A powerful and efficient algorithm for numerical function optimization: Artificial bee colony (ABC) algorithm. Journal of Global Optimization, 39(3), 459-471.

Kaveh, A., & Farhoudi, N. (2013). A new optimization method: Dolphin echolocation. Advances in Engineering Software, 59, 53-70.

Kaveh, A., & Farhoudi, N. (2015). Layout optimization of braced frames using differential evolution algorithm and dolphin echolocation optimization. Periodica Polytechnica: Civil Engineering, 59(3), 441-449.

Kaveh, A., & Farhoudi, N. (2016). Dolphin Echolocation Optimization for design of cantilever retaining walls. Asian Journal of Civil Engineering, 17(2), 193-211.

Kennedy, J., & Eberhart, R. (1995, Nov/Dec 1995). Particle swarm optimization. Paper presented at the Neural Networks, 1995. Proceedings., IEEE International Conference on.

Kirkpatrick, S., Gelatt Jr, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. Science, 220(4598), 671-680.

Klettenheimer, B. S., Temple-Smith, P. D., & Sofronidis, G. (1997). Father and son sugar gliders: more than a genetic coalition? Journal of Zoology, 242(4), 741-750.

Koza, J. R. (1990). Genetically breeding populations of computer programs to solve problems in artificial intelligence.

Kress, G., & Keller, D. (2007). Structural Optimization (pp. 165). Swiss Federal Institute of Technology, Zurich.

Krink, T., Filipic, B., & Fogel, G. B. (2004, 19-23 June 2004). Noisy optimization problems - a particular challenge for differential evolution? Paper presented at the Evolutionary Computation, 2004. CEC2004.

Lewis, S. (2012). Optimisation of a Scramjet Inlet using an Evolutionary Algorithm and the Barrine Cluster. (Bachelor of Engineering), The University of Queensland.

Lovbjerg, M., & Krink, T. (2002). Extending Particle Swarm Optimisers with Self-Organized Criticality. Aarhus, Denmark.

Mezura-Montes, E., & Coello, C. A. C. (2008). An empirical study about the usefulness of evolution strategies to solve constrained optimization problems. International Journal of General Systems, 37(4), 443-473.

Mirjalili, S., Mirjalili, S. M., & Lewis, A. (2014). Grey Wolf Optimizer. Advances in Engineering Software, 69, 46-61.

National Geographic. (2014). Deadly 60: 15 Deadly Animal Facts. Retrieved from http://animals.nationalgeographic.com/animals/wild/shows-deadly-60/fun-facts/

Omkar, S. N., Senthilnath, J., Khandelwal, R., Narayana Naik, G., & Gopalakrishnan, S. (2011). Artificial Bee Colony (ABC) for multi-objective design optimization of composite structures. Applied Soft Computing, 11(1), 489-499.

Polo-Corpa, M. J., Salcedo-Sanz, S., Pérez-Bellido, A. M., López-Espí, P., Benavente, R., & Pérez, E. (2009). Curve fitting using heuristics and bio-inspired optimization algorithms for experimental data processing in chemistry. Chemometrics and Intelligent Laboratory Systems, 96(1), 34-42.

Rudolph, G. (1999). Evolutionary Search under Partially Ordered Fitness Sets. Dortmund, Germany.

Storn, R., & Price, K. (1997). Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. Journal of Global Optimization, 11(4), 341-359.

Subramanian, C., Sekar, A., & Subramanian, K. (2013). A New Engineering Optimization Method: African Wild Dog Algorithm. International Journal of Soft Computing, 8(3), 163-170.

Technology, R. C. (2014). SHERPA – An Efficient and Robust Optimization/Search Algorithm. Michigan, USA.

Wolpert, D. H., & Macready, W. G. (1997). No free lunch theorems for optimization. IEEE Transactions On Evolutionary Computation, 1(1), 67-82.

Xie, X., Zhang, W., & Yang, Z. (2002). A Dissipative Particle Swarm Optimization. Paper presented at the Congress on Evolutionary Computation, Honolulu, USA.

Xinchao, Z. (2010). A perturbed particle swarm algorithm for numerical optimization. Applied Soft Computing, 10, 119-124.

Yang, X. S. (2009) Firefly algorithms for multimodal optimization. Vol. 5792 LNCS. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (pp. 169-178).

Yao, X., & Liu, Y. (1997). Fast evolution strategies. In P. J. Angeline, R. G. Reynolds, J. R. McDonnell, & R. Eberhart (Eds.), Evolutionary Programming VI: 6th International Conference, EP97 Indianapolis, Indiana, USA, April 13–16, 1997 Proceedings (pp. 149-161). Berlin, Heidelberg: Springer Berlin Heidelberg.

Young, E. (2016). What Wildlife Shows Don't Tell You About African Wild Dogs. Retrieved from http://phenomena.nationalgeographic.com/2016/03/29/what-wildlife-shows-dont-tell-you-about-african-wild-dogs/

Zielinski, K., & Laur, R. (2006). Parameter Adaptation for Differential Evolution with Design of Experiments. Paper presented at the 2nd IASTED International Conference on Computational Intelligence, San Francisco.

Zitzler, E., Deb, K., & Thiele, L. (1999). Comparison of Multiob jective Evolutionary Algorithms: Empirical Results. Zurich, Germany.

APPENDICES

CONTENTS

Appendix 1 – SGA Journal Article Draft……………………………………………………80
Appendix 2 – SGA Python Code……………………………………………………………101
Appendix 3 – SGA (Integer) Python Code…………………………………………………100
Appendix 4 – SGA (Plotting) Python Code…………………………………………………102
Appendix 5 – SGA (Pressure Vessel) Python Code…………………………………………105
Appendix 6 – Mathematical Function Tester Python Code…………………………………107
Appendix 7 – Engineering Problem Tester Python Code……………………………………109
Appendix 8 – Gearbox Evaluation Python Code……………………………………………112
Appendix 9 – Barebones Testing Script……………………………………………………115
Appendix 10 – Proof of Results……………………………………………………………...116

APPENDIX 1 – SGA JOURNAL ARTICLE DRAFT

The first appendix is a draft of the article that is intended to be submitted to a journal within the field of research. The article is contained in the pages that follow.

Sugar Glider Algorithm

Timothy Cassell, Dr Michael Heitzmann
The University of Queensland

Abstract

Bio-Inspired Algorithms (BIAs) are a class of metaheuristic that have proven to be effective at optimising a vast range of complex, black box function types. A new BIA is proposed that is based on a small, nocturnal gliding possum; the native Australian sugar glider (*Petaurus breviceps*). Sugar Glider Algorithm (SGA) imitates the leadership hierarchy and foraging behavior of a colony of gliders. Two co-dominant males lead a colony of five to seven gliders that forage for food by gliding between trees in search for insects or tree sap. The algorithm employs concurrent local exploitation (performed by the codominant males) and global exploration (performed by the remaining gliders). The performance of SGA has been quantitatively evaluated using five mathematical test functions, which are a mix of both unimodal and multimodal domains. The results are compared against Particle Swarm Optimisation, Differential Evolution and an Evolutionary Algorithm, with SGA performance amongst the best observed. Furthermore, SGA has been tested on three constrained engineering problems; coil spring design, welded beam design and pressure vessel design. SGA exhibited strong performance against seven existing algorithms, and found multiple new minimums than previously reported in literature. The results show that SGA is competitive against a wide range of existing algorithms in a variety of search domain topologies. These findings indicate that SGA is at the forefront of BIA performance and prove it is a superior candidate for the optimisation of engineering design problems.

1 Introduction

Optimization problems are prevalent in all facets of engineering. A general optimization task involves minimizing/maximizing an objective function via modification of variable values, whilst accounting for constraints. Mathematical optimization, through use of calculus, is often not a viable option for engineers as the objective function rarely takes a derivable algebraic form. The objective function can be presented in a complex mathematical form, via computational simulations, or even in terms of measurements obtained from real objects. For example, the objective of a car exterior design may be to minimize drag, where the drag is calculated via implementation of the model in a CFD program. In such cases, the only information known is the variable values and the resulting 'fitness' of the solution (how minimal the drag is). Thus, optimization methods that utilize only this information are required. Problems such as these are classified as black box functions.

Methods for finding optimal solutions to black box functions include metaheuristic search methods. Metaheuristics are algorithms that are problem-independent and make very few (often no) assumptions about the space being searched. Thus, they are widely applicable to a large range of optimization problems. Metaheuristics, however, do not guarantee that the global optimum will be found, as not every point in the search space is evaluated. Example metaheuristics include Simulated Annealing (Kirkpatrick, Gelatt Jr et al. 1983) and the human-memory inspired Tabu Search (Glover and McMillan 1986). In general, all metaheuristics share some common characteristics from which their robust performance is derived. Namely, these characteristics are simplicity, broad applicability and the ability to avoid local minima.

Simplicity is an important factor in a metaheuristic. It allows the algorithm to be implemented by people unfamiliar with the field of metaheuristics, increasing the possible user base. Simplicity often arises as a function of the simple processes upon which metaheuristics are frequently based. Inspiration is typically taken from areas such as animal behavior, evolutionary processes, or phenomena in nature, physics or chemistry.

Furthermore, metaheuristics are broadly applicable to a range of optimization tasks. They are problem-independent and don't usually require alteration between applications (apart from alterations necessary to facilitate different variable domain types; continuous and discreet). This arises as the metaheuristics make no assumptions about the space which they are searching. Due to this, gradient-based methods are avoided in favor of stochastic approaches (which utilize random operators).

Finally, metaheuristic algorithms are often able to avoid becoming trapped in the local optimums of a function. This is in part due to the random operators employed, but is also often a key design consideration for the algorithm creators. Methods are formulated specifically such that local minima convergence is unlikely (however, it is impossible to guarantee this). This feature further increases the strength of metaheuristics in the field of optimization routines.

Bio-Inspired Algorithms (BIAs) are a class of metaheuristic algorithm that utilize methods inspired by biological processes to solve optimization problems. BIAs have gained popularity as research topics due to their combination of fascinating inspiration sources and favorable performance outcomes. There are now algorithms inspired by a vast range of biological sources such as genetics, pack hunting of wolves, social behavior of bees and bird flocking. BIAs have been shown to be able offer a mix of both good performance and search space adaptability, meaning they can generally be effective at solving a broader range of problems. The adaptability is inherent in the design due to the way in which the biological organisms from which they are derived are able to adapt to their environment. For these reasons, algorithms that have taken inspiration from biology are a promising area of research within the field of metaheuristics.

The No Free Lunch Theorem provides both a motivation and an inherent design guideline for developing algorithms. The No Free Lunch theorem states that the performance of all search algorithms is the same when averaged over all possible objective functions (Wolpert and Macready 1997). That is, some algorithms perform exceedingly well with certain functions, but are inefficient with others. An implication of this theorem is that there is no single algorithm that offers the best performance across every objective function. Therefore, as the number of engineering applications that utilize optimization increases, as does the demand for the creation of new algorithms. This is one of the main driving forces behind the continual development of new optimization algorithms.

2 Literature Review

Bio-inspired algorithms can be split into two general classifications of Swarm Intelligence (SI) algorithms and Evolutionary Algorithms (EAs). Swarm Intelligence algorithms are based on the behavior of collectives of animals. More specifically, they take inspiration from the hunting, mating and movement behaviors, as well as social hierarchies. On the other hand, Evolutionary Algorithms are algorithms primarily inspired by both genetic operations and processes, and the Darwinian theory of Survival of the Fittest.

The most popular, and most widely known, SI algorithm is Particle Swarm Optimization (PSO). Proposed by Kennedy and Eberhart (1995), the algorithm simulates the generalized flocking behavior of birds and schooling behavior of fish. Search agents, referred to as particles, update their velocities based on both the swarm best known position, and their own best known position. The original version of the algorithm is very simple in its method, allowing people from all scientific fields to easily understand both the inspiration and the algorithm itself; a likely factor in its continued success.

Another popular SI algorithm is Ant Colony Optimization (ACO), proposed by Dorigo, Di Caro et al. (1999). ACO takes inspiration from the behavior of ants travelling between their colony and a food source. Ants lay down pheromone as they travel, increasing the tendency of other ants to travel along the same path. Thus, a self-reinforcing process ensues as ants travel along more optimal

paths. ACO, in its general form, is restricted to discrete optimization problems, limiting its possible applications. However, ACO has amassed popularity amongst the scientific community and is the most popular of the discrete-domain optimization routines.

Grey Wolf Optimizer (GWO) is a relatively new addition to SI algorithms, but has gained significant popularity since its inception. Mirjalili, Mirjalili et al. (2014) based the algorithm on the social hierarchy of grey wolves. The search agents are classified as either the alpha, beta, delta or omega wolves depending on their fitness values. Agents then update their position based on the relative positions of the pack leaders.

All Evolutionary Algorithms use some subset of the genetic operations of mutation, crossover, selection and recombination. The point of differentiation between algorithms is the way in which they implement these operations. Proposed by Holland (1975), Genetic Algorithm (GA) is the most well-known of the EAs. The algorithm aims to continuously improve the fitness of a population of solutions through the implementation of the aforementioned genetic operators.

Other EAs include Evolution Strategy (ES), which are a collection of closely-related algorithms that differ slightly in their selection and recombination technique, but all with the same general process (Rechenberg 1973). Differential Evolution (DE) is another population-based EA that was proposed by Storn and Price (1997). DE again has multiple algorithm variants that differ on mutation vector, the number of difference vectors, and the crossover scheme.

The apparent drawback of EAs is the often-large number of control parameters that must be selected. For example, DE first requires the selection of three parameters to define the algorithm methods, with a further three required to define the runtime parameter values. When information about the search domain topology is unknown, it can then be hard to select proper parameter values.

There are a range of other metaheuristic algorithms available, all of which vary in both their inspiration sources and demonstrated performance. However, there is currently no published algorithm based on the stimulating behavior of the sugar glider. Sugar gliders were identified as a promising inspiration source for a new bio-inspired SI algorithm. As such, attempts have been made to produce an SI optimization routine that both mimics the behavior of the sugar glider, and provides exceptional performance in engineering optimization tasks.

3 Sugar Glider Algorithm

3.1 Inspiration Source

Sugar gliders (*Petaurus breviceps*) are a small native Australian flying possum that are members of the marsupial infraclass. As their name suggests, sugar gliders are able to glide large distances between trees due to the existence of a gliding membrane (called a patagium) that extends from their forelegs to

their hindlegs. This allows them to glide for up to 50m in a single flight; a fascinating behavior that it is relatively unique amongst animals.

One of the interesting characteristics of sugar gliders is their hierarchical social structure. Klettenheimer, Temple-Smith et al. (1997) observed that there are two codominant males that lead the colony, with other males being suppressed. These two males cooperated with each other in activities such as grooming and fighting, but never cooperated with any subordinate males. These males are referred to as the codominant gliders and they are the decision makers within the group, leading the continual search for food.

Sugar gliders feed on insects, as well as supplementary nectars such as acacia gum and eucalyptus sap when the bugs are scarce. All the food sources for a glider are contained in the trees that they glide between. Thus, the motivation behind gliding is largely related to sustenance, indicating that the behavior is purposeful and directed, rather than a random practice.

Another fascinating trait of the gliders is their den-swapping behavior that has been observed in the wild. Sugar gliders have been found to simultaneously inhabit up to 13 dens, moving between the different locations as they desire (Lindenmayer, 2002). Gliders search for food in the areas surrounding their den, implying that the location of their current den impacts and directs their search for food.

3.2 Introduction to Sugar Glider Algorithm

The basis of SGA is the simulated search for food by a colony of gliders. Analogous to a real colony, the gliders are split into two groups; the codominant gliders and the subordinate gliders. The codominant gliders lead the search of the domain, with the subordinates updating their position based on the codominants' positions.

A glider's fitness is represented as the available food as its location. Gliders then move from tree-to-tree in search of the most-abundant food source. The glide distance decreases as the iterations progress, as gliders continuously find better food sources and thus do not need to fly as far.

To simulate the den position influencing the gliders' search for food, the subordinate gliders also update their position based on a randomly-generated home position (randomized at each iteration and shared between all gliders). This introduces a level of variability to the movement of the search agents, which increases the explorative characteristics of the algorithm.

The codominants are in charge of the colony, and thus it is critical that they are the strongest gliders of the group. Therefore, the codominant gliders are taken to be the search agents with the two best current positions. They update their position by performing a local search through a small fluctuation of variable values (introducing concurrent exploitation throughout all iterations). The codominants lead the search, implying they must make informed choices about their movement. Thus, the codominants only move if the new position is of a

better fitness. An advantage of this method is that the best known position is always carried forward throughout the generations.

The pseudocode for SGA is given in Figure 1 below.

$Objective\ function\ f(x), x = (x_1, \dots, x_d)$
$Generate\ initial\ population\ of\ gliders$
$Evaluate\ fitness\ of\ each\ glider$
$\boldsymbol{WHILE}\ i < iteration\ max$:
    $Rank\ colony\ and\ set\ two\ codominant\ gliders$
    $\boldsymbol{FOR}\ each\ codominant$:
        $Perform\ local\ search$
        $\boldsymbol{IF}\ fitness\ new < fitness\ current$:
            $Move\ to\ new\ position$
    $\boldsymbol{FOR}\ each\ subordinate\ glider$:
        $Update\ position\ based\ on\ codominants\ and\ home\ den$
    $i = i + 1$
$\boldsymbol{RETURN}\ best\ fitness, best\ position$

Figure 1 – SGA pseudocode

3.3 Algorithm Description

The algorithm starts by initialising the glider colony through assigning variable values, which are randomly selected from the user-defined ranges. The values are stored in a position matrix $\hat{P}$, with n rows (for n gliders) and d columns (for d dimensions).

$$\hat{P} = \begin{pmatrix} x_{1,1} & \cdots & x_{1,d} \\ \vdots & \ddots & \vdots \\ x_{n,1} & \cdots & x_{n,d} \end{pmatrix} \tag{1}$$

The fitness of the colony is then evaluated through calculation of the objective function value for each glider. These values are stored in a fitness vector $\vec{F}$.

$$\vec{F} = \begin{pmatrix} f_1 \\ \vdots \\ f_n \end{pmatrix} \tag{2}$$

The algorithm then enters the main loop that iterates until the maximum number of iterations has been reached. First, the colony is ranked in order of best fitness, meaning that the first two rows of the position matrix become the two codominant gliders.

$$\hat{P} = \begin{pmatrix} x_{c1,1} & \cdots & x_{c1,d} \\ x_{c2,2} & \ddots & x_{c2,d} \\ x_{n,1} & \cdots & x_{n,d} \end{pmatrix} \tag{3}$$

The codominant gliders then search for a move by observing a sighted position (sp) through use of a sight distance (SD) parameter (default at 0.1), according to Equations 4 through 6:

$$SD = (0, 1) \tag{4}$$

$$rand = random\big(1 - SD(1 + t), 1 + SD(1 - t)\big) \tag{5}$$

$$sp = current * rand \tag{6}$$

The variable t is the time factor, with linear range $[0 \rightarrow 1]$, and is given through:

$$t = \frac{iteration_{current}}{iteration_{max}} \tag{7}$$

The objective function is then evaluated for the sighted position. If the fitness at the new location is better, a move is performed. Otherwise, the codominant stays in its current position.

Next, the subordinate gliders' positions are updated. Three random vectors assist to increase the variability and prolong the convergence:

$$r1 = random(t, 2 - t) \tag{8}$$

$$r2 = random(t, 2 - t) \tag{9}$$

$$r3 = random(-2, 2) \tag{10}$$

The distance to move is then calculated by the addition of distances to the two codominants and the home den position:

$$CP = [1, 10] \tag{11}$$

$$\begin{aligned} dx &= r1 * (codom_1 - current) \\ &+ r2 * (codom_2 - current) * t^{\frac{1}{CP}} \\ &+ r3 * (home - current) * (1 - t)^{CP} \end{aligned} \tag{12}$$

The time factor (t) and a convergence power parameter (CP) act on both the home and second codominant distances such that a weighting toward the codominant distance increases throughout the iterations. The convergence power is default at 5, but is suggested to be altered depending on the domain topology (see Section 3.4).

Towards the later stages, dx will simply direct the gliders towards the codominants. The codominants will ideally, by this stage, be positioned in the same local area. Therefore, it is clear that:

$$\lim_{t \to 1} dx = 2 * (codom - current) \tag{13}$$

Therefore, the move distance must be halved in order to ensure proper convergence of the swarm in the later stages:

$$new = current + 0.5 * dx \tag{14}$$

The $\hat{P}$ matrix is then updated with the new glider positions. The fitness values for the new subordinate glider positions are then evaluated and the $\vec{F}$ matrix is updated. The codominant and subordinate gliders then continuously update their positon until the maximum number of iterations has been reached.

3.4 Parameter Selection

There are three parameters that must be defined for SGA, outlined in Table 1.

Table 1 – Valid parameter ranges

Parameter	Description	Valid Values
$Gliders$	The number of colony gliders.	$[3, \infty]$
SD	The sight distance for the codominant gliders.	$[0, 1]$
CP	The convergence power of the swarm.	$[1, 10]$

The first two parameters are the colony size and the sight distance SD. These values are default at 5 and 0.1 respectively. The value for SD is not able to be chosen intuitively and requires . Therefore, it is recommended the default value be used unless problem specific tuning is undertaken. The value of 0.1 has shown good performance on across a range of function types. Additionally, a colony of five gliders has given the best performance across a range of function types (for equal numbers of function evaluations). It is recommended that, for increased accuracy, the number of iterations is increased rather than the colony size.

The convergence power CP is the only parameter which is recommended to be altered by the user. The convergence power directly influences how quickly the swarm converges. This allows the user to customize the convergence rate depending on the domain type of the function being optimized. Figure 2 gives the weighting plots for various values of CP.

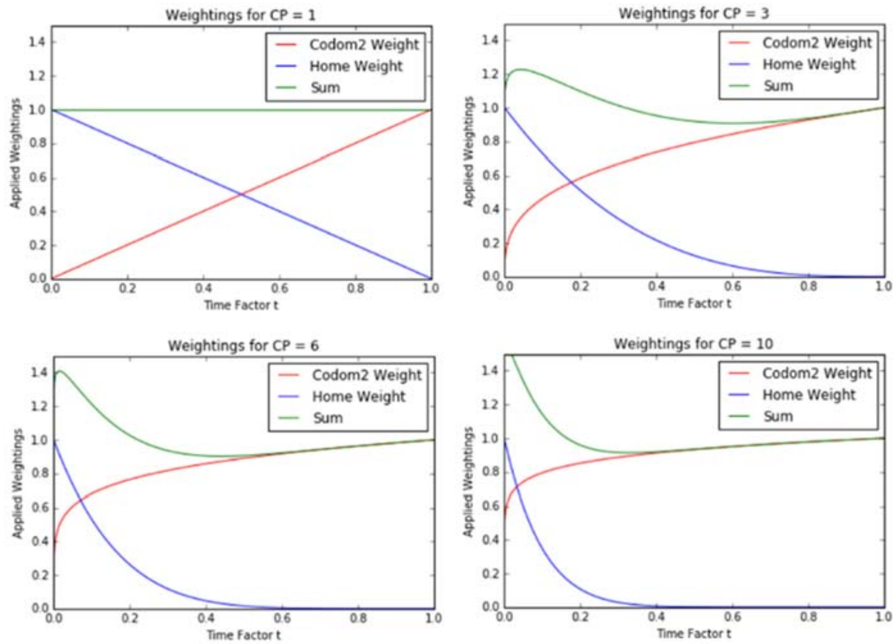


Figure 2 – Effect of CP value on distance weightings

For unimodal functions across lower numbers of dimensions, a high convergence power can be used in order to increase the optimum exploitation. For multimodal functions across higher numbers of dimensions, a low convergence power should be used in an attempt to guarantee to find the global optimum location. Figure 3 gives an example of the effect of the CP parameter, tested on a unimodal function.

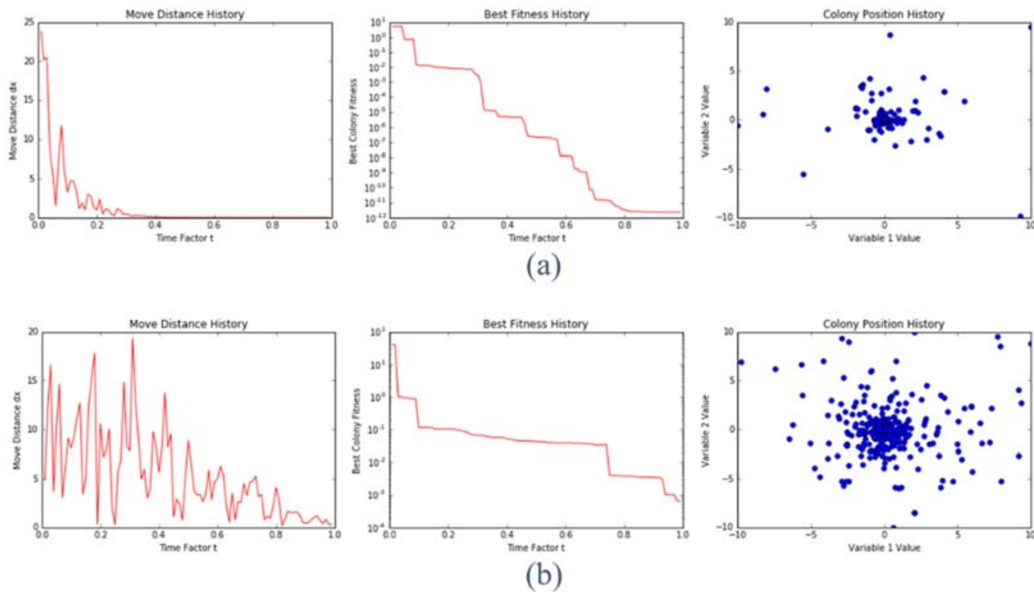


Figure 3 – Convergence for CP = 10 (a) and CP = 1 (b)

4 Mathematical Function Benchmarking

4.1 Test Functions

The performance of the algorithm has been benchmarked using five mathematical benchmark functions. The test functions that have been used are a mix of unimodal and multimodal, and separable and non-separable. Table 2 outlines the definitions and characteristics of the test functions used.

Table 2 – Mathematical test functions

Name	Definition	Type	Range	Dim
Sphere (F1)	$\sum x_i^2$	US	[-100, 100]	5
Schaffer (F2)	$0.5 + \frac{\sin^2\left(\sqrt{x_1^2 + x_2^2}\right) - 0.5}{\left(1 + 0.001(x_1^2 + x_2^2)\right)^2}$	MN	[-100, 100]	2
Griewank (F3)	$1 + \frac{1}{4000}\sum (x_i - 100)^2 - \prod \cos\left(\frac{x_i - 100}{\sqrt{i}}\right)$	MN	[-600, 600]	50
Rastrigin (F4)	$\sum x_i - 10\cos(2\pi x_i) + 10$	MS	[-5.12, 5.12]	50
Rosenbrock (F5)	$\sum 100\left(x_{i+1} - x_i^2\right)^2 + (x_i + 1)^2$	UN	[-50, 50]	50

4.2 Algorithmic Settings

The algorithm's performance has been compared to that of three existing bio-inspired algorithms. The results, and therefore the algorithmic settings, for DE, PSO and EA have been taken from Krink, Filipic et al. (2004). To ensure a fair comparison, the same number of function evaluations as the previous study have been used. The parameter settings are listed in Table 3.

Table 3 – Algorithmic settings

Parameter	Value
NFE (F1, F2)	100,000
NFE (F3, F4, F5)	500,000
No. of Runs	30
$Gliders$	5
CP	5
SD	0.1

4.3 Benchmarking Results

Table 4 shows the results for the test functions used. The average function value and the standard deviation across 30 runs are presented. All functions are minimisation problems with a global optimum value of zero. Values below E-12 have been presented as zero.

Table 4 – Mathematical benchmarking results

Function		DE	PSO	EA	SGA
Sphere	Mean	0	2.51E-08	0	0
	Std Dev	0	0	0	0
Schaffer	Mean	0	0.00453	0	0
	Std Dev	0	0.00090	0	0
Griewank	Mean	0	1.549	0.00624	0
	Std Dev	0	0.06695	0.00138	0
Rastrigin	Mean	0	13.1162	32.6679	261.842
	Std Dev	0	1.44815	1.94017	32.114
Rosenbrock	Mean	35.3176	5142.45	79.818	39.1265
	Std Dev	0.2744	2929.47	10.4477	0.19824

The results show that SGA is competitive with the existing BIAs. It achieved the global minimum in 3 of 5 functions, whereas EA achieved 2 of 5 and PSO didn't find the global optimum for any function. DE outperforms SGA on the last two functions, however the difference in the Rosenbrock function is minor. SGA outperformed PSO and EA on all functions apart from Rastrigin. The SGA result for the Rastrigin function is the worst compared to the other three algorithms. This is likely due to the highly multimodal nature of the function. As such, performance would likely improve if a lower convergence power was used.

5 Engineering Design Problems

5.1 Problem Definitions

The performance of the algorithm has been evaluated on three semi-real constrained engineering design problems. These problems are common in the literature and have previously been solved by various methods.

5.1.1 Coil Spring Design

The objective of the coil spring design problem is to minimise the total mass via alteration of the spring dimensions. The design is subject to constraints on deflection, shear stress and surge frequency that limit the feasible space. The variables also have limits on their range of valid values.

The problem is mathematically formulated as follows:

With $\vec{x} = [x_1, x_2, x_3] = [d, D, N]$

Minimise	$f(\vec{x}) = (x_3 + 2)x_2 x_1^2$	(15)
Subject to	$g_1(\vec{x}) = 1 - \frac{x_2^3 x_3}{71785 x_1^4} \le 0$	
	$g_2(\vec{x}) = \frac{4x_2^2 - x_1 x_2}{12566(x_2 x_1^3 - x_1^4)} + \frac{1}{5108 x_1^2} - 1 \le 0$	
	$g_3(\vec{x}) = 1 - \frac{140.45 x_1}{x_2^2 x_3} \le 0$	
	$g_4(\vec{x}) = \frac{x_1 + x_2}{1.5} - 1 \le 0$	
With variable ranges	$0.05 \le x_1 \le 2$	
	$0.25 \le x_2 \le 1.3$	
	$2 \le x_3 \le 15$	

5.1.2 Welded Beam Design

The welded beam design problem attempts to minimize the total material and fabrication cost of a beam that is loaded in bending. Beam dimensions are varied to reduce the total mass (thus reducing material cost). However, the cost of welding is also considered, introducing more complexity to the problem. The objective function is the total cost, and is minimized subject to constraints on shear and bending stresses, buckling loads and end deflection. The variables also have limits on their range of valid values.

The problem is mathematically formulated as follows:

With	$\vec{x} = [x_1, x_2, x_3, x_4] = [h, l, t, b]$	
Minimise	$f(\vec{x}) = 1.10471 x_1^2 x_2 + 0.04811 x_3 x_4 (14 + x_2)$	(16)
Subject to	$g_1(\vec{x}) = \tau(\vec{x}) - \tau_{max} \le 0$	
	$g_2(\vec{x}) = \sigma(\vec{x}) - \sigma_{max} \le 0$	
	$g_3(\vec{x}) = x_1 - x_4 \le 0$	
	$g_4(\vec{x}) = 0.10471 x_1^2 + 0.04811 x_3 x_4 (14 + x_2) - 5 \le 0$	
	$g_5(\vec{x}) = 0.125 - x_1 \le 0$	
	$g_6(\vec{x}) = \delta(\vec{x}) - \delta_{max} \le 0$	
	$g_7(\vec{x}) = P - P_c(\vec{x}) \le 0$	
Where	$\tau(\vec{x}) = \sqrt{(\tau')^2 + 2\tau'\tau'' \frac{x_2}{2R} + (\tau'')^2}$	
	$\tau' = \frac{P}{\sqrt{2} x_2 x_2}, \tau'' = \frac{MR}{J}, M = P(L + \frac{x_2}{2})$	
	$R = \sqrt{\frac{x_2^2}{4} + \left(\frac{x_1 + x_3}{2}\right)^2}$	
	$J = 2\left(\sqrt{2} x_1 x_2 \left[\frac{x_2^2}{12} + \left(\frac{x_1 + x_3}{2}\right)^2\right]\right)$	
	$\sigma(\vec{x}) = \frac{6PL}{x_4 x_3^2}, \delta(\vec{x}) = \frac{4PL^3}{E x_4 x_3^3}$	
	$P_c = \frac{4.013E\sqrt{\frac{x_3^2 x_4^6}{36}}}{L^2}\left(1 - \frac{x_3}{2L}\sqrt{\frac{E}{4G}}\right)$	

For	$P = 6000\ lb, L = 14\ in, E = 30 \times 10^{-6}\ psi,$ $G = 12 \times 10^{-6}\ psi, \tau_{max} = 13600\ psi,$ $\sigma_{max} = 30000\ psi, \delta_{max} = 0.25\ in$
With variable ranges	$0.1 \le x_1 \le 2$ $0.1 \le x_2 \le 10$ $0.1 \le x_3 \le 10$ $0.1 \le x_4 \le 2$

5.1.3 Pressure Vessel Design

The pressure vessel design problem again aims to minimise the total manufacturing cost, including material, welding and forming costs. The problem is based on a pressure vessel with internal pressure capacity and volume requirements. Dimensions are again the variables, and the objective function is the total cost, which is subject to various constraints. The variables also have limits on their range of valid values.

The problem is mathematically formulated as follows:

With	$\vec{x} = [x_1, x_2, x_3, x_4] = [T_s, T_h, R, L]$	
Minimise	$f(\vec{x}) = 0.6224x_1x_3x_4 + 1.7781x_2{x_3}^2 + 3.1661{x_1}^2x_4 + 19.84{x_1}^2x_3$	(17)
Subject to	$g_1(\vec{x}) = -x_1 + 0.0193x_3 \le 0$ $g_2(\vec{x}) = -x_2 + 0.00954x_3 \le 0$ $g_3(\vec{x}) = -\pi{x_3}^2x_4 - \frac{4}{3}\pi{x_3}^3 + 1296000 \le 0$ $g_4(\vec{x}) = x_4 - 240 \le 0$	
With variable ranges	$1 \le x_1 \le 99$ $1 \le x_2 \le 99$ $10 \le x_3 \le 200$ $10 \le x_4 \le 200$	

5.2 Algorithmic Settings

The engineering design problems have previously been solved through a number of different methods, including:

- Genetic Algorithm (Coello, 2000)
- Differential Evolution (Huang, Wang, & He, 2007)
- Harmony Search (Mahdavi, Fesanghary, & Damangir, 2007)
- Particle Swarm Optimization (He & Wang, 2007)
- Evolution Strategy (Mezura-Montes & Coello, 2008)
- African Wild Dog Algorithm (Subramanian et al., 2013)
- Grey Wolf Optimizer (Mirjalili et al., 2014)

As well as the accuracy of the final results, the efficiency of the algorithms was also important. Therefore, it was imperative to note the number of function evaluations (NFEs) used to obtain the reported minimums. Table 5 lists the total number of function evaluations used to solve the problems.

Table 5 – Number of function evaluations for the engineering design problems

Algorithm	Spring Design	Beam Design	Pressure Vessel
PSO	200,000	200,000	200,000
GA	900,000	900,000	900,000
ES	25,000	25,000	25,000
DE	240,000	240,000	240,000
HS	50,000	300,000	200,000
AWDA	30,000	150,000	25,000

As Table 5 shows, there is a large variance amongst the total number of function evaluations used. In order to truly examine the efficiency of SGA, it has been tested using the minimum number of function evaluations reported in the literature of 25,000. The authors of GWO negated to report the number of function evaluations used in their testing, making it hard to draw a comparison in efficiency.

The remaining algorithmic parameter values are given in Table 6.

Table 6 – Algorithmic settings

Parameter	Value
No. of Runs	30
Iterations	5,000
$Gliders$	5
SD	0.1
CP	5

5.3 Design Problem Results

Table 7 and Table 8 outline the results obtained for the coil spring design problem.

Table 7 – Best minimum results for the coil spring problem

Rank	Algorithm	Optimum Variables			Optimum Weight
		d	D	N	
1	SGA	0.051659	0.356002	11.33104	0.0126652
2	AWDA	0.051655	0.355918	11.33603	0.0126653
3	GWO	0.051690	0.356737	11.28885	0.0126662
4	DE	0.051609	0.354714	11.41083	0.0126702
5	HS	0.051154	0.349871	12.07643	0.0126706
6	PSO	0.051728	0.357644	11.24454	0.0126747
7	ES	0.051643	0.355360	11.39792	0.0126980
8	GA	0.051480	0.351661	11.63220	0.0127047

Table 7 shows that SGA produced a better result than previously reported in the literature. This is in spite of SGA only using 25,000 function evaluations; much less than most other algorithms.

Table 8 – Statistical results for the coil spring problem

Algorithm	Best	Mean	Worst	Std Dev
SGA	0.0126652	0.012898	0.015269	4.7E-05
AWDA	0.0126653	-	-	-
GWO	0.0126660	-	-	-
DE	0.0126702	0.012703	0.012790	2.7E-05
HS	0.0126706	-	-	-
PSO	0.0126747	0.012730	0.012924	5.2E-05
ES	0.0126980	0.013461	0.016485	9.7E-04
GA	0.0127047	0.012769	0.012822	3.9E-05

Table 8 shows the statistical analysis of the results for the first problem. It is important to note that all algorithms except ES used a greater number of function evaluations, meaning they should have a smaller spread across the best-to-worst range, and a smaller standard deviation. At an equal number of function evaluations, SGA outperformed ES in every criterion.

Table 9 and Table 10 outline the results obtained for the welded beam design problem.

Table 9 – Best minimum results for the welded beam problem

Rank	Algorithm	Optimum Variables				Optimum Cost
		h	l	t	b	
1	HS	0.205730	3.47049	9.03662	0.205730	1.72480
2	AWDA	0.205729	3.47048	9.03662	0.205729	1.72485
3	SGA	0.205727	3.47054	9.03662	0.205729	1.72486
4	GWO	0.205676	3.47837	9.03681	0.205778	1.72624
5	PSO	0.202369	3.54421	9.04821	0.205723	1.72802
6	DE	0.203137	3.54299	9.03349	0.206179	1.73346
7	ES	0.199742	3.61206	9.03750	0.206082	1.73730
8	GA	0.208800	3.42050	8.99750	0.210000	1.74830

Table 9 shows that SGA ranks third amongst the existing literature for the minimum reported values. However, SGA used 92% fewer function evaluations than HS for a comparable result. This highlights the efficiency of the algorithm at producing accurate results in a much lower number of function evaluations.

Table 10 – Statistical results for the welded beam problem

Algorithm	Best	Mean	Worst	Std Dev
HS	1.72480	-	-	-
AWDA	1.72485	-	-	-
SGA	1.72486	1.729997	1.77763	0.01227
GWO	1.72624	-	-	-
PSO	1.72802	1.748831	1.782143	0.01292
DE	1.73346	1.768158	1.824105	0.02219
ES	1.73730	1.813290	1.994651	0.07050
GA	1.74830	1.771973	1.785835	0.01122

Table 10 shows the statistical analysis of the results for the second problem. Of the four other algorithms that reported statistical values, SGA outperforms all algorithms at all criterion. The one exception is that GA has a slightly better standard deviation. However, this is expected as GA used 36 times as many function evaluations as SGA. Again, for the same number of function evaluations, SGA outperformed ES in every criterion.

Table 11 and Table 12 outline the results obtained for the pressure vessel design problem. The result obtained by GWO did not satisfy the requirement of T_s and T_h being integer multiples of 0.0625 inches, and as such, it has been omitted. Furthermore, HS breached the valid range of values for the length, and AWDA used different variable ranges.

Table 11 – Best minimum results for the pressure vessel problem

Rank	Algorithm	Optimum Variables				Optimum Cost
		T_s	T_h	R	L	
1	SGA	0.8125	0.4375	42.09844	176.6365	6059.7143
2	DE	0.8125	0.4375	42.09841	176.6376	6059.7340
3	ES	0.8125	0.4375	42.09808	176.6405	6059.7456
4	PSO	0.8125	0.4375	42.09126	176.7465	6061.0777
5	GA	0.8125	0.4375	40.32390	200.0000	6288.7445

Table 11 shows that, again, SGA has found a better value than previously reported in the literature, whilst satisfying all constraints.

Table 12 – Statistical results for the pressure vessel problem

Algorithm	Best	Mean	Worst	Std Dev
SGA	6059.7143	6231.6808	7381.7174	282.37
DE	6059.7340	6085.2303	6371.0455	43.01
ES	6059.7456	6850.0049	7332.8799	426.00
PSO	6061.0777	6147.1332	6363.8041	86.45
GA	6288.7445	6293.8432	6308.1497	7.41

Table 12 shows the statistical analysis of the results for the third problem. It is observed that SGA has a higher standard deviation than most of the other

algorithms. However, this is due to the much larger number of function evaluations used, which reduce the variability in the final result. An equal comparison between SGA and ES shows that SGA has a much better mean value and a smaller standard deviation.

6 Conclusion

A novel bio-inspired algorithm has been presented that was based on the native Australian sugar glider, named Sugar Glider Algorithm (SGA). The algorithm has taken inspiration from the gliding and foraging behaviors, and the social hierarchy adopted by the animals. The algorithm has displayed strong performance in its preliminary testing stage. It set two new benchmark minimums for common engineering design problems, as well as being comparable in the statistical analyses performed. It has proven to be effective at finding optimal solutions at relatively low total function evaluation numbers. This enables shorter runtimes for achieving equivalent outcomes. The combination of the algorithm simplicity, robustness, and strong performance ensure that it is a valuable addition to the existing literature.

7 References

Dorigo, M., et al. (1999). "Ant algorithms for discrete optimization." Artificial Life 5(2): 137-172.

Glover, F. and C. McMillan (1986). "Applications of Integer ProgrammingThe general employee scheduling problem. An integration of MS and AI." Computers & Operations Research 13(5): 563-573.

Holland, J. (1975). Genetic Algorithms and the Optimal Allocation of Trials. U. o. Michigan. Ann Arbor: 88-104.

Kennedy, J. and R. Eberhart (1995). Particle swarm optimization. Neural Networks, 1995. Proceedings., IEEE International Conference on.

Kirkpatrick, S., et al. (1983). "Optimization by simulated annealing." Science 220(4598): 671-680.

Klettenheimer, B. S., et al. (1997). "Father and son sugar gliders: more than a genetic coalition?" Journal of Zoology 242(4): 741-750.

Krink, T., et al. (2004). Noisy optimization problems - a particular challenge for differential evolution? Evolutionary Computation, 2004. CEC2004. Congress on.

Mirjalili, S., et al. (2014). "Grey Wolf Optimizer." Advances in Engineering Software 69: 46-61.

Rechenberg, I. (1973). Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution, Frommann-Holzboog.

Storn, R. and K. Price (1997). "Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces." Journal of Global Optimization 11(4): 341-359.

Wolpert, D. H. and W. G. Macready (1997). "No free lunch theorems for optimization." IEEE Transactions On Evolutionary Computation 1(1): 67-82.

APPENDIX 2 – SGA PYTHON CODE

```
# -*- coding: utf-8 -*-
"""
Created on Tue Oct 25 11:00:00 2016

@author: Tim Cassell
@email: tcassell@uq.edu.au

Python 2.7 based code formulation of Sugar Glider Algorithm. Applicable to
continuous domains only. A mixed-variable version is currently in development.
"""

from __future__ import division
from random import uniform
import numpy as np
from scipy.stats import rankdata


def sga(objfunc, lb, ub, itermax=1000, colonies=1, gliders=5, cp=5, sd=0.1, guess=False):
    #---- Assertion Checks and Variable Initialisation -----
    assert len(lb) == len(ub)
    assert (itermax > 1 and type(itermax) == int)
    assert (cp >= 0 and cp <= 100)
    assert (sd > 0 and sd < 1)
    assert (gliders >= 3 and type(gliders) == int)
    if guess: assert (type(guess) == list or type(guess) == np.ndarray)
    dim = len(ub)
    count = 1

    glider_array = np.zeros([colonies, gliders, dim])
    fitness_array = np.zeros([colonies, gliders])

    #----- Colony Initialisation -----
    if not guess:
        for i in range(0, colonies):
            for j in range(0, gliders):
                for k in range (0, dim):
                    glider_array[i, j, k] = uniform(lb[k], ub[k])
                fitness_array[i, j] = objfunc(glider_array[i, j])
    else:
        for i in range(0, colonies):
            glider_array[i, 0] = guess
            fitness_array[i, 0] = objfunc(glider_array[i, 0])
            for j in range(1, gliders):
                for k in range (0, dim):
                    glider_array[i, j, k] = uniform(lb[k], ub[k])
                fitness_array[i, j] = objfunc(glider_array[i, j])

    #----- Main Loop Begin -----
    while count < itermax:
        t = count/itermax
        #----- Ranking of Colonies -----
        for i in range(0, colonies):
            ranks = rankdata(fitness_array[i], method='ordinal')
            gliders_copy = np.copy(glider_array[i])
            fitness_copy = np.copy(fitness_array[i, :])
            k = 0
            for j in ranks:
                glider_array[i, j-1] = gliders_copy[k]
                fitness_array[i, j-1] = fitness_copy[k]
                k += 1
```

Continued over page.

```
        #----- Position Updating -----
        for i in range(0, colonies):
            home = np.ones(dim)
            for j in range(dim):
                home[j] = uniform(lb[j], ub[j])
            for j in range(0, gliders):
                if j == 0 or j == 1:
                    rand = np.ones(dim)
                    for k in range(dim):
                        rand[k] = uniform(1 - sd*(1 + t), 1 + sd*(1 - t))
                    current = np.copy(glider_array[i, j])
                    sighted_pos = current*rand
                    sighted_pos = np.clip(sighted_pos, lb, ub)
                    sighted_fit = objfunc(sighted_pos)
                    if sighted_fit < fitness_array[i, j]:
                        fitness_array[i, j] = sighted_fit
                        glider_array[i, j] = np.copy(sighted_pos)
                else:
                    r1 = np.ones(dim)
                    r2 = np.ones(dim)
                    r3 = np.ones(dim)
                    for k in range(len(r1)):
                        r1[k] = uniform(t, 2-t)
                        r2[k] = uniform(t, 2-t)
                        r3[k] = uniform(-2, 2)
                    current = np.copy(glider_array[i, j])
                    dx = (np.copy(glider_array[i, 0]) - current)*r1 + \
                         (np.copy(glider_array[i, 1]) - current)*r2*t**(1/cp)  + \
                         (np.copy(home) - current)*r3*(1-t)**cp
                    glider_array[i, j] = current + 0.5*dx
                    glider_array[i, j] = np.clip(glider_array[i, j], lb, ub)
                    fitness_array[i, j] = objfunc(glider_array[i, j])

        count += 1
    #----- Main Loop End -----

    #---- Setting Best Values -----
    best_fit = 1e100
    for i in range(0, colonies):
        if np.amin(fitness_array[i]) < best_fit:
            best_fit = np.amin(fitness_array[i])
            best_pos = list(glider_array[i, np.argmin(fitness_array[i])])

    return best_fit, best_pos


if __name__ == '__main__':

    def fun(arg):
        dim = len(arg)
        lst = []
        for i in range(0, dim):
            x = arg[i]
            lst.append(x**2)
        return sum(lst)

    out = sga(fun, [-10]*30, [10]*30, cp=5)

    print 'Best fitness is: ', out[0]
    print 'Best position is:', out[1]
```

APPENDIX 3 – SGA (INTEGER) PYTHON CODE

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Oct 25 11:00:00 2016

@author: Tim Cassell
@email: tcassell@uq.edu.au

Python 2.7 based code formulation of Sugar Glider Algorithm that works with
variables that all take integer values. This was used in the gearbox case
study.
"""

from __future__ import division
from random import uniform
import numpy as np
from scipy.stats import rankdata


def rnd(var):
    new_var = np.zeros([len(var)])
    for i in range(len(var)):
        new_var[i] = round(var[i])
    return new_var


def sga_int(objfunc, lb, ub, itermax=1000, colonies=1, gliders=5, cp=5, sd=0.1, guess=False):
    #---- Assertion Checks and Variable Initialisation -----
    assert len(lb) == len(ub)
    assert (itermax > 1 and type(itermax) == int)
    assert (cp >= 0 and cp <= 100)
    assert (sd > 0 and sd < 1)
    assert (gliders >= 3 and type(gliders) == int)
    if guess: assert (type(guess) == list or type(guess) == np.ndarray)
    dim = len(ub)
    count = 1

    glider_array = np.zeros([colonies, gliders, dim])
    fitness_array = np.zeros([colonies, gliders])

    #----- Colony Initialisation -----
    if not guess:
        for i in range(0, colonies):
            for j in range(0, gliders):
                for k in range (0, dim):
                    glider_array[i, j, k] = uniform(lb[k], ub[k])
                glider_array[i, j] = rnd(glider_array[i, j])
                fitness_array[i, j] = objfunc(glider_array[i, j])
    else:
        for i in range(0, colonies):
            glider_array[i, 0] = guess
            fitness_array[i, 0] = objfunc(glider_array[i, 0])
            for j in range(1, gliders):
                for k in range (0, dim):
                    glider_array[i, j, k] = uniform(lb[k], ub[k])
                glider_array[i, j] = rnd(glider_array[i, j])
                fitness_array[i, j] = objfunc(glider_array[i, j])

    #----- Main Loop Begin -----
    while count < itermax:
        t = count/itermax
        #----- Ranking of Colonies -----
        for i in range(0, colonies):
            ranks = rankdata(fitness_array[i], method='ordinal')
            gliders_copy = np.copy(glider_array[i])
            fitness_copy = np.copy(fitness_array[i, :])
            k = 0
            for j in ranks:
                glider_array[i, j-1] = gliders_copy[k]
                fitness_array[i, j-1] = fitness_copy[k]
                k += 1
```

Continued over page.

```
    #----- Position Updating -----
    for i in range(0, colonies):
        home = np.ones(dim)
        for j in range(dim):
            home[j] = uniform(lb[j], ub[j])
        for j in range(0, gliders):
            if j == 0 or j == 1:
                rand = np.ones(dim)
                for k in range(dim):
                    rand[k] = uniform(1 - sd*(1 + t), 1 + sd*(1 - t))
                current = np.copy(glider_array[i, j])
                sighted_pos = current*rand
                sighted_pos = np.clip(sighted_pos, lb, ub)
                sighted_pos = rnd(sighted_pos)
                sighted_fit = objfunc(sighted_pos)
                if sighted_fit < fitness_array[i, j]:
                    fitness_array[i, j] = sighted_fit
                    glider_array[i, j] = np.copy(sighted_pos)
            else:
                r1 = np.ones(dim)
                r2 = np.ones(dim)
                r3 = np.ones(dim)
                for k in range(len(r1)):
                    r1[k] = uniform(t, 2-t)
                    r2[k] = uniform(t, 2-t)
                    r3[k] = uniform(-2, 2)
                current = np.copy(glider_array[i, j])
                dx = (np.copy(glider_array[i, 0]) - current)*r1 + \
                     (np.copy(glider_array[i, 1]) - current)*r2*t**(1/cp)  + \
                     (np.copy(home) - current)*r3*(1-t)**cp
                glider_array[i, j] = current + 0.5*dx
                glider_array[i, j] = np.clip(glider_array[i, j], lb, ub)
                glider_array[i, j] = rnd(glider_array[i, j])
                fitness_array[i, j] = objfunc(glider_array[i, j])

    count += 1
#----- Main Loop End -----

#---- Setting Best Values -----
best_fit = 1e100
for i in range(0, colonies):
    if np.amin(fitness_array[i]) < best_fit:
        best_fit = np.amin(fitness_array[i])
        best_pos = list(glider_array[i, np.argmin(fitness_array[i])])

return best_fit, best_pos
```

APPENDIX 4 – SGA (PLOTTING) PYTHON CODE

```
# -*- coding: utf-8 -*-
"""
Created on Fri Sep 2 11:35:00 2016

@author: Tim Cassell
"""

from __future__ import division
from random import uniform
import numpy as np
from pylab import plot, semilogy, axis, title, xlabel, ylabel, show
from scipy.stats import rankdata


def sga_plot(objfunc, lb, ub, itermax=100, colonies=1, gliders=5, cp=5, sd=0.1, guess=False):
    #---- Assertion Checks and Variable Initialisation -----
    assert len(lb) == len(ub)
    assert (itermax > 1 and type(itermax) == int)
    assert (cp >= 0 and cp <= 100)
    assert (sd > 0 and sd < 1)
    assert (gliders >= 3 and type(gliders) == int)
    if guess: assert (type(guess) == list or type(guess) == np.ndarray)
    dim = len(ub)
    count = 1
    iter_curve, cvg_best_curve, cvg_mean_curve, dx_curve = [], [], [], []
    x_list, y_list = [], []

    glider_array = np.zeros([colonies, gliders, dim])
    fitness_array = np.zeros([colonies, gliders])

    #----- Colony Initialisation -----
    if not guess:
        for i in range(0, colonies):
            for j in range(0, gliders):
                for k in range (0, dim):
                    glider_array[i, j, k] = uniform(lb[k], ub[k])
                fitness_array[i, j] = objfunc(glider_array[i, j])
    else:
        for i in range(0, colonies):
            glider_array[i, 0] = guess
            fitness_array[i, 0] = objfunc(glider_array[i, 0])
            for j in range(1, gliders):
                for k in range (0, dim):
                    glider_array[i, j, k] = uniform(lb[k], ub[k])
                fitness_array[i, j] = objfunc(glider_array[i, j])

    #----- Main Loop Begin -----
    while count < itermax:
        t = count/itermax
        #----- Ranking of Colonies -----
        for i in range(0, colonies):
            ranks = rankdata(fitness_array[i], method='ordinal')
            gliders_copy = np.copy(glider_array[i])
            fitness_copy = np.copy(fitness_array[i, :])
            k = 0
            for j in ranks:
                glider_array[i, j-1] = gliders_copy[k]
                fitness_array[i, j-1] = fitness_copy[k]
                k += 1
```

Continued over page.

```
    #----- Position Updating -----
    for i in range(0, colonies):
        home = np.ones(dim)
        for j in range(dim):
            home[j] = uniform(lb[j], ub[j])
        for j in range(0, gliders):
            x_list.append(glider_array[i, j, 0])
            y_list.append(glider_array[i, j, 1])
            if j == 0 or j == 1:
                rand = np.ones(dim)
                for k in range(dim):
                    rand[k] = uniform(1 - sd*(1 + t), 1 + sd*(1 - t))
                current = np.copy(glider_array[i, j])
                sighted_pos = current*rand
                sighted_pos = np.clip(sighted_pos, lb, ub)
                sighted_fit = objfunc(sighted_pos)
                if sighted_fit < fitness_array[i, j]:
                    fitness_array[i, j] = sighted_fit
                    glider_array[i, j] = np.copy(sighted_pos)
            else:
                r1 = np.ones(dim)
                r2 = np.ones(dim)
                r3 = np.ones(dim)
                for k in range(len(r1)):
                    r1[k] = uniform(t, 2-t)
                    r2[k] = uniform(t, 2-t)
                    r3[k] = uniform(-2, 2)
                current = np.copy(glider_array[i, j])
                dx = (np.copy(glider_array[i, 0]) - current)*r1 + \
                     (np.copy(glider_array[i, 1]) - current)*r2*t**(1/cp)  + \
                     (np.copy(home) - current)*r3*(1-t)**cp
                glider_array[i, j] = current + 0.5*dx
                glider_array[i, j] = np.clip(glider_array[i, j], lb, ub)
                fitness_array[i, j] = objfunc(glider_array[i, j])
    #----- Plotting List Updatings -----
                if i == 0 and j == 3:
                    dx_curve.append(np.linalg.norm(dx))

    cvg_best_curve.append(np.min(fitness_array[0]))
    cvg_mean_curve.append(np.average(fitness_array[0]))
    iter_curve.append(t)



    count += 1
#----- Main Loop End -----

#---- Setting Best Values -----
best_fit = 1e100
for i in range(0, colonies):
    if np.amin(fitness_array[i]) < best_fit:
        best_fit = np.amin(fitness_array[i])
        best_pos = list(glider_array[i, np.argmin(fitness_array[i])])

return best_fit, best_pos, iter_curve, cvg_best_curve, \
        cvg_mean_curve, dx_curve, x_list, y_list
```

Continued over page.

```
if __name__ == '__main__':

    def fun(arg):
        dim = len(arg)
        lst = []
        for i in range(0, dim):
            x = arg[i]
            lst.append(x**2)
        return sum(lst)

    out = sga_plot(fun, [-10]*2, [10]*2, cp=5)

    print 'Best fitness is: ', out[0]
    print 'Best position is:', out[1]

    plot(out[2], out[4], 'r-')
    xlabel('Time Factor t')
    ylabel('Move Distance dx')
    title('Move Distance History')
    show(plot)

    semilogy(out[2], out[3], 'r-')
    xlabel('Time Factor t')
    ylabel('Best Colony Fitness')
    title('Best Fitness History')
    show(plot)

    semilogy(out[2], out[4], 'r-')
    xlabel('Time Factor t')
    ylabel('Mean Colony Fitness')
    title('Mean Fitness History')
    show(plot)

    plot(out[6], out[7], 'bo')
    xlabel('Variable 1 Value')
    ylabel('Variable 2 Value')
    title('Colony Position History')
    axis([-10, 10, -10, 10])
    show(plot)
```

APPENDIX 5 – SGA (PRESSURE VESSEL) PYTHON CODE

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Oct 25 11:00:00 2016

@author: Tim Cassell
@email: tcassell@uq.edu.au

Python 2.7 based code formulation of Sugar Glider Algorithm that has been
customised for use with the mixed variable pressure vessel design problem.
"""

from __future__ import division
from random import uniform
import numpy as np
from scipy.stats import rankdata


def rnd(var):
    new_var = [1e100, 1e100]
    for i in range(0, 2):
        num = var[i]
        a = num - (num%0.0625)
        b = a + 0.0625 - (a%0.0625)
        if (b-num) <= (num-a):
            new_var[i] = b
        else:
            new_var[i] = a
    return new_var


def sga_pv(objfunc, lb, ub, itermax=1000, colonies=1, gliders=5, cp=5, sd=0.1, guess=False):
    #---- Assertion Checks and Variable Initialisation -----
    assert len(lb) == len(ub)
    assert (itermax > 1 and type(itermax) == int)
    assert (cp >= 0 and cp <= 100)
    assert (sd > 0 and sd < 1)
    assert (gliders >= 3 and type(gliders) == int)
    if guess: assert (type(guess) == list or type(guess) == np.ndarray)
    dim = len(ub)
    count = 1

    glider_array = np.zeros([colonies, gliders, dim])
    fitness_array = np.zeros([colonies, gliders])

    #----- Colony Initialisation -----
    if not guess:
        for i in range(0, colonies):
            for j in range(0, gliders):
                for k in range (0, dim):
                    glider_array[i, j, k] = uniform(lb[k], ub[k])
                glider_array[i, j, 0], glider_array[i, j, 1] = rnd(glider_array[i, j, 0:2])
                fitness_array[i, j] = objfunc(glider_array[i, j])
    else:
        for i in range(0, colonies):
            glider_array[i, 0] = guess
            fitness_array[i, 0] = objfunc(glider_array[i, 0])
            for j in range(1, gliders):
                for k in range (0, dim):
                    glider_array[i, j, k] = uniform(lb[k], ub[k])
                glider_array[i, j, 0], glider_array[i, j, 1] = rnd(glider_array[i, j, 0:2])
                fitness_array[i, j] = objfunc(glider_array[i, j])

    #----- Main Loop Begin -----
    while count < itermax:
        t = count/itermax
```

Continued over page.

```
    #----- Ranking of Colonies -----
    for i in range(0, colonies):
        ranks = rankdata(fitness_array[i], method='ordinal')
        gliders_copy = np.copy(glider_array[i])
        fitness_copy = np.copy(fitness_array[i, :])
        k = 0
        for j in ranks:
            glider_array[i, j-1] = gliders_copy[k]
            fitness_array[i, j-1] = fitness_copy[k]
            k += 1
    #----- Position Updating -----
    for i in range(0, colonies):
        home = np.ones(dim)
        for j in range(dim):
            home[j] = uniform(lb[j], ub[j])
        for j in range(0, gliders):
            if j == 0 or j == 1:
                rand = np.ones(dim)
                for k in range(dim):
                    rand[k] = uniform(1 - sd*(1 + t), 1 + sd*(1 - t))
                current = np.copy(glider_array[i, j])
                sighted_pos = current*rand
                sighted_pos = np.clip(sighted_pos, lb, ub)
                sighted_pos[0], sighted_pos[1] = rnd(sighted_pos[0:2])
                sighted_fit = objfunc(sighted_pos)
                if sighted_fit < fitness_array[i, j]:
                    fitness_array[i, j] = sighted_fit
                    glider_array[i, j] = np.copy(sighted_pos)
            else:
                r1 = np.ones(dim)
                r2 = np.ones(dim)
                r3 = np.ones(dim)
                for k in range(len(r1)):
                    r1[k] = uniform(t, 2-t)
                    r2[k] = uniform(t, 2-t)
                    r3[k] = uniform(-2, 2)
                current = np.copy(glider_array[i, j])
                dx = (np.copy(glider_array[i, 0]) - current)*r1 + \
                     (np.copy(glider_array[i, 1]) - current)*r2*t**(1/cp)  + \
                     (np.copy(home) - current)*r3*(1-t)**cp
                glider_array[i, j] = current + 0.5*dx
                glider_array[i, j] = np.clip(glider_array[i, j], lb, ub)
                glider_array[i, j, 0], glider_array[i, j, 1] = rnd(glider_array[i, j, 0:2])
                fitness_array[i, j] = objfunc(glider_array[i, j])

    count += 1
#----- Main Loop End -----

#---- Setting Best Values -----
best_fit = 1e100
for i in range(0, colonies):
    if np.amin(fitness_array[i]) < best_fit:
        best_fit = np.amin(fitness_array[i])
        best_pos = list(glider_array[i, np.argmin(fitness_array[i])])

return best_fit, best_pos
```

APPENDIX 6 – MATHEMATICAL FUNCTION TESTER PYTHON CODE

```
# -*- coding: utf-8 -*-
"""
Created on Thu Aug 11 15:27:30 2016

@author: s4317871
"""

from __future__ import division
from SGA import sga
from math import sin, cos, pi, exp, sqrt
import numpy as np

# ==============================================================================
#                          Classical Benchmarking Problems
# ==============================================================================

def fun1(arg): # Min @ 0.0, D=5, [-100, 100], Sphere
    return sum(arg**2)

def fun2(arg): # Min @ 0.0, D=2, [-100, 100], Schaffer's F6
    x, y = arg
    eq = 0.5 + ((sin(sqrt(x**2+y**2)))**2-0.5)/(1+0.001*(x**2+y**2))**2
    return eq

def fun3(arg): # Min @ 0.0, D=50, [-600, 600], Griewank
    dim = len(arg)
    lstA = []
    lstB = []
    for i in range(1, dim+1):
        x = arg[i-1]
        lstA.append(x**2)
        lstB.append(cos(x/sqrt(i)))
    return 1 + (1/4000)*sum(lstA) - np.prod(lstB)

def fun4(arg): # Min @ 0.0, D=50, [-5.12, 5.12], Rastrigin
    dim = len(arg)
    lst = []
    for i in range(1, dim+1):
        x = arg[i-1]
        lst.append(x**2 - 10*cos(2*pi*x) + 10)
    return sum(lst)

def fun5(arg): # Min @ 0.0, D=50, [-50, 50], Rosenbrock
    dim = len(arg)
    lst = []
    for i in range(1, dim): #i=1 to n-1
        x = arg[i-1]
        y = arg[i]
        lst.append(100*((y-x**2)**2)+(x-1)**2)
    return sum(lst)
```

Continued over page.

```python
lst1 = []
lst2 = []
lst3 = []
lst4 = []
lst5 = []

runs = 30
glider_num = 5
iters1 = 20000
iters2 = 100000

for i in range(0, runs):
    val = sga(fun1, [-100]*5, [100]*5, gliders=glider_num, itermax=iters1)
    lst1.append(val[0])

print 'Average1 is:', np.average(lst1)
print 'Std dev1 is:', np.std(lst1)

for i in range(0, runs):
    val = sga(fun2, [-100]*2, [100]*2, gliders=glider_num, itermax=iters1)
    lst2.append(val[0])

print 'Average2 is:', np.average(lst2)
print 'Std dev2 is:', np.std(lst2)

for i in range(0, runs):
    val = sga(fun3, [-600]*50, [600]*50, gliders=glider_num, itermax=iters2)
    lst3.append(val[0])

print 'Average3 is:', np.average(lst3)
print 'Std dev3 is:', np.std(lst3)

for i in range(0, runs):
    val = sga(fun4, [-5.12]*50, [5.12]*50, gliders=glider_num, itermax=iters2)
    lst4.append(val[0])

print 'Average4 is:', np.average(lst4)
print 'Std dev4 is:', np.std(lst4)

for i in range(0, runs):
    val = sga(fun5, [-50]*50, [50]*50, gliders=glider_num, itermax=iters2)
    lst5.append(val[0])

print 'Average5 is:', np.average(lst5)
print 'Std dev5 is:', np.std(lst5)
```

APPENDIX 7 – ENGINEERING PROBLEM TESTER PYTHON CODE

```
# -*- coding: utf-8 -*-
"""
tester_classical_engineering.py

@author: Tim Cassell (t.cassell@uq.edu.au)

Test script for classical engineering design problems. The problems are taken
from the paper by C. Coello Coello.

References:
Coello Coello, C. A. (2000). Use of a self-adaptive penalty approach for
    engineering optimization problems. Computers in Industry, 41(2), 113-127.
    doi:http://dx.doi.org/10.1016/S0166-3615(99)00046-9


"""
from __future__ import division
from SGA_pressure_vessel import sga_pv
from SGA import sga
from math import pi, sqrt
import numpy as np


# ==============================================================================
#                     Constrained Engineering  Problems
# ==============================================================================


def spring_fun(var):
    x1 = var[0] # Wire diameter, d
    x2 = var[1] # Mean coil diameter, D
    x3 = var[2] # Number of active coils, N

    obj_fun = (x3 + 2)*x2*x1**2

    g1 = 1 - (x2**3*x3)/(71785*x1**4)
    g2 = (4*x2**2-x1*x2)/(12566*(x2*x1**3 - x1**4)) + 1/(5108*x1**2) - 1
    g3 = 1 - (140.45*x1)/(x2**2*x3)
    g4 = (x1 + x2)/1.5 - 1

    g_list = [g1, g2, g3, g4]
    violate_list = []

    for i in g_list:
        if i <= 0:
            violate_list.append(0)
        else:
            violate_list.append((10**15)*i**2)

    return obj_fun + sum(violate_list)


def beam_fun(var):
    x1 = var[0] # Thickness of weld, h
    x2 = var[1] # Length of attached bar, l
    x3 = var[2] # Height of the bar, t
    x4 = var[3] # Thickness of the bar, b

    obj_fun = 1.10471*x1**2*x2 + 0.04811*x3*x4*(14 + x2)

    P, L, delta_max, E, G, tau_max, sigma_max = \
       6000, 14, 0.25, 30e6, 12e6, 13600, 30000
```

Continued over page.

```
    M = P*(L + x2/2)
    R = sqrt(0.25*x2**2 + ((x1 + x3)/2)**2)
    J = 2*(sqrt(2)*x1*x2*((x2**2/12)+((x1+x3)/2)**2))
    tau_dash = P/(sqrt(2)*x1*x2)
    tau_ddash = M*R/J

    tau = sqrt(tau_dash**2 + 2*tau_dash*tau_ddash*x2/(2*R) + tau_ddash**2)
    sigma = (6*P*L)/(x4*x3**2)
    delta = (4*P*L**3)/(E*x3**3*x4)
    Pc = (4.013*E*sqrt(x3**2*x4**6/36))/(L**2) * (1 - (x3/(2*L))*sqrt(E/(4*G)))

    g1 = tau - tau_max
    g2 = sigma - sigma_max
    g3 = x1 - x4
    g4 = 0.10471*x1**2 + 0.04811*x3*x4*(14 + x2) - 5
    g5 = 0.125 - x1
    g6 = delta - delta_max
    g7 = P - Pc

    g_list = [g1, g2, g3, g4, g5, g6, g7]
    violate_list = []

    for i in g_list:
        if i <= 0:
            violate_list.append(0)
        else:
            violate_list.append((10**15)*i**2)

    return obj_fun + sum(violate_list)


def pressure_fun(var):
    x1 = var[0] # Thickness of the shell, Ts
    x2 = var[1] # Thickness of the head, Th
    x3 = var[2] # Inner radius, R
    x4 = var[3] # Length of the cylindrical section without head, L

    obj_fun = 0.6224*x1*x3*x4 + 1.7781*x2*x3**2 + \
              3.1661*x1**2*x4 + 19.84*x1**2*x3


    g1 = -1*x1 + 0.0193*x3
    g2 = -1*x2 + 0.00954*x3
    g3 = -pi*x3**2*x4 - (4/3)*pi*x3**3 + 1296000
    g4 = x4 - 240

    g_list = [g1, g2, g3, g4]
    violate_list = []

    for i in g_list:
        if i <= 0:
            violate_list.append(0)
        else:
            violate_list.append((10**15)*i**2)
    return obj_fun + sum(violate_list)
```

Continued over page.

```
lstA1 = []
lstB1 = []
lstC1 = []
lstA2 = []
lstB2 = []
lstC2 = []

runs = 1
glider_num = 5
iters = 5000

for i in range(0, runs):
    lbA = [0.05, 0.25, 2.00]
    ubA = [2.00, 1.30, 15.0]
    val = sga(spring_fun, lbA, ubA, gliders=glider_num, itermax=iters)
    lstA1.append(val[0])
    lstA2.append(val[1])

print 'MinimumA is:', min(lstA1)
print 'AverageA is:', np.average(lstA1)
print 'MaximumA is:', max(lstA1)
print 'Std devA is:', np.std(lstA1)

for i in range(0, runs):
    lbB = [0.1, 0.1, 0.1, 0.1]
    ubB = [2, 10, 10, 2]
    val = sga(beam_fun, lbB, ubB, gliders=glider_num, itermax=iters)
    lstB1.append(val[0])
    lstB2.append(val[1])

print 'MinimumB is:', min(lstB1)
print 'AverageB is:', np.average(lstB1)
print 'MaximumB is:', max(lstB1)
print 'Std devB is:', np.std(lstB1)

for i in range(0, runs):
    lbC = [0, 0, 10, 10]
    ubC = [99, 99, 200, 200]
    val = sga_pv(pressure_fun, lbC, ubC, gliders=glider_num, itermax=iters)
    lstC1.append(val[0])
    lstC2.append(val[1])

print 'MinimumC is:', min(lstC1)
print 'AverageC is:', np.average(lstC1)
print 'MaximumC is:', max(lstC1)
print 'Std devC is:', np.std(lstC1)
```

APPENDIX 8 – GEARBOX EVALUATION PYTHON CODE

```
# -*- coding: utf-8 -*-
"""
Created on Tue Sep 20 10:04:48 2016

@author: s4317871
"""

from __future__ import division
import numpy as np
from math import sin, cos, pi, sqrt
from SGA_integer import sga_int


def gearbox_eval(input_var):
    """
    N_P1 = Number of teeth of Pinion1 or Gear1
    FW_1 = Face width MULTIPLIER of the first meshing pair (Width = FW_1*MO_1)
    MO_1 = Module of the first meshing pair

    """
    N_P1, N_G1, N_P2, N_G2, N_P3, N_G3, N_P4, N_G4, N_P5, N_G5, N_P6, N_G6, \
    FW_1, FW_2, FW_3, FW_4, FW_5, FW_6, MO_1, MO_2, MO_3, MO_4, MO_5, MO_6 \
    = input_var


    ## ---------- CONSTANTS ----------
    Tin = 9.41 #Nm
    Vin = 60 #RPM
    Tout = 5000 #Nm
    Vout = 0.112875846 #RPM
    Power = Vin*2*pi*Tin/60

    ContactFatigueLimit = 17921 #MPa
    BendingFatigueLimit = 1156 #MPa
    Total_Reduc = Vin/Vout


    N_array = np.array([N_P1, N_G1, N_P2, N_G2, N_P3, N_G3, \
                        N_P4, N_G4, N_P5, N_G5, N_P6, N_G6])
    FW_array = np.array([FW_1*MO_1, FW_2*MO_2, FW_3*MO_3, FW_4*MO_4, FW_5*MO_5, FW_6*MO_6])
    MO_array = np.array([MO_1, MO_2, MO_3, MO_4, MO_5, MO_6])

    Reduc_array = np.zeros([6])
    for i in range(0, len(Reduc_array)):
        Reduc_array[i] = N_array[2*i+1]/N_array[2*i]

    RPM_array = np.zeros([12])
    for i in range(len(RPM_array)):
        if i == 0:
            RPM_array[i] = Vin
        else:
            if i%2 == 0:
                RPM_array[i] = RPM_array[i-1]
            else:
                RPM_array[i] = RPM_array[i-1]/Reduc_array[int(i/2)]
```

```
## ---------- GEAR STRESS CONSTANTS ----------
PA = 20*(2*pi/360) #Rads
Ko = 1.5
Cp = 191

Cg_array = np.zeros([6])
for i in range(len(Cg_array)):
    if MO_array[i] < 5.4:
        Cg_array[i] = 0.85
    else:
        Cg_array[i] = 1

Dp_array = np.zeros([12])
I_array = np.zeros([12])
for i in range(len(N_array)):
    index = int(0.5*(i - i%2))
    Dp_array[i] = N_array[i]*MO_array[index]
    I_array[i] = 0.5*sin(PA)*cos(PA)*\
                (Reduc_array[index]/(1 + Reduc_array[index]))

PLV_array = np.zeros([12])
for i in range(len(PLV_array)):
    PLV_array[i] = (pi/12)*Dp_array[i]*RPM_array[i]*0.0393701

Ft_array = np.zeros([12])
for i in range(len(Ft_array)):
    Ft_array[i] = Power/(PLV_array[i]*0.00508)

J_xp = [12, 15, 17, 20, 24, 30, 35, 40, 50, 60, 80, 125, 275]
J_yp = [0.21, 0.25, 0.29, 0.34, 0.365, 0.38, 0.4, 0.42, 0.43, 0.45, 0.455, 0.475, 0.5]
J_array = np.interp(N_array, J_xp, J_yp)

Km_xp = [0, 1, 6, 9, 16, 100]
Km_yp = [1, 1.3, 1.4, 1.5, 1.8, 1.8]
Km_array = np.interp(FW_array/25.4, Km_xp, Km_yp)

Kv_array = np.zeros([12])
for i in range(len(Kv_array)):
    Kv_array[i] = (50 + sqrt(PLV_array[i]))/50

ContactStress_array = np.zeros([12])
BendingStress_array = np.zeros([12])
for i in range(len(N_array)):
    ContactStress_array[i] = \
            Cp*sqrt(Ft_array[i]*Kv_array[i]*Ko*Km_array[int(0.5*(i - i%2))] \
                        /(FW_array[int(0.5*(i - i%2))]*Dp_array[i]*I_array[i]))
    BendingStress_array[i] = \
            Ft_array[i]*Kv_array[i]*Ko*Km_array[int(0.5*(i - i%2))] \
                /(J_array[i]*MO_array[int(0.5*(i - i%2))]*FW_array[int(0.5*(i - i%2))])


## ---------- CONSTRAINTS ---------- ##

#Contact Stress
g1 = ContactStress_array - ContactFatigueLimit
#Bending Stress
g2 = BendingStress_array - BendingFatigueLimit
#Total Reduction
g3 = abs(np.prod(Reduc_array) - Total_Reduc)

violate_list = []

for i in g1:
    if i > 0:
        violate_list.append((10**15)*i**2)
for i in g2:
    if i > 0:
        violate_list.append((10**15)*i**2)
if g3/Total_Reduc - 0.01 > 0:
    violate_list.append((10**15)*g3**2)
```

```
    ## ---------- OBJECTIVE FUNCTION ---------- ##
    obj_fun = sum(((pi/4)*Dp_array)*(np.repeat(FW_array, 2)))*0.00785


    return obj_fun + sum(violate_list)


if __name__ == '__main__':

    lb = [12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, \
            9, 9, 9, 9, 9, 9, 1, 1, 1, 1, 1, 1]
    ub = [60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, \
            14, 14, 14, 14, 14, 14, 20, 20, 20, 20, 20, 20]

    lst1, lst2 = [], []
    runs = 30

    for i in range(0, runs):
        val = sga_int(gearbox_eval, lb, ub, gliders=5, cp=5, itermax=5000)
        lst1.append(val[0])
        lst2.append(val[1])

    print 'Minimum is:', min(lst1)
    print 'Average is:', np.average(lst1)
    print 'Maximum is:', max(lst1)
    print 'Std dev is:', np.std(lst1)
```

APPENDIX 9 – BAREBONES TESTING SCRIPT

```
# -*- coding: utf-8 -*-
"""
Created on Wed Oct 26 10:07:00 2016

@author: Tim Cassell
@email: tcassell@uq.edu.au

A barebones script that sllows the user to simply fill out the function
definition and the variable ranges. Ensure that the SGA script is in the
same folder. Then, press "Run" or hit F5.
"""



from __future__ import division
from SGA import sga


def function(var):
    """
    function(list) -> float

    The input variable, var, is a list of length equal to the dimension
    of the problem.
    """



    return



if __name__ == '__main__':
    lower_bounds = []
    upper_bounds = []

    iterations = 1000
    convergence_power = 5

    out = sga(function, lower_bounds, upper_bounds, \
            cp=convergence_power, itermax=iterations)
    print 'Optimum value is:', out[0]
    print 'Optimum location:', out[1]
```

Appendix 10 – Proof of Results

Mathematical Function Results

```
In [7]: runfile('D:/Users/s4317871/Desktop/tester_abc.py', wdir='D:/Users/s4317871/Desktop')
Average1 is: 3.85227513152e-90
Std dev1 is: 4.05937112949e-90
Average2 is: 0.0
Std dev2 is: 0.0
Average3 is: 3.30846461338e-15
Std dev3 is: 1.74471115604e-15
Average4 is: 261.842445731
Std dev4 is: 32.1143319812
Average5 is: 39.1265657219
Std dev5 is: 0.198243739314
```

Engineering Design Problem Results

```
In [440]: runfile('D:/Users/s4317871/Desktop/tester_classical_engineering.py', wdir='D:/Users/s4317871/Desktop')
Reloaded modules: solution, SGA_test_ok_plot, SGA_classical_engineering
MinimumA is: 0.0126652489646
AverageA is: 0.0128988719897
MaximumA is: 0.0152690260267
Std devA is: 0.000473050752739
MinimumB is: 1.72485579752
AverageB is: 1.72999706015
MaximumB is: 1.77763063677
Std devB is: 0.0122703009435
MinimumC is: 6059.71433734
AverageC is: 6231.6808089
MaximumC is: 7381.71748509
Std devC is: 282.374733713

In [441]: argmin(lstA1)
Out[441]: 5

In [442]: lstA2[5]
Out[442]: [0.051659296387706735, 0.35600210167716084, 11.331045049893579]

In [443]: argmin(lstB1)
Out[443]: 4

In [444]: lstB2[4]
Out[444]:
[0.20572706303552507,
 3.4705441252933311,
 9.0366239088188127,
 0.2057296398561457]

In [445]: argmin(lstC1)
Out[445]: 5

In [446]: lstC2[5]
Out[446]: [0.8125, 0.4375, 42.098445577040096, 176.63659607558546]
```