Evolutionary Learning of Synthetic Circuits

Designing a Genetic Algorithm to Optimise a Synthetic Biological Simulation

Department of Computing and Mathematics

Manchester Metropolitan Univeristy

27-09-2019

Acknowledgements

I would like to thank my supervisor, Dr. Matteo Cavaliere, whose guidance and support led to the completion of this project. I wish to thank the developers of all the software used herein, especially the people behind KaSim, without which this work would not be possible. I would also like to thank [redacted] for her continued support and troubleshooting assistance.

Abstract

A genetic algorithm is a type of artificial adaptive system that uses evolutionary processes to effectively learn how to perform tasks over several generations. We apply such an algorithm to stochastic rule-based simulation of a synthetic molecular biological system. We detail the design of this algorithm and how it was implemented, describing the structure of the algorithm and the hardware and software used. We present analysis of the results of gathered from the algorithm in the form of processed values extracted from the 'fitness function', and the raw unprocessed simulation output. We find that the algorithm can successfully optimise the results of the simulation over a number of generations and discuss the strengths of this method. We also present additional features that could be implemented to extend the algorithm's functionality and develop it to a stage suitable for real-world applications in the field of synthetic biology.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Genetic Algorithms

Adaptation is a process that originates in the natural world. Through sensory interaction with its environment, a living organism processes information about the world it inhabits and makes decisions using this information. The success of an organism is defined by how effectively it can perform tasks essential to its survival. Highly successful organisms will be those that excel at finding food and shelter, avoiding danger and attracting potential mates. Since they will inevitably live longer, successful organisms pass on their genetic information to future generations. If the offspring of the successful organism inherit genetic traits that maximise their effectiveness, then they will be successful too. Over many generations the organisms will refine their performance though this inheritance.

The idea of applying the logic behind natural adaptation was first popularise by John Holland in the 1960s [1] and expanded upon in his 1992 book 'Adaptation in Natural and Artificial Systems' [2]. Holland presents a method of study that incorporates influence from several seemingly disparate fields: from genetics and psychology to game theory and artificial intelligence. The purpose of combining various aspects of these fields was to present a way to mimic the adaptive nature of organic systems in an artificial system. He presents a comprehensive formal and theoretical behind the ideas that later genetic algorithms (GAs) would go on rely on.

Genetic algorithms are used to handle analysis of systems that could not be completed by ordinary manual methods in a reasonable timescale. With a

population of potential solutions to a problem, a GA can process each of these solutions and evaluate their effectiveness before 'breeding' a new population of solutions. The new population will be constructed in a way that allows the most successful solutions to pass on characteristics they possess with the intent of creating new solutions that find even more success. This process of evaluating the population and creating a new population is analogous to the natural selection described at the start of this report. The successful solutions are successful organisms and the process of creating new solutions from the current population mimics the way these successful organisms reproduce and create offspring that inherit their characteristics, or genetic information. Because of this close analogy, the term used to describe a population of solutions is a 'Generation', since each of these populations will be used to create the next. Each generation is the production of reproduction between members of the previous generation. The term used to refer to the solutions that make up these generations is a 'genotype'. The biological meaning of this term refers to the complete genetic makeup of an organism. In a GA, a genotype is typically made up a series of variables, the values of which are analogous to the genetic information contained within a biological genotype. The vast numbers of genotypes that can exist within a single generation present the biggest strength of using a GA: The ability to process amounts of data that would be impossible for any individual or even team of human analysts. The random seeding of the initial generation provides massive variation to the algorithm. Where it could potentially take a scientist weeks to research a topic and meticulously arrive at the optimal solution to a problem using a typical scientific process, a GA can blindly process millions of

potential solutions and find one that meets all the required criteria. Each individual genotype has a very low chance of producing a perfect or even adequate solution to a problem, but the magnitude of solutions processed through the duration of a GA means that finding a solution is only a matter of time and processing power. The GA will also require no prior input on the topic of the problem: it will not need to be taught how the problem works or told how to approach the problem, only what is to be classified as a successful solution.

The method of implementing the principles of genetic algorithms used for the development of this project was based on a simplified structure presented on the Nature of Code website created by Daniel Shiffman [3]. This implementation involves creating genotypes consisting of an array or floating-point variables. The value of each of these variables is a single genetic characteristic that the genotype possesses. We then genotypes are then used to perform a simulation using their variable values as simulation parameters. Each genotype will have an output from this process, and it is this output that is judged to work out how successful the genotype is. In contrast to how an organism will naturally be successful just by surviving in the world, an artificial genotype will have to be evaluated using a man-made algorithm. This evaluation algorithm is referred to as a 'fitness function', and the score that a genotype receives is its fitness value. The fitness values are essential when populating the next generation, since the genotypes with the highest fitness values are the ones that should be able to pass on their characteristics. We use random function to decide which genotypes will breed, so the method used to maximise the successful genotypes is to make sure they have a higher

probability of being randomly selected. We make a gene pool and fill it with all the potential parents of the future generation. Instead of only adding each genotype once, as this would mean they all have the same chance of being selected, we add the most successful genotypes into the gene pool multiple times. This method ensures that a genotype added in only once has a much lower chance of being randomly selected from the gene pool than one added in tens of times.

The power of the GA comes from its ability to iterate over this entire process numerous times. Natural selection only takes shape over many generations, so many artificial generations are required to show the results of a GA at work. There are numerous fields that genetic algorithms are applicable too, since their ability to process complex solutions to problems is not dependent on any understanding of the problem.

1.2 Synthetic Biology

The genetic algorithm developed for this project was applied to a Synthetic Biological simulation using the KaSim stochastic simulator [4]. KaSim simulations biological simulations using graphical methods (referring to graph theory, not a visual representation) where each simulation event corresponds to the application of a rewriting event, in which the structure of the simulation graph is altered. KaSim uses a rule-based system where events occur according to defined rules that dictate how objects within the simulation will behave [reference rule-based modelling]. The stochastic nature of KaSim means that, from a specified set of simulation parameters, events are fired randomly.

Synthetic biology is a growing field of research that combines the theory and concepts behind molecular biology with an approach influenced by engineering. In essence, synthetic biology is a kind of biological engineering wherein biological systems can be built from known parts in order to accomplish certain functions. This project was developed using the understanding of synthetic biology presented in Synthetic Biology: A Primer [5]. This book details the theory behind building biological system up component by component in a way similar to how mechanical systems are designed. The main difference between these two types of systems is the inherent complexity found in biology. Biological systems contain components not on a fixed circuit board, but moving in real space, so the components are defined not by their exact physical positions, but by their relationships with each other. If two molecules are connected, they will not be defined by 'molecule a is situated next to molecule b' but by 'molecule a is bonded to molecule b'. Biological systems also feature an evolutionary component, where cells that have damaging or negative properties can be competed out of a population to ensure the system functions properly. This evolutionary component is not present in ordinary engineering, so synthetic biological system must meet new criteria for robustness. A system must be able to continue functioning as its unwanted parts are discarded through natural selection. Biological systems also handle the flow of information differently than mechanical of even electronic systems. In an electronic system, information is passed between components in the form of data that can be read and written as needed.

The theory behind information flow in biological systems is referred to as the 'central dogma'. The central dogma describes how information is stored in

DNA. DNA can self-replicate and, when needed, transcribe information to RNA molecules which act as messengers. The RNA molecules 'deliver' this information to proteins, which are the components that actually carry out the instructions written within the DNA.

2 Design

The following section outlines the structure of the genetic algorithm. We present the list of sections the algorithm was designed to implement along with a flowchart to visualise the structure. We then present each section individually and detail what processes and operations take place within them. Some considerations that were taken in the implementation section are also discussed. Finally we discuss how the loop structure of the algorithm was designed such that it can iterate over multiple generations.

The goal of the project was to create a genetic algorithm that could be used to optimise the output of a simulation of a synthetic biological system to match pre-determined criteria. The general outline of the design of the algorithm can be seen in Figure 1. The flowchart shows the structure that the algorithm was designed to follow and consists of multiple subsections which are separated based on the stage of the algorithm that they represent. The subsections of the algorithm are as follows:

1. Initialisation The first generation is created
2. Simulation Output values are calculated
3. Evaluation Output values are assessed using the fitness function
4. Reproduction A new generation is created using the previous generation

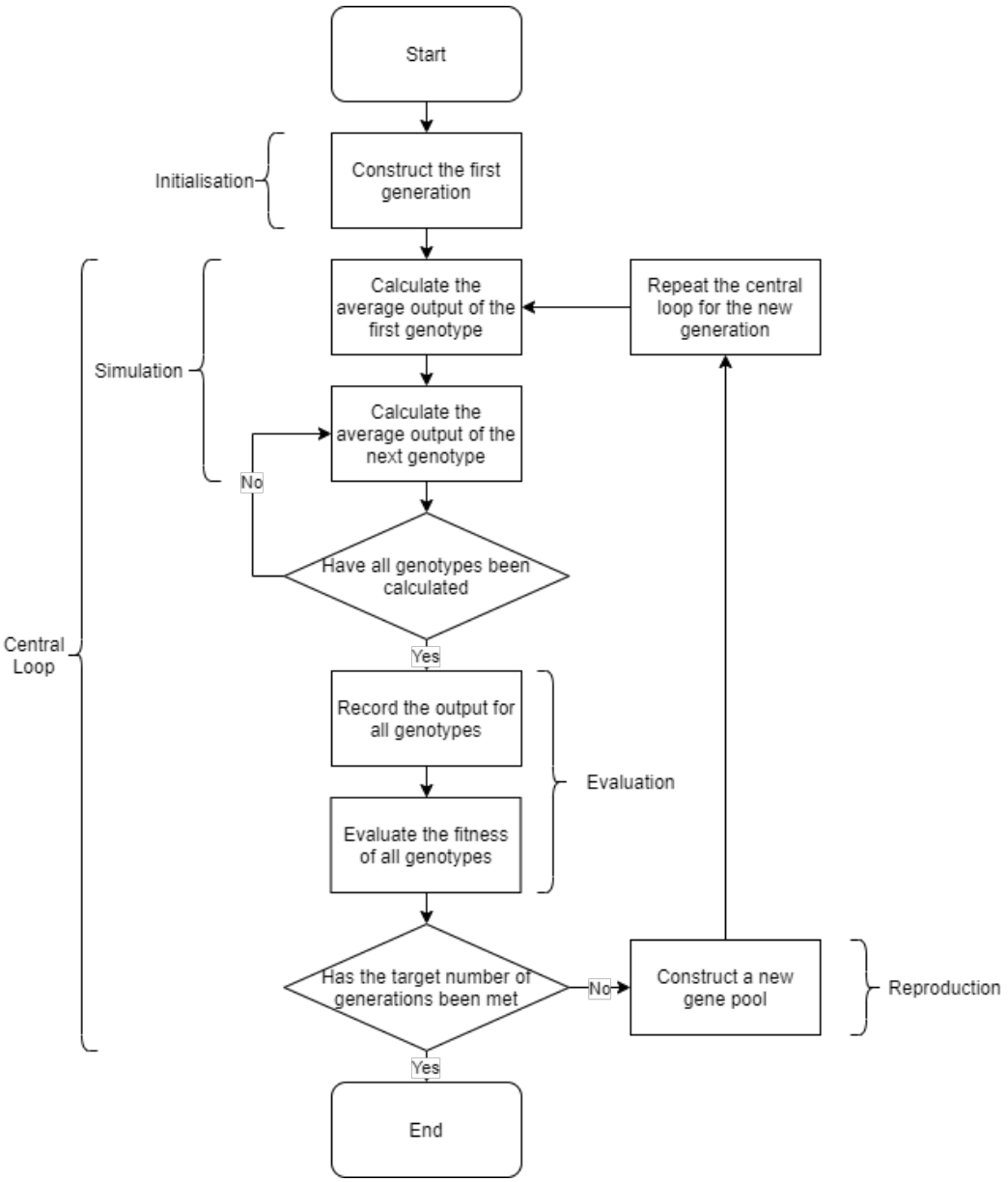

Figure 1: Main Flowchart

2.1 Initialisation

The first stage in the design is the Initialisation stage, in which the algorithm needs to create an initial set of genotypes, or the first generation. The genotypes in this stage needs to be randomly generated to introduce variety between genotypes. Starting with randomly generated genotype values also allows the algorithm to begin 'blind' with no prior information regarding what kind of values could generate desirable results.

2.2 Simulation

Once the first generation is created, the algorithm needs to enter the Central Loop section, which is to be repeated until a pre-determined number of generations had been processed. The first stage of the Central Loop is the Simulation section, during which the genetic algorithm needs to be able to perform a single simulation multiple times to calculate an average outcome for each input file, and then to perform many different simulations and collect the results as a single generation of the algorithm.

The averaging was important since the stochastic nature of the simulation meant that there could be discrepancies between successive instances of identical simulations and by calculating the average any potential anomalous results could be accounted for and minimised. Anomalous results are uncommon by definition, so they should always be countered by a much greater number of more typical results. The algorithm needs to be able to handle the averaging function very quickly due to the high number of simulations to be performed over the course of the total runtime. If 10 averages are taken

of a simulation that takes one minute, then that one simulation will take ten minutes to perform with the required averaging. This was considered when deciding the scale of the algorithm.

For the most optimal results, each of the genotypes that made up a generation would need to be completely different from every other genotype. This would give the largest possible variation in the 'gene pool' as it would minimise the number of duplicate genotypes.

2.3 Evaluation and Reproduction

Once the algorithm has completed the Simulation stage loop for every genotype, it needs to begin the Evaluation stage. In this stage the algorithm needs to evaluate each genotype's averaged output against a fitness function to determine how successful each is at producing a desirable result. To promote the characteristics of the most desirable genotypes, the algorithm needs to 'breed' a new generation using methods of creating new offspring that prioritise the continuation of the highest fitness ranking genotypes, and eliminate the continuation of any genotypes that contain data that is in some way invalid or non-functional. This process is marked as the Reproduction stage of the Algorithm and is directly tied to the evaluation stage.

A new potential gene pool needs to be constructed out of the initial set of genotypes, with the presence of high fitness genotypes amplified and the presence of low fitness genotypes minimised. Each new genotype created will be produced using two 'parent' genotypes selected from the constructed gene pool. By using a 'breeding' algorithm the characteristics of successful parents will be passed on to offspring genotypes. The intention of this is to

create combinations of characteristics that did not exist in any of the parent genotypes but will allow the offspring genotypes to potentially reach a greater fitness value.

2.4 Further Iterations

The Algorithm needs to return to the Simulation stage and carry out the Central Loop again using the newly created offspring genotypes as the new generation. The simulated results for the new generation then have to be evaluated and, if the algorithm has not run for the required number of generations, put through the Reproduction stage. After the algorithm has completed the Central Loop the required number of times, the current active generation needs to exit the loop at the point at which the Evaluation stage ends before the Reproduction stage began.

For the algorithm detailed previously to be considered successful, the output would need to show a clear trend in the direction of the criteria laid out in the fitness function. Each successive generation would need to resemble the ideal outcome more clearly than the last. This would clearly demonstrate progression between generations and would indicate that each newly generated set of offspring has adapted in some way to its conditions.

3 Implementation

The following section outlines how the algorithm design presented in the previous section was implemented in actual code. We first discuss a general overview of what software was used to accomplish this, along with a section detailing how this software functions together. The following sections expand on the algorithm stages presented in the design section and include code sections and flowcharts. The flowcharts show the logic behind the implementation of these stages and the code sections highlight how specific operations within the stages were written. Finally a selection of particularly prominent errors and problems that were encountered during implementation are listed and their solutions described.

A genetic algorithm was designed to optimise the results of a specific simulation performed using a synthetic biological simulation. The genetic algorithm was written in the Python 2.7 programming language [6] on a Linux machine and used the Kappa based stochastic simulator KaSim to handle the synthetic biological component. The interaction between these two components made up the base of the system. The Python algorithm was used to provides inputs to, and control the actions of KaSim and in turn read the simulation outputs to inform the actions of the genetic algorithm.

3.1 Software

The Spyder integrated development environment [7] was used to write the genetic algorithm component of the system. Spyder is designed for scientific computation and data analysis and provides a GUI that simplifies the

debugging process. Spyder provides a very streamlined workflow for iterating on the design of an algorithm as the program can be launched from within the IDE and all variables can be monitored to ensure the program is functioning as intended. Functionality to step through the code line by line also greatly aids the debugging and development process. There were multiple components of the algorithm that required functionality from additional python modules and from custom python scripts and Spyder allowed for easy management of these extra modules. Existing modules can be imported at the start of the file in which they are required, and custom modules can be written easily and saved to the root directory of the main file so they can be imported as an existing module would be. The Spyder interface allows the user to edit multiple files across simultaneous tabs, which also presents the advantage of being able to adjust and test the additional files without running the main file in which they are used.

KaSim is an implementation of the Kappa programming language. Kappa [8] is a rule-based [9] language developed to model systems that consist of interacting agents. The development of Kappa is currently focussed on modelling molecular systems biology but the language has a wide range of applications due to the flexible nature of the rule-based system it uses.

Rule-based modelling is a method of modelling that allows the construction of mechanically complex systems through defined rules. This method is primarily used in cases where the resulting system is much more complex than the set of rules that gave rise to it. Models are repeatedly subjected to a set of rules and conditions that inform its interactions, meaning as the behaviour of the model can be become progressively more complex as the rules

that define this behaviour remain simple in comparison.

A rule-based modelling approach is distinct from an Agent-based model [10], in which the agents are autonomous actors that communicate with each other and respond to changes to their environment. A rule-based model can make complex processing less intensive since actions are only processed when the conditions are met for them to occur; An agent need not simulate interactions with all other agents in the system, only those that meet the criteria for interaction set by the rules of the model. In an agent-based model, agents are 'intuitive', meaning they actively perceive their surroundings and use this information to make decisions. The decision-making process can range from basic 'if' statements where agents perform actions when conditions are met to complex behavioural models. Due to hardware limitations, a rule-based language was preferable. Since ruled-based systems can be less resource intensive, they allow for vastly more simulations to be carried out over the timescale of the project. The genetic algorithm requires tens of thousands of simulations to be performed sequentially and any increase to the time and resources needed to perform a single simulation will have a cumulative effect over all the required simulations.

A KaSim simulation requires an input file of a '.ka' filetype that consists of code that establishes the conditions of the simulation. A template KaSim input file was provided and the algorithm was designed to maximise one of the three observables in the output file that this input file creates. The simulation tracks the values of three variables. It is the third of these that is of interest: P, or the total amount of proteins present in the system. The input file was written to function using KaSim debug version 3.5. This version of

KaSim has since been discontinued in favour of the newer KaSim version 4. To install KaSim 3.5 on a Linux machine, version 3.12 of the Ocaml Compiler [11] must first be installed.

The relevant version of Ocaml caused an error where the installation of KaSim could not be identified, even though test were performed to ensure that KaSim had been installed correctly. A solution to this problem was to use the Wine compatibility layer [12] for Linux to effectively emulate the windows version of KaSim. Wine is used to run Windows applications without emulating a Windows system and instead by translating Windows operations to be used by Linux systems on the fly. The combination of Wine, Ocaml 3.12 and KaSim 3.5 was able to perform simulations successfully.

The reliability of this setup was tested using an example model presented in the KaSim 3.5 manual [13]. The sample code presented for this model was used as the contents of the KaSim input file and the same conditions were set when initialising the simulation and the output when graphed matched the corresponding figure in the manual.

The new version features a GUI that was not present in KaSim 3.5. This means that KaSim must be interacted with through the Linux Terminal. A terminal command must be used to initiate KaSim and specify the location of both the input file and the output file that will be created. This method of using KaSim is only productive for running one simulation at a time. To perform any greater number of simulations, a different terminal command could be used to point to another input file and output file location, or the contents of the initial input file must be modified if the same terminal command is to be used. This method would overwrite the first output file, however.

To make KaSim work in conjunction with a genetic algorithm, which would require potentially tens of thousands of simulations to be run (the total for this algorithm reached around 75000 simulations), a programmatic method to run KaSim simulations was developed.

3.2 Algorithm Structure

For this genetic algorithm, each generation consists of a number, n, of genotypes. Each genotype consists of 8 numeric values, all which correspond to variables within the KaSim input file. Further simulation conditions are then specified in the terminal command. This command also specifies a filename and location for the output file to be saved. The python subprocess was used to perform the Linux terminal command to execute a KaSim simulation from within the Python program.

A custom python module, seen in figure 2, was used to rewrite the contents of the KaSim input file to perform a simulation with the conditions specific to each genotype. The module, named 'Generate', used a compound string made that was written to the same location and with the same filename each time. The compound string consisted of template sections that were to remain the same for each simulation and points at which the relevant genotype's data was used to determine the values of certain variables.

Another custom python module was written to handle the creation of new genotypes, either by randomly assigning values to a new genotype or by implementing a 'breeding' algorithm to pass on characteristics of two parent genotypes to a new offspring genotype.

```
command = 'wine KaSim -i sim002.ka -e 100000 -t 250 -p 1000 -o '

filename = "/home/shaun/Documents/kasim/bin/sim002.ka"

def save_file(filename, variables):
        if os.path.exists(filename):
                os.remove(filename)
        f = open(filename,'w+')
        f.write(make_file(variables))
        f.close()
```

Figure 2: Code Section: Saving input files

3.3 Initialisation

Variable Number	Variable Name	Range	Description of Variable Function
Var1	c	$0 \leq c \leq 1$	Cooperativity
Var2	f	$0 \leq f \leq 1.1$	Enhancement Factor
Var3	r1	$0 \leq r1 \leq 2$	Basal Transcription Rate
Var4	k2	$0 \leq k2 \leq 1$	MRNA Degradation
Var5	k3	$0 \leq k3 \leq 1$	Protein Degradation
Var6	k7	$0 \leq k7 \leq 1$	Translation Rate
Var7	ka-on	$0 \leq$ ka-on ≤ 1	Association Rate
Var8	ka-off	$0 \leq$ ka-off ≤ 1	Dissociation Rate

Table 1: The variables within a KaSim input file that vary between genomes.

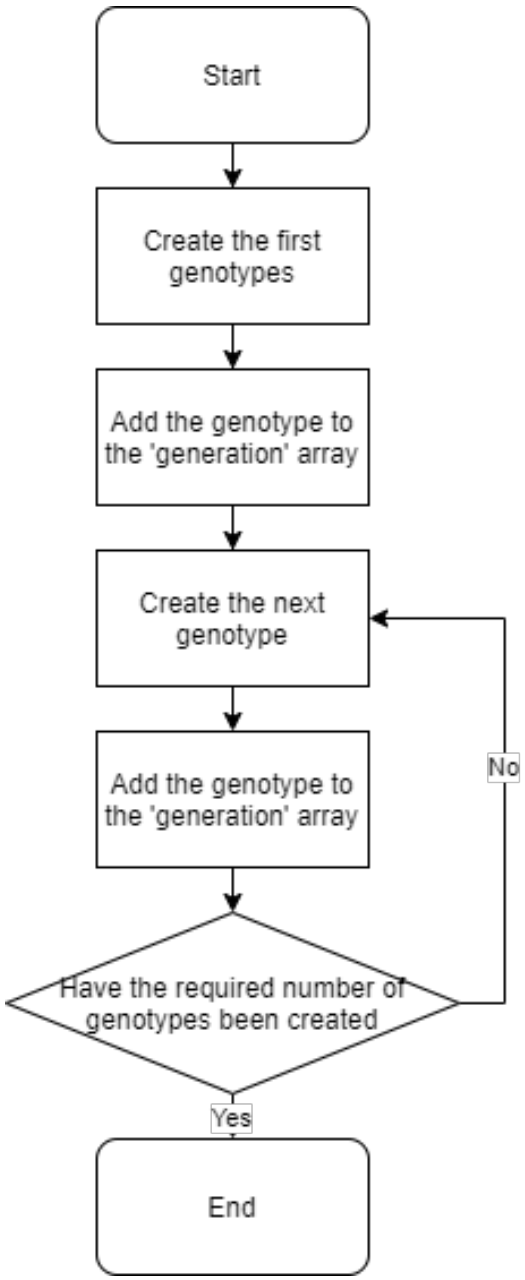

Figure 3: Initialisaition Flowchart: Creating the first generation

The algorithm begins in the Initialisation stage by creating a base set of genomes, all which possess randomly generated values for each its variables. The process for generating a full generation of distinct genomes is outlined in Figure 3 and the more specific process of assigning the random values to every variable in each genome is detailed in Figure **??**. The code used to implement these processes can be seen in Figure 4.

Each variable was assigned possible values in the range $0 < x < 1$. Some values are multiplied up to fit them within a different range. The variables and their functions are listed in Table 1.

```
generation = generate.make_generation(generation_size)

def make_generation(gen_size):
        generation = []
        variables = [0]*8
        for i in range(0,gen_size):
                variables[0] = random.random()
                variables[1] = random.random() * 1.1
                variables[2] = random.random() * 2
                variables[3] = random.random()
                variables[4] = random.random()
                variables[5] = random.random()
                variables[6] = random.random()
                variables[7] = random.random()
                generation.append(np.asarray(variables))

        return generation
```

Figure 4: Code Section: Creating genotypes and population generations

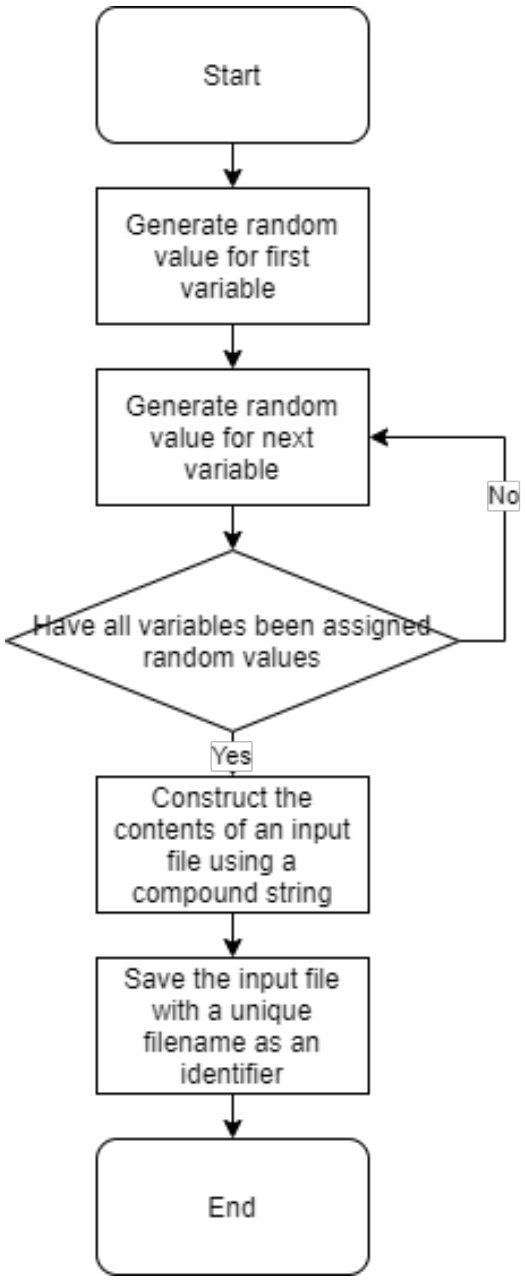

Figure 5: Initialisaition Flowchart: Creating new genotypes

3.4 Simulation

After populating the first generation, the algorithm begins the Simulation stage. The process to run simulations for each genome is shown in Figure 7 and the process used to average the simulation outputs for a single genome and to check for file output errors is detailed in Figure **??**. For each of these genomes, a KaSim simulation is performed using standard parameters [13] for simulation options.

The generic form of the terminal code used to run an instance of KaSim is shown in Figure 6.

```
'wine KaSim -i "input filename" -e 100000 -t 250 -p 1000 -o "output filename"'
```

Figure 6: Code Section: Terminal code for launching KaSim

Where "-e 100000" specifies that the simulation will be carried out over 100000 events with a total of 1000 measurements taken over the course of the simulation (specified using "-p 1000"). The simulation is carried out over 250s (in simulation time-steps, not in real time) as there are typically no notable events that occur after this time in a simulation, and increasing the timescale of the simulation would only increase the processing time of each step in the algorithm without adding any clarity to the results obtained. As the simulation takes place over 250 seconds and records 1000 measurements, the time step between each measurement is 0.25 seconds.

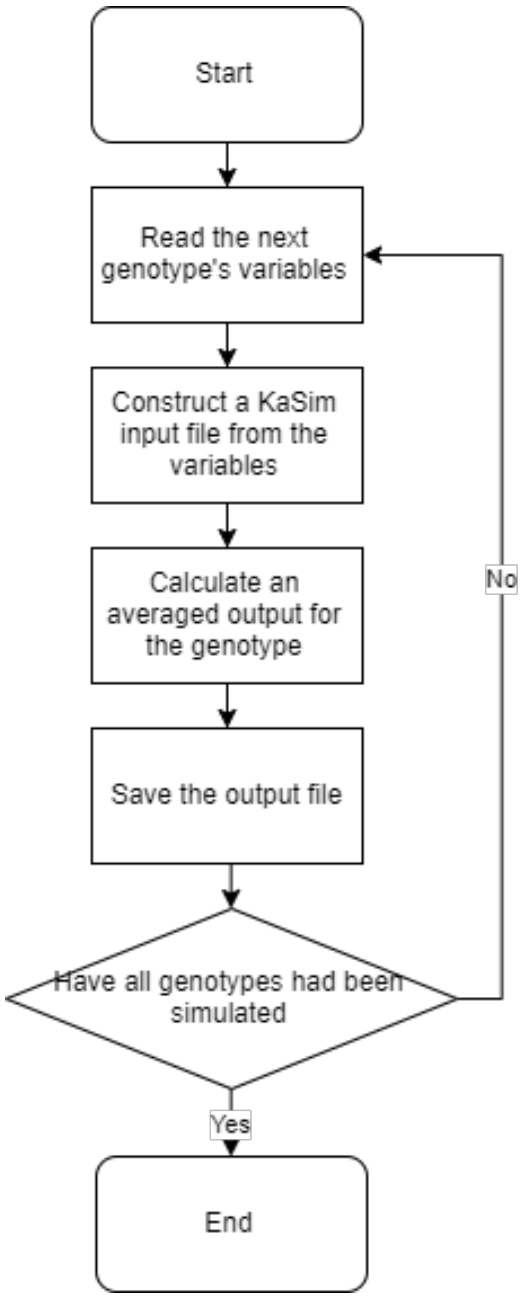

Figure 7: Simulation Flowchart: Running a KaSim simulation

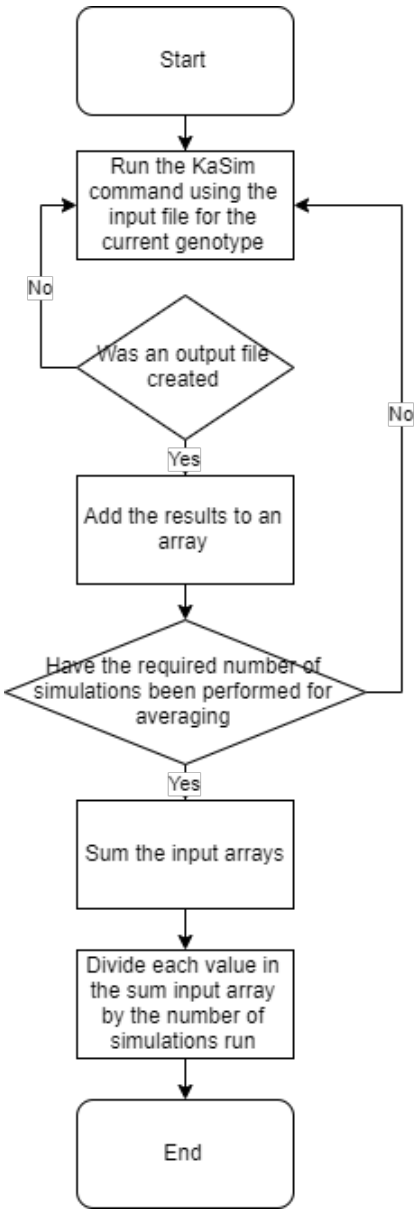

Figure 8: Simulation Flowchart: Averaging the results of a simulations

To average the results of the first genotype the simulation is run n times and creates n separate temporary output files, the numerical contents of which are summed, divided by n and saved to the final output file for that genotype.

```
for i in range(0, len(col1)):
        col1[i] += float(tcol1[i])
        col2[i] += float(tcol2[i])
        col3[i] += float(tcol3[i])
        col4[i] += float(tcol4[i])

col1 = np.array(col1) / repeats
col2 = np.array(col2) / repeats
col3 = np.array(col3) / repeats
col4 = np.array(col4) / repeats
```

Figure 9: Code Section: Summing and averagind simulation results

This averaging is performed to minimise the effect of any anomalous results on the functioning of the algorithm. Potential anomalous results could occur from unlikely behaviour in the simulation due to its stochastic nature, or from any error in the calculations that could occur at this point. The algorithm did not include a function to detect anomalous results as they will always exist alongside a much higher number of typical results. To improve the accuracy of the system, a check could be implemented to determine whether any results are incongruous with the other results that it will be averaged with. If the results are incongruous, the simulation could be run once more to replace the 'bad' results. This feature was not considered in order to minimise the timescale of the algorithm, as the check function would need to analyse all simulation results before they were averaged. It would also be

difficult to algorithmically differentiate between results that were anomalous and results that had ordinary but notably different results.

3.5 Evaluation

The Simulation stage processes every genome and creates an array of outputs corresponding to the genomes that produced them; the collected outputs are then processed in the Evaluation Stage. In this stage the outputs of the previous generation are assessed using a fitness function, shown in Figure 11 and a gene pool is constructed based on the results of this assessment. The whole process is outlined in Figure 10. First the output of each genome is read, and a corresponding fitness value is calculated.

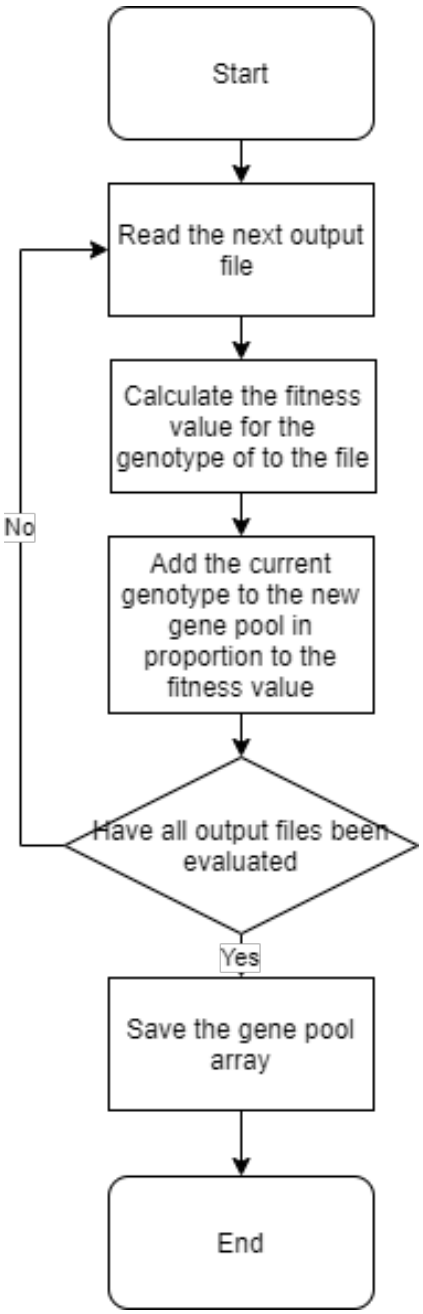

Figure 10: Evaluation Flowchart: Evaluating the results of the genotypes

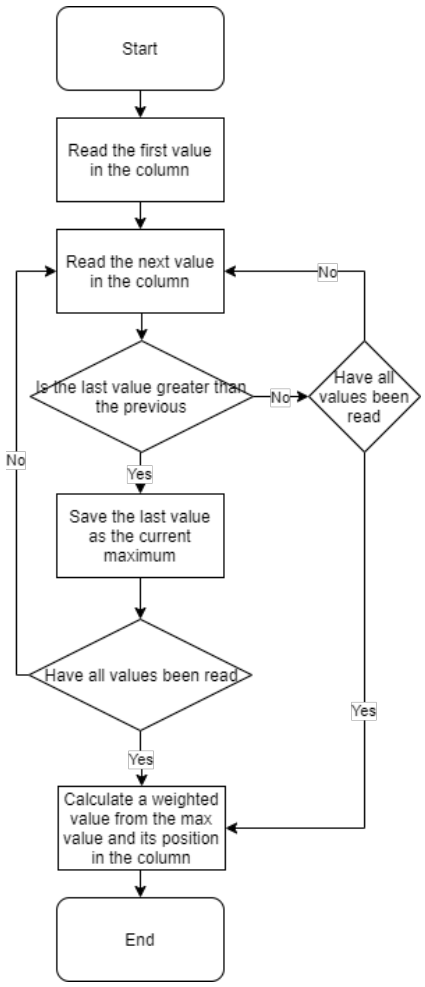

Figure 11: Evaluation Flowchart: Applying the fitness function

The fitness value is used to calculate how prevalent that genome will be in the gene pool. The fitness value is typically in the thousands due to the method used to calculate it, seen in Figure 13 so it is divided by 1000 and rounded to give an integer number. This is referred to as the Weighted Result of the genome. The Weighted Result is the number of instances of that genome that are added to the gene pool. [EXAMPLE: wr = 13, so the genome will exist 13 times in the gene pool]. It would be possible to scale the weighted results each generation such that the maximum weighted result is the same value for each generation, but by leaving them unscaled it is possible to create a gene pool that, while holding a much greater number of values for each generation processed, is a more accurate representation of the fitness values of all the genomes.

The method used to calculate the fitness is designed to positively value the highest peak, or maximum, that the output reaches throughout the simulation and the point at which this highest value is reached. This point is measured using the last position in the output file at which the maximum occurs. To encourage a constant increase in the output value, the fitness evaluation gives a higher value the later the maximum appears in the output. A hypothetical ideal genome, seen in Figure 12, judged using this fitness evaluation would be a genome that reaches a higher maximum output than all other genomes and has a constant increase in its output value across the whole simulation time. The genome's highest output value will be the final entry in the output file, and it will be a higher value than the maximum of all other genome outputs. The output value will have a positive linear correlation with the time elapsed in the simulation.

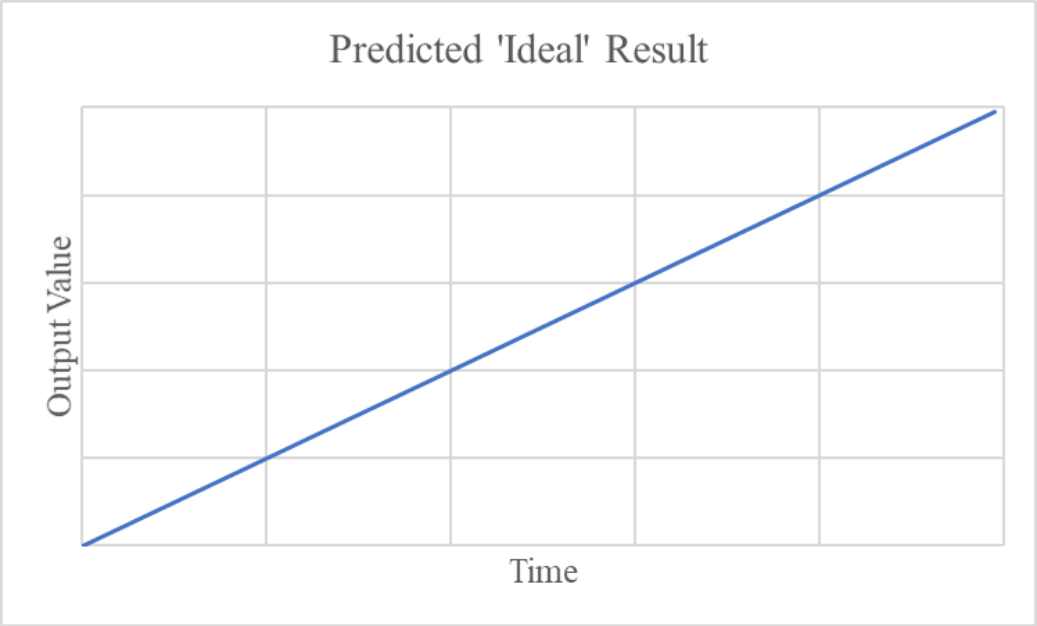

Figure 12: A hypothetical plot of the results of an ideal genome

```
max_val = np.amax(vals)

max_val_pos = np.amax(np.where(vals == max_val))

wr = max_val * (max_val_pos/1000)

gen_fitness.append(wr)

for j in range(0,int(wr)):
        parent_bank.append(i)
```

Figure 13: Code Section: The fitness function used to evaluate genotypes

3.6 Reproduction

The gene pool is used to create a new generation of the same size as the previous one. An offspring genome always contains values inherited from its two parent genomes. The process of creating a new genome is detailed in Figure **??**and the process of assigning parent values to the offspring is shown in Figure 15. The code used to produce new offspring genotypes is shown in Figure 16.

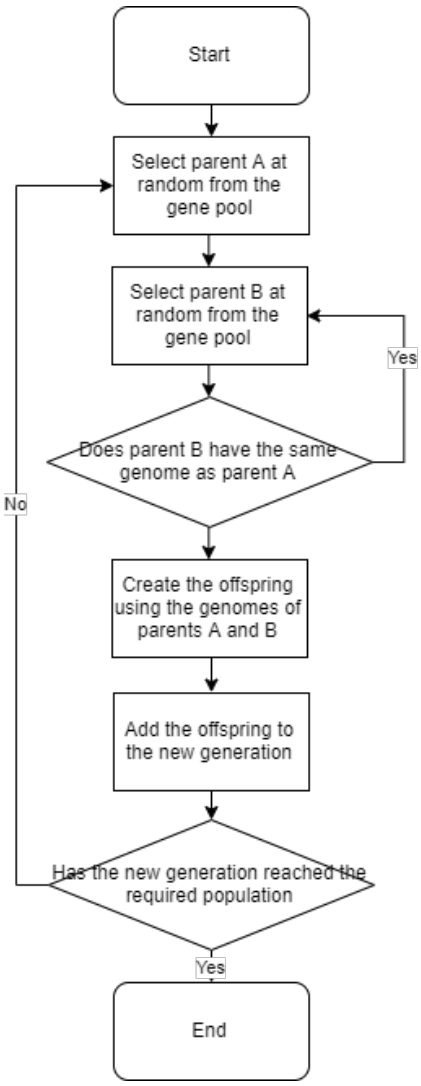

Figure 14: Reproduction Flowchart: Populating a new generation

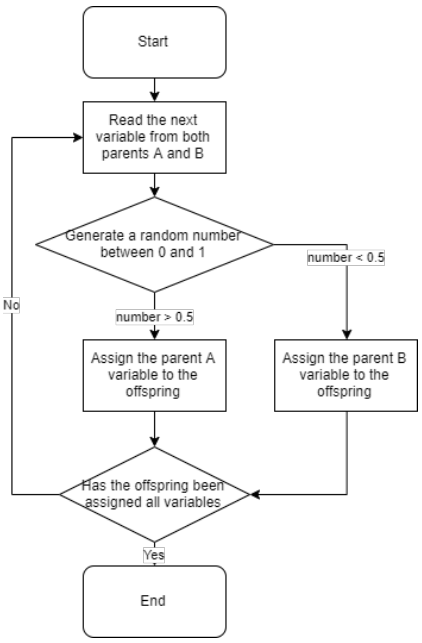

Figure 15: Reproduction Flowchart: Creating an offspring genotype from two parents

Two genomes, a 'Parent A' and a 'Parent B', are selected at random from the gene pool array and a check is performed to determine if both parents are identical genomes. In the case of two identical parents, a replacement 'Parent B' is selected and the check is performed again. Once two distinct parents have been selected a 'Breeding' operation is performed to create an offspring genome using the variable values of the two parent genomes. For each of the 8 variables that make the data in a genome, a random selection is made between the 'Parent A' and 'Parent B' genomes and the value of the variable for the selected genome is assigned to the offspring. The random nature of the breeding process means that multiple offspring with identical set of parents can be assigned different variable values.

```
def make_offspring(parentA, parentB):
        genotype = [0] * 8
        for i in range(0,8):
                if random.random() > 0.5:
                        genotype[i] = parentA[i]
                else:
                        genotype[i] = parentB[i]
        return genotype
```

Figure 16: Code Section: The crossover algorithm used to create offspring genotypes

After fully populating a new generation with offspring genomes, the algorithm returns to the simulation stage and performs another iteration of the central loop. This repetition continues until the number of generations processed meets a pre-decided value. The output files from the latest generation processed will show the progress the algorithm has made towards meeting the ideal outcome of the fitness function.

3.7 Error Analysis

The repetition of the simulation process means that a very high volume of KaSim simulations are performed in succession. Due to some unreliability issues with the version of KaSim used, there was a failure rate when executing the KaSim terminal command of around $\frac{1}{1000}$ attempts. Ordinarily this would not present an issue since any failed attempt could simply be launched and there would be a negligible chance that the simulation would fail twice consecutively. In this algorithm any KaSim launch failure will result in an output file not being created which initially is not a problem as the algorithm moves over the unsuccessful genotype and proceeds to the next as it would if the simulation had been successful. The problem emerges in the Evaluation stage when the algorithm attempts the read the output files to use them to calculate fitness values. The solution used to address this issue was to implement a function to search for a file with the filename of the output file that should have been created. If no such file is found, then the simulation is called again, and the test function is run again. This process repeats until the test finds the correct output file.

4 Results

The following section describes the parameters used to run the genetic algorithm and the results gathered from the output of the KaSim simulations. We first state the variables of the genetic algorithm and the variables used to perform the KaSim simulations and discuss the methods used to process the data. We then present processed data regarding the fitness function output of the algorithm over multiple generations and the raw data taken from the KaSim output files.

	Fitness Values				
Generation	Mean Average	Range	Maximums	Minimums	Medians
1	3.80	3.76×10^3	3.76×10^3	0.00	0.00
			7.07×10^2	0.00	0.00
			4.78×10^2	0.00	0.00
2	2.09×10^3	1.44×10^4	1.44×10^4	0.00	9.62×10^2
			1.43×10^4	0.00	9.55×10^2
			1.40×10^4	0.00	9.53×10^2
3	4.79×10^3	2.76×10^4	2.76×10^4	0.00	4.05×10^3
			2.63×10^2	0.00	4.05×10^3
			2.60×10^2	0.00	4.04×10^3
4	6.41×10^3	2.70×10^4	2.70×10^4	0.00	5.29×10^3
			2.64×10^2	0.00	5.28×10^3
			2.60×10^2	0.00	5.27×10^3
5	8.23×10^3	2.72×10^4	2.72×10^4	0.00	8.09×10^3
			2.72×10^4	0.00	8.08×10^3
			2.67×10^4	0.00	8.08×10^3

Table 2: Average Fitness Values.

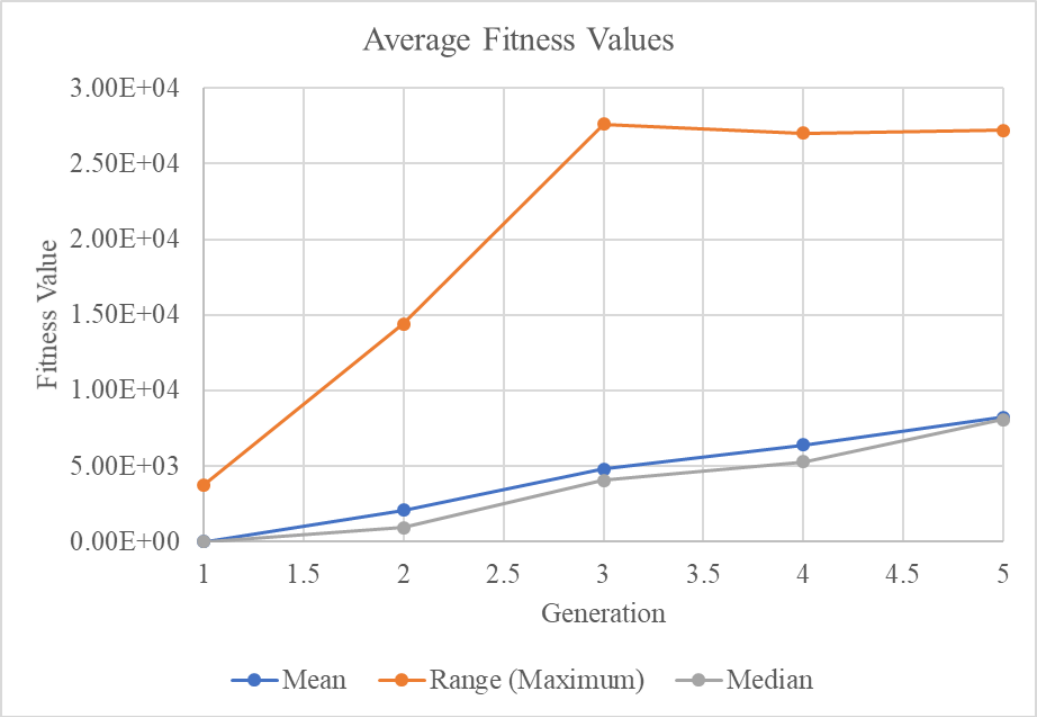

Figure 17: Plot of the average fitness values across generations

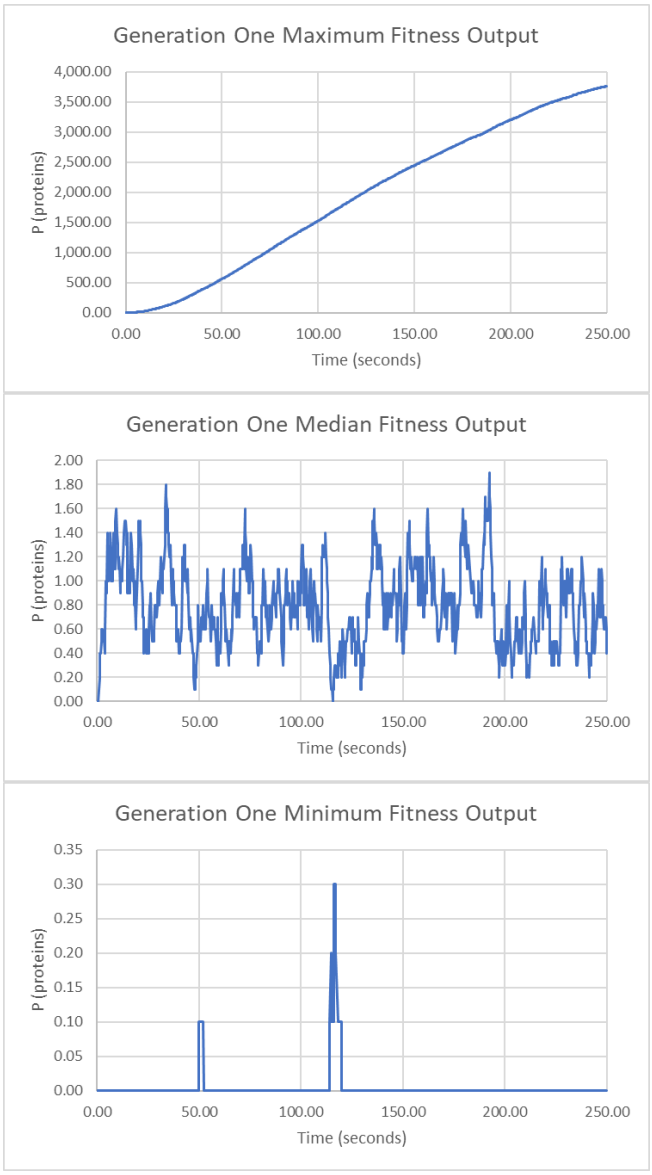

Figure 18: Output plots from the first generation

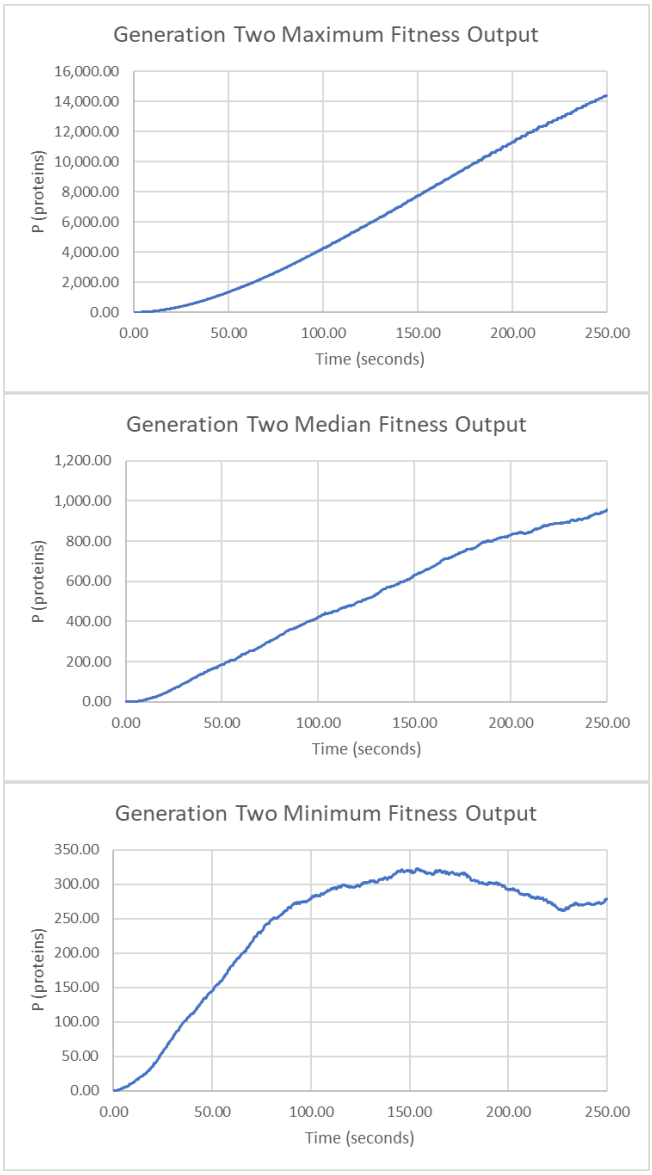

Figure 19: Output plots from the second generation

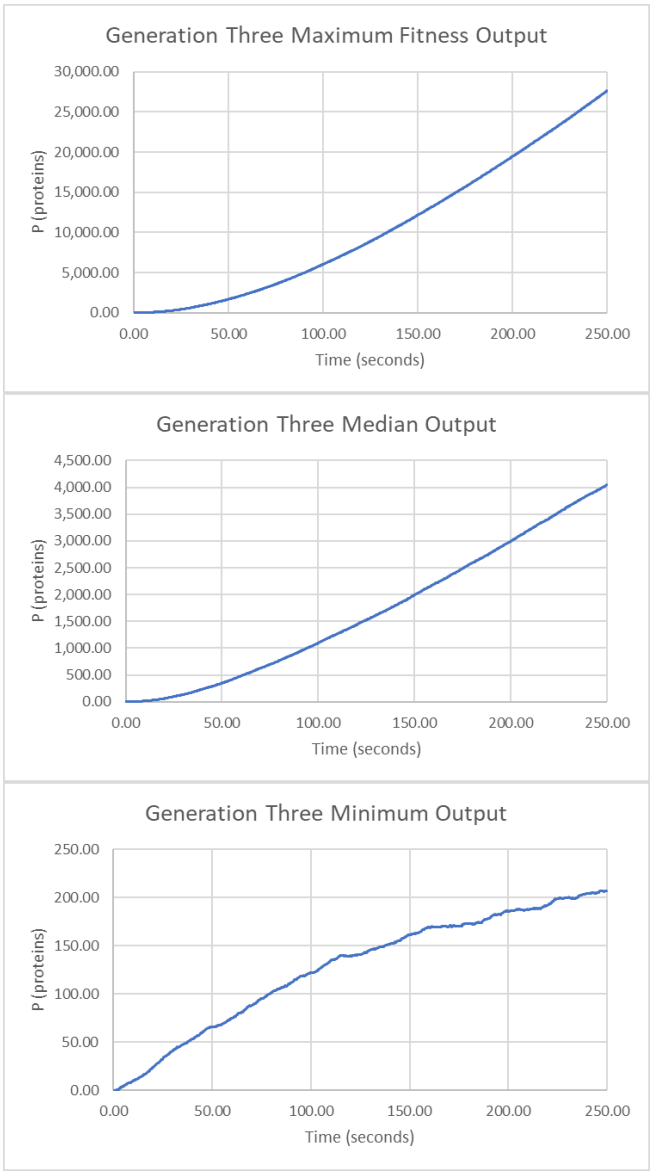

Figure 20: Output plots from the third generation

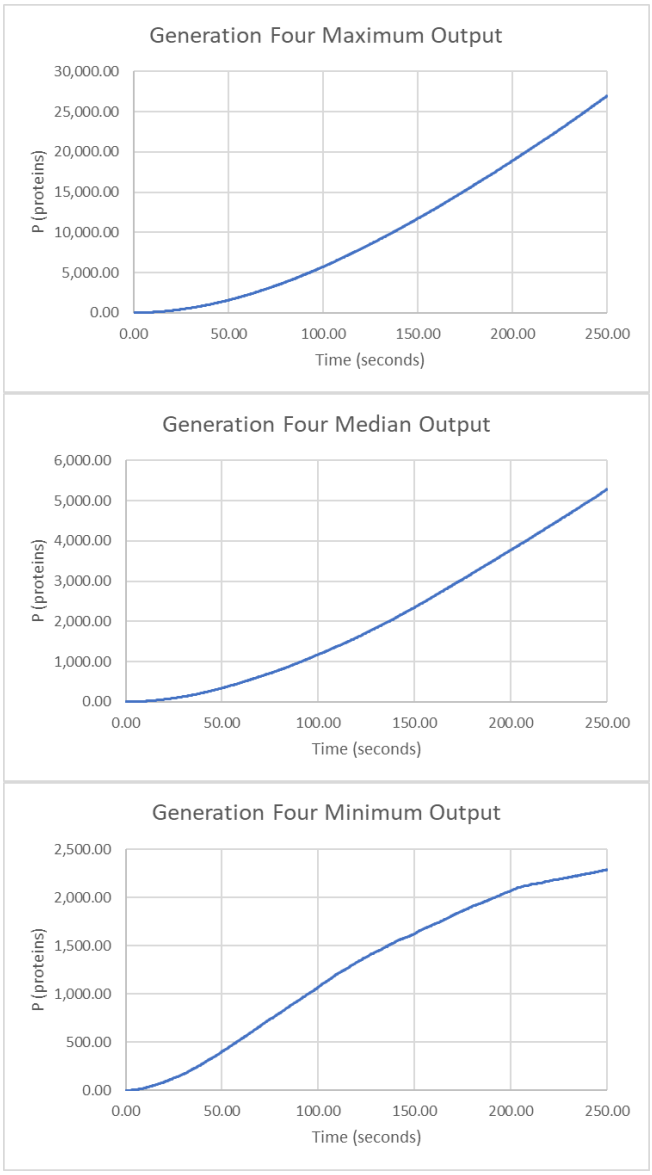

Figure 21: Output plots from the fourth generation

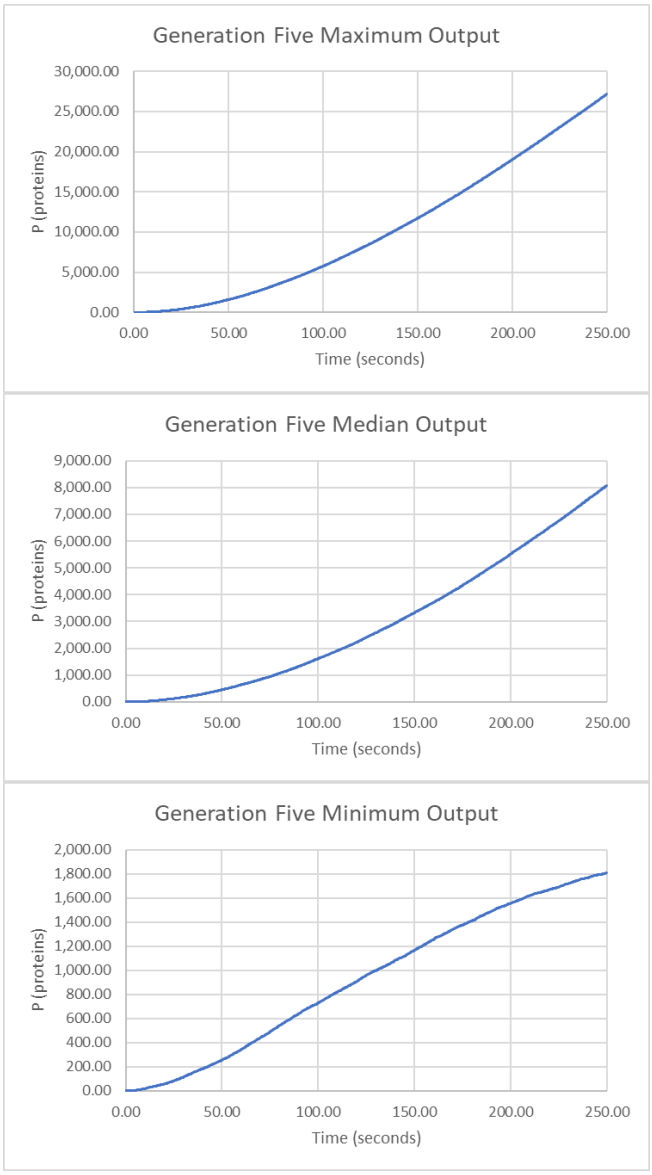

Figure 22: Output plots from fifth generation

The algorithm was run using a generation size of 1500 genomes and iterated over 5 generations. Each simulation was performed 10 times to calculate an average of the output files. The algorithm ran over approximately 9000

seconds or 2.5 hours and created 7500 full output files, as well as a record of the fitness function of every genome in each generation. Maximum and minimum, as well as mean and median averages and the range of fitness values for each generation were calculated. All output files were run through a simple graphing script in python written to process large numbers of files quickly. This graphing process was carried out so the shape of the output graphs could be visually compared to each other and the predicted ideal output graph.

The maximum values of each generation were found by sorting the three highest scoring genomes and the minimum values were found by sorting the three lowest. The three genomes that exist across the median fitness value (the median genome, and a genome one place higher and one place lower). The mean average was calculated by summing the fitness value of every genome in each generation and dividing by the number of genomes, and the range was simply the difference between the maximum and minimum fitness values. Since all generations had a minimum fitness value of 0, the range was always equal to that generations maximum value. Multiple values for the maximum, minimum and median were taken to observe the difference between neighbouring values. If the highest value was vastly greater than the second highest value, it can be assumed that the highest value is somewhat exceptional, however if the highest value is close to the two values below, it can be assumed that the algorithm has consistently produced genomes capable of reaching such a value. All the relevant values extracted from the fitness output files are shown in Table 2.

By graphing the changes in the observed values over all generations of the

algorithm, various aspects of the algorithm's functionality can be evaluated. The graph seen in Figure **??** shows the maximum (equal to the range), mean and median fitness values for each generation. All values show an upwards trend, meaning the algorithm produced increasing fitness results over sequential generations.

The maximum value sharply increases from the first to second, and second to third generations and then appears to remain at a consistent value from the third to fourth, and fourth to fifth generations. The median and mean values show much more gradual increases, but both values consistently show increases between each generation. For both these values, the highest value is reached at the fifth and final generation.

Higher resolution plots of the output protein value against time elapsed were made using the highest of the three maximum genotypes, the lowest of the three minimums, and the exact median. These plots can be seen in Figures 18, 19, 20, 21 and 22. These plots show distinct changes in the performance between generations. Each maximum graph exhibits the same shape but has different measured protein values. The protein values of these genotypes increase for the first three generations but remains approximately equal from the third to the fifth generations. This is consistent with the pattern seen in the plot of the average fitness values. The first generation maximum has a shallower gradient than the following four generations.

The median graphs have greater variation in shape between the generations. The first-generation median shows a seemingly random shape, with sharp peaks and troughs throughout the whole time elapsed. The protein value never exceeds 2.0. The second generation median, similar to the first gener-

ation maximum, has a shallow gradient and the following three generation median graphs have relatively steeper gradients. Each median graph reaches a higher peak than the previous generation.

The first-generation minimum graph shows a flat line across the y-axis (where the protein value is equal to zero) with only two short and sharp peaks. The highest of the two peaks is less than 0.35. The second-generation minimum graph shows an increase in protein value until around the 100 second mark. After this point the gradient sharply decreases and the line dips below the peak and ends on a lower protein value.

The following three minimum graphs have a similar shape to the maximum and median graphs in their corresponding generations albeit with shallower gradients and lower protein values. The peaks of the minimum increases greatly from the first to the second generation then slightly decreases in the third generation. The fourth and fifth generation have much higher peaks than the previous three generations.

5 Discussion

The following section will discuss the significance of the results featured in the previous section and present analysis of the methods used to obtain the results. We first discuss the results of the fitness function over each generation and how these results can be used to determine the effectiveness of the algorithm. We then discuss the raw data output of the KaSim simulations, their physical meaning and how these results could inform applications of the algorithm. We then discuss the methods used in the implementation of the algorithm in the following order: The fitness function, the reproduction function, and the general software and hardware limitations faced during the implementation process. The final topic presented in this section is concerned with potential topics and directs for future research based on the groundwork laid by this algorithm.

5.1 Analysis of Average Fitness Values

The progress of the algorithm over each generation processed can be seen very clearly in the TABLE and FIGURE tracking the average fitness values over all generations. The shape of the Maximum Value curve indicates that the algorithm was relatively quick to find the limit of what was possible with the set of variables it had affect over for the given input value. A fitness value of 2.7×10^4 appears to be the limit of what this algorithm can create. Since the fitness value is only informed by and not directly equal to the measured output of the simulations, it is possible that a higher output value could be obtained using a specific set of variables in the input file.

The consistent increase of the mean and median average values indicates that each generation contained a higher number of successful genotypes than the previous generations. It implies the algorithm would continue to increase these values if it were iterated over further generations. The difference between these two trends can be seen as the algorithm very efficiently finding the peak fitness value, but more slowly improving the general fitness of each generation. During testing, the number of generations over which the algorithm would be iterated was based on this peak value. Test showed that the peak value of the fourth generation was not notably higher than the third, so five generations was selected as it was inferred that no notable changes would take place after this number. It can now be seen that, while there would likely be no changes to the peak value after this generation, the averages would continue to increase. This can be seen as a proliferation of the 'successful genes', in that each generation will only pass in its most successful genotypes, so each following generation will inherit a higher number of genes from high performing genotypes than lower performing (but still non-zero) genotypes. It can be inferred that the average values would eventually plateau at similar values to the peak fitness, since the peak fitness genotypes will consistently pass their genes on more than any other genotypes.

It follows that if the algorithm were run indefinitely, a point would be reached where all genotypes would achieve uniform fitness results, since the rate at which less successful genes are passed on to offspring is always less than that of more successful genes. Due to the random nature of the algorithm, it is also possible that any offspring genotype may inherit a set of genes that is incompatible with the KaSim simulation from otherwise successful parents.

The algorithm finds successful genotypes not through any understanding of what specific genes accomplish so it is possible for the breeding process to follow every rule in its programming and still result in an offspring genotype that is less successful than its parent genotypes.

Using the fitness values as a measure of the effectiveness of algorithm shows that it was able to successfully able to optimise the results of the KaSim simulations to meet the criteria set by the fitness function that was used. The algorithm only took three generations to reach a fitness value that was never meaningfully surpassed by the following generations. This shows that the features of the algorithm that affected the characteristics of genotypes from one generation to the next (the fitness evaluation function and the reproduction function) were effective. The fitness functions effectiveness arises from its sharp prioritisation of the position of the maximum in the KaSim output files. Because the calculated fitness value was directly proportional to the position of the maximum, any genotypes that produced a maximum value at any early point were de-prioritised. The speed at which genotypes with a highest end-value came to dominate the generations was somewhat surprising. Since the relationship between the position of the highest value and the fitness of the genotype is linear, it was expected that the constantly increasing shape seen in the hypothetical ideal genotype would not be reached until a much later generation.

5.2 Analysis of KaSim Output Files

The trend of increasing fitness values is, predictably, also seen in the increasing protein output values of the KaSim output files. The rapidly increasing

and subsequently plateauing maximum fitness genotypes all show the same traits when their corresponding output files are graphed. The peak of the five maximum genotypes is greater in each following generation, with the first reaching approximately 4,000, followed by 16,000 and the next three generations reaching close to 30,000. The peaks of the median genotypes following the pattern of constant increase: the first generation reaches below 2.0, the second below 1,000, the third reaches approximately 4,000, the fourth approximately 5,000 and the fifth 8,000. The minimum genotypes show a different trend when graphed this way. The first generation barely reaches a peak value of 0.3 and the following two (second and third) reach around 350 and 250 respectively. The fourth reaches a peak between 2,000 and 2,500, and the fifth peaks at approximately 1,800.

The minimum values present a notably different trend because they do not show incremental increase as can be observed in the first three maximum genotypes and all five median genotypes. The minimum genotypes appear to make 'jumps' in scale, with the first ending the simulation time with a protein value and the second and third ending on a value that, while high compared to the previous minimum, is vastly lesser than the maximum and median of their respective generations. This likely arises because of the division function used to modify the scale of the fitness values.

All the graphed genotypes except the minimum genotypes of the first two generations and the median genotype of the first generation share a similar shape. They all show a consistent positive correlation between the protein value and the time elapsed. This shape is consistent with the hypothetical idea genotype that the fitness function was specifically designed to recreate

in the reproduction stage. Where the ideal genotype was predicted to produce a linear graph with a constant rate of increase of the protein value along the time elapsed, the actual genotype results rarely exhibit this strictly linear shape. If the previously mention genotypes (first- and second-generation minimums, and first-generation median) are regarded as 'unsuccessful' genotypes for not replicating the ideal genotype in any way, then the remaining 'successful' genotypes fall into two broad categories defined by their gradient, or the rate of change of the protein value over time.

The first of these categories is occupied by many of the earlier occurring 'successful' genotypes (generation one's maximum, generation two's median and the minimum from the final three generations) and it shows a gradient that begins steep, and decreases to a shallower gradient over the course of the simulation. The second category is occupied by the later, more 'successful' genotypes (all remaining unspecified genomes) and shows a gradient that steepens over time. The first group show genotypes that were more successful at maximising the protein value early in the timescale but became relatively less successful as time elapsed. The second group shows genotypes that become more successful at maximum the protein value over time. The second group is more desirable for their ability maximise the protein value effectively.

Detailed numerical analysis of the shape of the output graphs could be carried out during the fitness evaluate such that genotypes exhibiting a 'group two' shape with an increasing rate of change are more highly valued than genotypes with 'group one' shapes. This would likely be unnecessary, as the 'group two' shape naturally came to dominate the results of all late-

generation successful genotypes with no prior instruction for such a process. This is because the features of a genotype that would result in a 'group two' curve are the same features that would result in a higher fitness evaluation using the current fitness function; a higher and later occurring peak.

The 'unsuccessful' genotypes exhibit a variety of shapes that appear to have achieved low fitness values through various means. The first-generation median reaches a low peak in comparison to the maximum of the same generation (2.0 compared to 3,500) and peaks around 75 percent through the simulation time. The first-generation minimum peaks at an even lower value (0.3) and peaks earlier in the simulation time (50 percent through the time elapsed). Bothe these genomes are visibly unsuccessful as there is no upwards trend in the protein value. Both genotypes have clear downward trending sections where the protein value falls as low as zero on some occasions. These two genomes both received a zero for their fitness value. In these cases, this is likely solely due to the low peak value. The second-generation minimum genotype depicts a shape similar to the 'group one' successful genomes in the first half of the elapse time; The protein value increases with a gradually decreasing rate. This genotype differs and becomes visibly unsuccessful after this increase, where the protein value dips below its peak (between 300 and 350) and finishes the simulation time at a lower value (between 250 and 300). This genome did achieve a relatively low peak compared the median of the same generation (300 compared to 1,000) but the key feature that led to a zero-fitness value is the position of the peak. The peak occurs around two thirds through the simulation time, and this early peak position would have required an exceptionally high peak value to push the fitness function out

of the zero-range. The nature of this fitness function cut-off is discussed in Section 3.6.

5.3 Analysis of Methods

5.3.1 Fitness Function

A disadvantage to an analysis focussing on the fitness value is that the fitness function was designed with little knowledge or awareness of nature of the KaSim simulation, and therefore a genotype that achieves a fitness value deemed as 'high' may not produce a meaningful simulation result. The danger of this approach is that the algorithm is optimising the results of a function, and not optimising the actual simulation outputs to reach a desired value. A way to counter this issue would be run multiple further trials of the algorithm with alternate fitness values. Some possibilities for alternate fitness functions include simple variations to the way that the maximum value and its position are weighted. The root or the square of either of these values could be used instead of the whole values, which would replace the linear relationship mentioned earlier with a quadratic or inverse quadratic relationship. There are other features of the KaSim output file that could be incorporated into future fitness functions, such as the gradient of the line plot, or the values of the other two observables that the input file specifies. The two unused observables are 'freeprom', a Boolean measure of whether both sites of the simulation are free and 'M', a numerical measure of the amount of MRNA present in the system. The value of the 'freeprom' variable across the timescale of the simulation could be analysed and the portion

of the time elapsed during the simulation that both sites are free could be used in fitness calculations. This method would be used not to optimise the result of the 'freeprom' variable but instead to analyse correlations between the state of this variable and the value of other variables. Hypothetically, genotypes that produces a high amount of proteins may have similar 'freeprom' data. They may all show that the value is equally often in the 'true' or 'false' value, and this ratio may be favoured when evaluating the fitness of further generations. The MRNA value could be analysed in a similar way to the Proteins value, where the value is plot against the time elapsed. A function could be implemented to analyse correlation between the MRNA value and the proteins value, similar to the proposed 'freeprom' function, and the results could be used to add further fitness criteria.

The problems discussing concerning the 'blindness' of the fitness function due to it existing as an abstraction of the actual simulation output are somewhat inherent to using a genetic algorithm. A genetic algorithm has no way to 'know' what the values it is analysing mean; It can only act based on the criteria it has been provided with. This means that any inadequacies of a fitness function are due to poor design and not due to the performance of the function. The design of the used in this genetic fitness function was clearly effective at maximising a certain kind of result, so it can be classified as a successful function in regards to its purpose, but it is not thorough enough to provide much more information than the characteristics of the genotype that achieves the highest fitness value. Any further research seeking to implement the algorithm into a precise biological system would require some modifications to made to cater the performance of the algorithm to the re-

quirements of the system. The fitness function works as an effective tool to maximise a single output value from a biological simulation so by changing the value to be maximised, the algorithm could be applied to a number of different biological systems. By implementing the additional fitness criteria discussing previously in this section, the algorithm could be implemented in more complex biological systems where something more than a single output value is required to be maximised.

5.3.2 Reproduction

The function that calculates the fitness value finds the product of the maximum protein value in the output value, and then the numbered position of that value along the columns of the file. The maximum value and its position value are then multiplied, and the results is divided by 1,000 to keep the scale of the values down to the integer level for the first generation. This means that any genotypes that have a non-zero maximum-position product can be divided to a sub-integer value and will be treated identically to any genotypes whose maximum-position product is zero due to an inability to create any protein. There will be a cut-off value for the maximum-position product of all genotypes, below which they are all given zero places in the gene pool.

The division was implemented to both limit the number of potential parent genotypes in the gene pool and the encourage only the most highly valued genotypes to be used as parents. The gene pool limiting was implemented since early tests showed that genotypes achieved fitness values in the thousands and with a generation size of 1,500, a typical gene pool could contain

millions of parent genotypes. This was a limitation mainly due to hardware. There are ways to work around this limitation: one of which could involve scaling the fitness values based on the highest value achieved in the generation. This would result in the genotypes being ranked, and each genotype could be added to the gene pool inversely proportionally to its ranking. The genotype ranking 1st would be added 1500 times, the 2nd added 1499 times and so on until the 1500th genotype is only added once. This would keep the gene pool the same size for each generation, though this size would still just exceed 1,000,000. This size of gene pool is considerably greater than what is usual for the initial generations but is much smaller than the following generations. Another method could involve the creation of a gene pool that does not involve creating a data object with millions of components. This could be accomplished by creating an array of reference values, each of which refers to a genotype from the previous generation. Instead of containing all of that genotype's characteristics, each entry in the gene pool array would just by a single string which is then used to search for the genotype with the matching string name to find the characteristics. This may also prove ineffective since the process of searching through every genotype for each entry in the gene pool could prove just as resource intensive as using the unoptimized gene pool storage method.

The other aspect of the reproduction function of the algorithm to be discussed is the function that creates new offspring genotypes from parent genotypes. The method that was used is an example of a crossover algorithm, wherein every value that an offspring value can possess originates from one its parent genomes. The result of using such an algorithm is that the set of character-

istics assigned to the first generation in the initialisation stage of algorithm are the only values that will be present throughout the rest of the algorithm. No new values can be introduced at any point since the new genotype characteristics are always inherited from the previous generation. The problem with this method of reproduction lies in the lack of any diversity in each generation. It is possible although unlikely that none of the initially assigned characteristics will give desirable results. This issue is worked around by using a large generation size [14]. Early tests were carried out using generation sizes as low as 250 genotypes before the population of 1,500 was decided on. With these smaller populations the GA would fail to produce enough genotypes that reached a non-zero fitness score. This meant that there would be no variation between genotypes and generations following the first would contain identical genotypes since many of them had the same parents.

The crossover function, seen in Figure 23, that was used decided which characteristics that the offspring should inherit from the parents on an individual basis. Each characteristic had a discrete chance of inheriting its value from either of the two parent genotypes.

parent a	c1	c2	c3	c4	c5	c6	c7	c8
parent b	d1	d2	d3	d4	d5	d6	d7	d8
offspring	c1	d2	d3	c4	c5	d6	c7	d8

Figure 23: Crossover function used in this algorithm

There are numerous other crossover methods that could be implemented in further iteration on this algorithm [15]. The method in this algorithm is a kind of 'uniform crossover', seen in Figure 24, where each value is deter-

mined individually, but it differs from the usual implementation of a uniform crossover as only one offspring genotype is produced. Usually there is an inverse offspring genotype as well, resulting in two genotypes for every parent pair. The inverse offspring inherits characteristics from the parents that the first offspring does not. If the first offspring inherits the first four characteristics from the first parent and the last four characteristics from the second parent, the inverse offspring will inherit the first four characteristics from the second parent and the last four from the first parent.

parent a	c1	c2	c3	c4	c5	c6	c7	c8
parent b	d1	d2	d3	d4	d5	d6	d7	d8
offspring a	c1	d2	d3	c4	c5	d6	c7	d8
offspring b	d1	c2	c3	d4	d5	c6	d7	c8

Figure 24: Uniform crossover function

There are also single-point, seen in Figure 25 and two-point crossover, seen in Figure 26, operations, where one or two points are selected along the list of genotype characteristics and the offspring inherit from the first parent between the first points, and from the second between the next points. Both of these crossover operations result in two offspring unlike the uniform crossover. operation that was used for this GA.

parent a	a1	a2	a3	a4	a5	a6	a7	a8
parent b	b1	b2	b3	b4	b5	b6	b7	b8
offspring a	a1	a2	a3	a4	b5	b6	b7	b8
offspring b	b1	b2	b3	b4	a5	a6	a7	a8

Figure 25: Single-point crossover function

parent a	c1	c2	c3	c4	c5	c6	c7	c8
parent b	d1	d2	d3	d4	d5	d6	d7	d8
offspring a	c1	c2	c3	d4	d5	d6	c7	c8
offspring b	d1	d2	d3	c4	c5	c6	d7	d8

Figure 26: Two-point crossover function

In future implementation of this GA, the crossover operation could feature the two-offspring feature of the previously mentioned crossover functions, as it would half the number of reproduction operations that would need to be carried out to fully populate new generations. Using Mutation algorithms would account for the lack of introduced between generations. A mutation would involve existing genotypes undergoing random variations. Unlike the other reproduction methods, mutation does no require two parent genotypes. This means that mutation could be introduced after the usual crossover operation is carried out. After the new generation is created each genotype would have a small random chance to undergo mutations on any of their characteristics. With the introduction of a mutation chance, the generation size could be reduced in future iterations in order to reduce processing cost.

5.4 Software and Hardware

There were several limitations presented by the software and hardware available during the development of the algorithm. The first limitation was the lack of compatibility between the available hardware and the software required. KaSim 3.5 was the only piece of software available that could accomplish the goal of the project and due to required software becoming obsolete

and unsupported, it does not run on a windows system. Due to the circumstance under which the project was carried out, work was to be done remotely instead of on a dedicated machine optimised for scientific computation. The first stretch of time spent on the project was taken up installing the Ubuntu Linux operating system to a partition on an available personal computer and tracking down old links to install the required version Ocaml. Once the required software was installed and configured, issues arose related to the limited power of the computer that was used.

The entire GA was programmed on a laptop with limited processing power compared to dedicated scientific computing setups, so the scale was reduced to account for processing time. The final version of the GA took approximately 2.5 hours to process 5 generations of 1500 genotypes. This scale was decided upon after testing that found that adding larger populations or further generations exponentially increased the processing time. Due to physical limitations such as overheating and memory limits of the computer used, any longer timescales were generally unworkable. With access to higher performance hardware, this time could have been greatly reduced which would have allowed a greater scale to be brought to the project. This was not an issue because of the nature of the project; the genetic algorithm was designed to function as a proof of concept to demonstrate the effectiveness of such an algorithm.

6 Conclusion

The previous analysis of the results obtained from the genetic algorithm shows that stated goal of creating "a genetic algorithm that could be used to optimise the output of a simulation of a synthetic biological system to match pre-determined criteria" was achieved. The hypothetical ideal genotype output, while not accurate to scale, predicted the shape of the output graph of the genotypes which achieved the highest fitness values, and therefore predicted the relationship between the measured output value and the elapsed time of the simulation. We analysed the results of the simulations and used various averages of the fitness values to plot the algorithms progress over all generations. We also analysed the raw KaSim output files to determine how the actual data corresponds to the trends found in the fitness analysis.

Analysis of the fitness values shows that the algorithm was very efficient at raising the maximum fitness value per each generation. This value plateaued after only three generations. This is replicated in the plots of the raw KaSim data; the peak of the protein value output vastly increases between the first, second and third generations before staying approximately the same for the remaining generations. The algorithm is notably less efficient at raising the average fitness value of a generation. The median and minimum fitness values increased much more gradually than the maximum but did show an increase between each generation.

We discussed methods that could be used to improve the effectiveness and the scope of the fitness function. One such method involves a more detailed analysis of the gradient of the graphed results. This would add further criteria for evaluating the success of a genotype, meaning a genotype would have

to be successful in even more ways to be considered successful. It would also allow further complexity to be added to the fitness function. By analysing the gradient, more complex graph shapes could be used as the target result. Instead of a simple linear increase, the algorithm could be trained to search for a quadratic increase, or a graph the increases and plateaus or decreases after the peak.

The completed algorithm serves as a proof of concept regarding the use of genetic algorithms for synthetic biological system. The previously discussed alterations could be implemented to extend the scope of this proof. The algorithm could then be applied to more complex biological simulations, the results of which could be used in the actual engineering of real synthetic biological systems.

References

[1] J Carr. An introduction to genetic algorithms. *https://www.semanticscholar.org/paper/An-Introduction-to-Genetic-Algorithms-Carr/e9f8d49686a4c8d99d0a5ceba85c4508c30d57c4*, 2014.

[2] H Holland, J. Adaptation in natural and artificial systems. *The MIT Press*, 1992.

[3] D Shiffman. *The Nature of Code*. 2012.

[4] J Feret and J Krivine. Kappa-dev/kasim. *https://github.com/Kappa-Dev/KaSim/releases/tag/v3.5-190914*, 2014.

[5] Baldwin et al. *Synthetic Biology: A Primer*. 2012.

[6] Python Software Foundation. Python 3.7.4 documentation. *https://docs.python.org/3/*, 2001-2019.

[7] MIT License. Spyder: The scientific python development environment - documentation. *https://docs.spyder-ide.org/*, 2018.

[8] Feret J Boutillier, P and et al. Krivine, J. The kappa language and kappa tools: A user manual and guide. *KappaLanguage.org*, 2019.

[9] Goldstein B et al. Faeder JR, Blinov ML. Rule-based modeling of biochemical networks. *Complexity, Volume 10, Issue 4, April 2005*, 2005.

[10] Theodoropoulos GK Abar S and P. Lemarinier. Agent based modelling and simulation tools: A review of the state-of-art software. *Computer Science Review 24 (2017) 13-33*, 2016.

[11] Doligez D Leroy X and et al. Frisch A. The ocaml system: Documentation and user's manual. *Institut National De Recherche en Informatique et en, 2019*, 2019.

[12] Wine HQ. Wine hq. *https://www.winehq.org/*.

[13] J Feret and J Krivine. Kasim3 reference manual: Release 3.5. *KappaLanguage.org*, 2014.

[14] Gotshall S and Rylander B. Optimal population size and the genetic algorith. *University of Portland*, 2002.

[15] J Magalhaes-Mendes. A comparative study of crossover operators for genetic algorithms to solve the job shop scheduling problem. *WSEAS Transactions on computers, Issue 4, Volume 12, April 2013*, 2013.