Rudimentary Evolution Simulation using a Genetic Algorithm

MComp Honours Computer Science (G405)

Supervisor: Dr Phillip Lord

May 2021

Word Count: 7874

Devon Foister (180282173)

Abstract

Declaration

“I declare that this dissertation represents my own work except, where otherwise stated.”

Acknowledgements

Table of Contents

Section 1: Introduction

1.1 Topic

Newcastle University CSC3423 Genetic Algorithms Lecture [1]. This lecture gave me the inspiration to start this project and goes in depth into the history of genetic algorithms (GA) as well as in depth information on different selection methods, mutation/crossover methods and many different use cases. Along with the speaking aspect of the lecture, this gave me a solid groundwork to begin planning my own GA.

1.2 Motivation

The motivation for this project began after I had taken a module at Newcastle University, titled Biologically Inspired Computing. After reading through the lecture notes for genetic algorithms [3] I wanted to see the algorithm used in the context of natural evolution. At this point, I had no idea of the extent and reach of evolutionary algorithms and just found the subject interesting. Around this same time, I was still struggling to come up with a topic for my third-year dissertation. This is when the idea of creating an evolution simulation was spawned. I then had the idea of making it interactive in order to showcase what a genetic algorithm does and make the program enjoyable to use for others who are interested in genetic algorithms. I know that the project is possible due to similar concepts being achieved by others such as Nathan Rooy [4] who managed to achieve a similar result, with the use of neural networks. Here is a screenshot from Rooy's simulation.

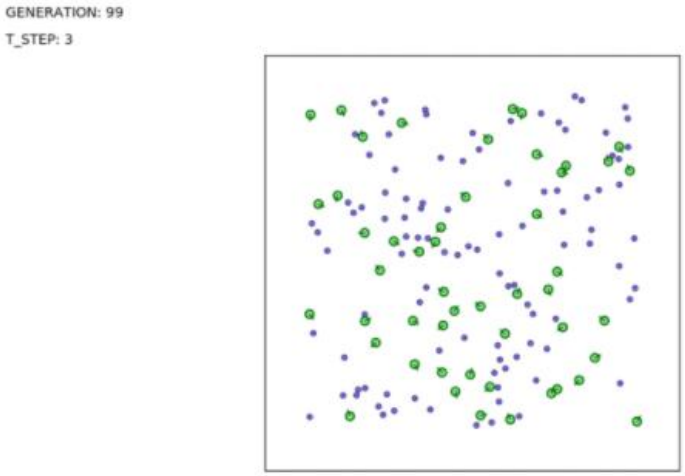


Figure 1.2 - Nathan Rooy's Solution [4]

1.2.1 The Problem

When choosing to study biologically inspired computing, I was expecting the topic to be more based around nature itself, and simulating nature within the scope of computing. Mainly due to my own poor research, I found this was not the case, so I decided to relate it more to nature myself, learning about evolution and genetic algorithms in the process. The genetic algorithms I have used so far involve simply changing parameters and watching numbers pop up in a console. I did not find this very engaging and thought it would be more interesting if I could actively see what the algorithm was doing over several generations. This could make the subject more interesting for those who want to learn about genetic algorithms, those with an interest in biologically inspired computing or people who find the idea of evolution and tiny virtual organisms interesting.

1.2.2 My Approach

Similar projects, such as one by Nathan Rooy [4], tend to ignore negative factors in an organism's environment such as predators or poison. This project will, at the very least, include poison in the environment to add another factor for the organism to adapt to. My project will also be interactive, allowing the user to alter the genetic algorithm to produce vastly different results. This could be used in teaching to provide a more exciting approach to genetic algorithm optimisation. I will be using C++'s graphics library with the help of a manual from Stanford University WRONG [4] to display the entities on screen along with information pertaining to the algorithm. I will be using a collection of external resources [5] as well as University lecture material [6] in order to detect and handle collisions between organisms and food/poison, affecting the organism's health accordingly. Each organism will have a chromosome, comprising of several genes. These genes will contain important information on the organisms, such as size, speed, food/poison perception radius. There will also be random variables used in movement calculations that gage the general intelligence of an organism. Each generation, the healthiest organisms will be carried over to the next generation, along with a selection of their offspring, generated from the genetic algorithm. The end result should be a self-sustaining colony of organisms.

1.3 Aims & Objectives

1.3.1 Aim

Explore how a genetic algorithm can be manufactured to simulate evolution amongst artificial organisms, providing an interactive interface where parameters can be altered to adjust the effectiveness of evolution.

1.3.2 Objectives

1. Theorize and create a genetic algorithm that increases the calculated fitness of an organism, effectively evolving it.

This is crucial to the entire project and will need to be completed early on to ensure the rest of the objectives are met on time.

2. Implement a GUI which displays organisms, other entities, and the current statistics of the genetic algorithm.

This is one of the more straightforward objectives but utterly new territory for me. Once implemented, it will not need to be modified much at all.

3. Create a movement system whereby organisms choose where to move based on their size, quality of their vision, and nearby food and poison positions.

The movement itself will be easy to achieve but calculating how to move the organisms will take a lot of research and be based on the organism's genes.

4. Create a collision detection system that allows organisms to interact with food/poison entities and have their health affected accordingly. This will be visible in the program.

This is important for the organisms to be able to react to the environment around them.

5. Program a death and reproduction cycle which will remove the previous organisms, reproduce them using the genetic algorithm and create a new set of organisms with new chromosome values.

This is key for the actual evolution to occur as there will need to be many generations for the organisms to evolve and become more suited to the environment. The genetic algorithm will manage the reproduction.

6. Introduce an interactive component to the GUI that would allow a user to modify the algorithm's specific parameters to change how fast or effective the evolution of organisms is.

This is another crucial objective as it makes up the entire interactive portion of the project. This will most likely consist of sliders for mutation/crossover chance and the option to change the selection method the algorithm uses.

1.4 Project Schedule

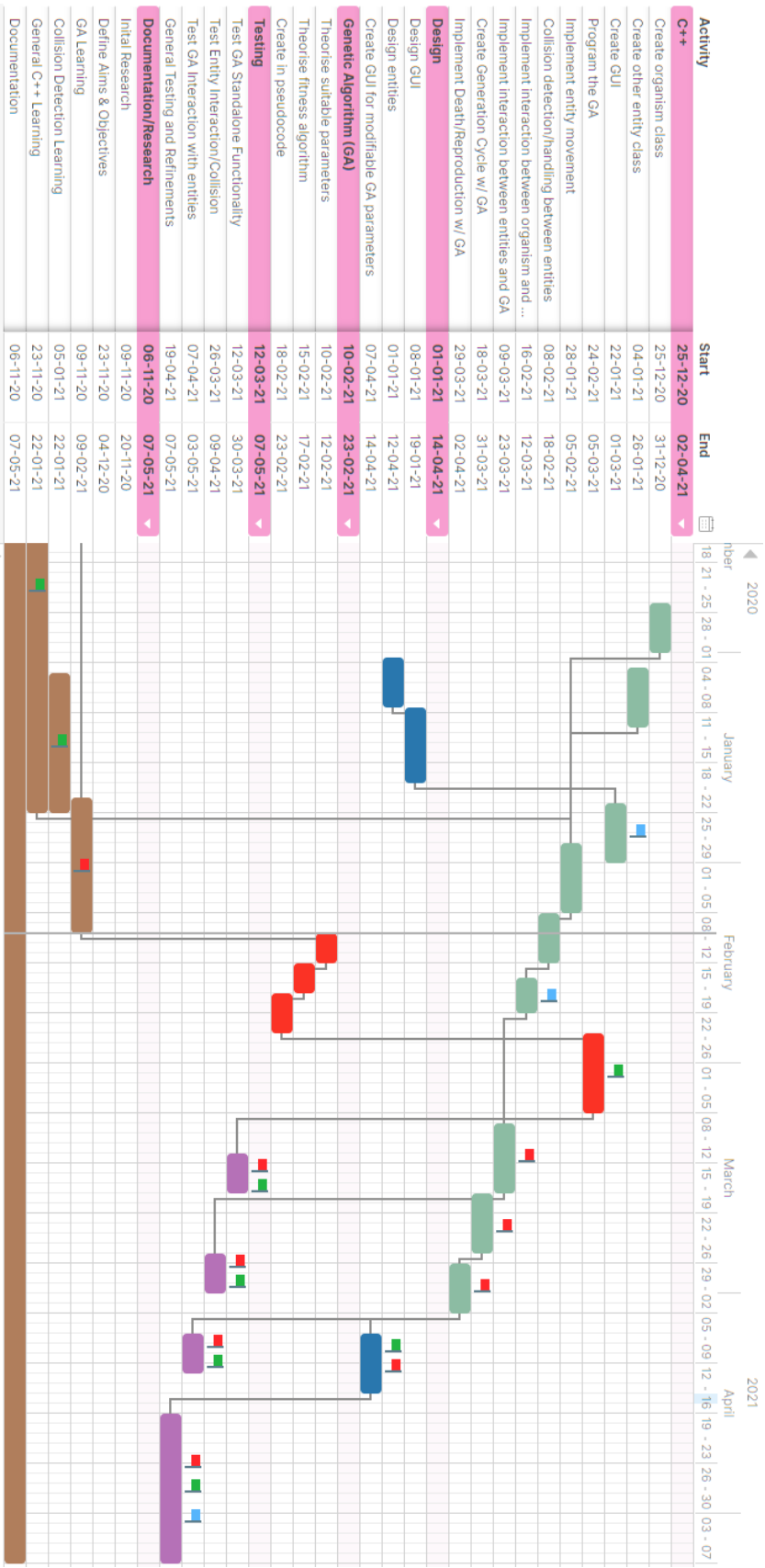


Figure 1.3 Project Plan Diagram

1.4.1 Schedule Explanation

The diagram above details the project plan that I have used throughout the course of the year, beginning in November 2020, and ending in May 2021. The diagram was created using tomsplanner.com [X] in the early stages of the project. It is split into 5 sections:

1. **Programming**

   Any task that involves using Visual Studio and creating the actual simulation in which this project is aiming to produce. This will often have overlap with the genetic algorithm and design section, indicated by a flag of the respective colour of these sections.

2. **Genetic Algorithm**

   This section is for any task related to the genetic algorithm used on the project. The algorithm needed to be built from the ground up, tailored specifically to work hand in hand with the C++ program. This meant careful planning was needed, as well as achieved in a suitable time for the rest of the project to run according to plan.

3. **Design**

   These tasks relate to general designing of aspects of the project. This ended up being a very small part of the project in the end as I kept the visual front of the program very simple to spend more time on the complicated backend.

4. **Testing**

   Any task related to ensuring correct functionality of the program and algorithm. This was a long and important part of the project as every single algorithm method and aspect of the physics engine had to be tested. Although a lot of testing was done at the tail end of the project, testing was continuous throughout due to adopting an Agile development methodology.

5. **Research/Documentation**

   Constant research and documentation have taken place throughout the lifespan of the project. Despite there being start and end dates for each of the learning tasks, I have been continually learning new things since beginning this dissertation. Documentation has been frequent, with me writing down how I am implementing features as I create them and keeping the aims and objectives noted down ready for the final writeup.

As you can see from the dependency arrows in Figure 1.1, nearly every piece of the project had to be completed in a set order, or else continuing would become impossible. Later I will discuss how this affected the rate in which the project was completed and whether I would change my approach in the future.

Section 2: Background

This section of the

2.1 Research Sources

2.2 Genetic Algorithms

2.2.1 Overview

To understand genetic algorithms, you need to take a step back and first look at evolutionary computing. Evolutionary computing is a research area within computer science, inspired by the process of natural evolution. Although you probably have a solid grasp on the concept of evolution, the Introduction to Evolutionary Computing describes it as follows:

> A given environment is filled with a population of individuals that strive for survival and reproduction. The fitness of these individuals is determined by the environment, and relates to how well they succeed in achieving their goals. In other words, it represents their chances of survival and of multiplying. [X]

This style of problem solving is known as trial-and-error or generate-and-test, where you have a collection of possible solutions of varying qualities that establish how close they are to the ideal solution of a particular problem. Depending on their quality, they are then used as a basis for creating new solutions. This step is repeated until, ideally, a perfect solution is found. Most of the time it is more likely to be a very good solution, rather than perfect. There is a clear link between evolution and this method of problem solving.

Environment ←→ Problem Individual ←→ Possible Solution Fitness ←→ Quality

Genetic algorithms in particular follow a very specific structure. The algorithm starts by generating the initial population of individuals. These individuals each possess a chromosome, which in turn contains a selection of genes. As these terms will be used frequently throughout this paper, here is a diagram to aid in understanding. In this context, A1 to A5 are the individuals.

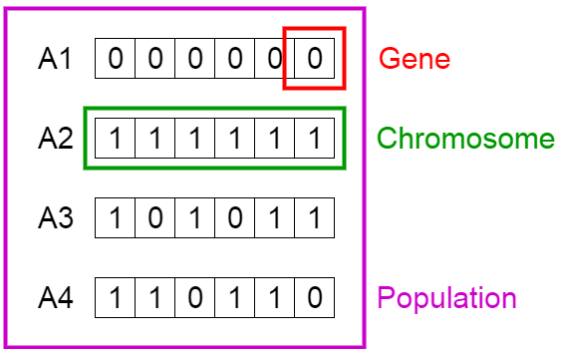


Figure 1.1 – Population, Chromosomes and Genes [X]

Next, each individual has their fitness calculated which ranks them depending on their ability to solve the given problem. The selection phase is next which selects the fittest individuals to have their genes

passed on to the next generation. This process places the selected individuals into pairs, which then become parents.

The next two stages in the algorithm are the reproduction stages, the first being crossover. Crossover takes each pair of parents and creates offspring by combining their genes to create new chromosomes. It is called crossover as this switching of genes normally takes place at different places, dependent on one or many crossover points. This concept and its different variations will be discussed later. Mutation is the next step in the algorithm. For each new offspring, there is a low probability for each of its genes to be mutated. This could be as simple as flipping a bit or inverting entire subsets of chromosomes. Once again, this will be expanded on later.

This new population of individuals will then have their fitness values calculated and the evolution cycle continues. The algorithm will have a termination criterion that is activated when either a suitable solution is converged upon, the population has not improved for X generations or a specified generation limit has been reached.

2.2.2 Algorithm Selection

Population Management

There are two key population management models for genetic algorithms, the generational model, and the steady-state model. In the generational model, the entire population is replaced by offspring at the end of each generation. Most simple genetic algorithms use this model, keeping the population size, mating pool size and offspring size the same, to make replacing the entire population simple.

With the steady-state model, only part of the population is replaced by its offspring, each generation. The proportion of the population that gets replaced is known as the generational gap. As these models are performed based on a population's fitness, either can be used independent of what the chosen problem is.

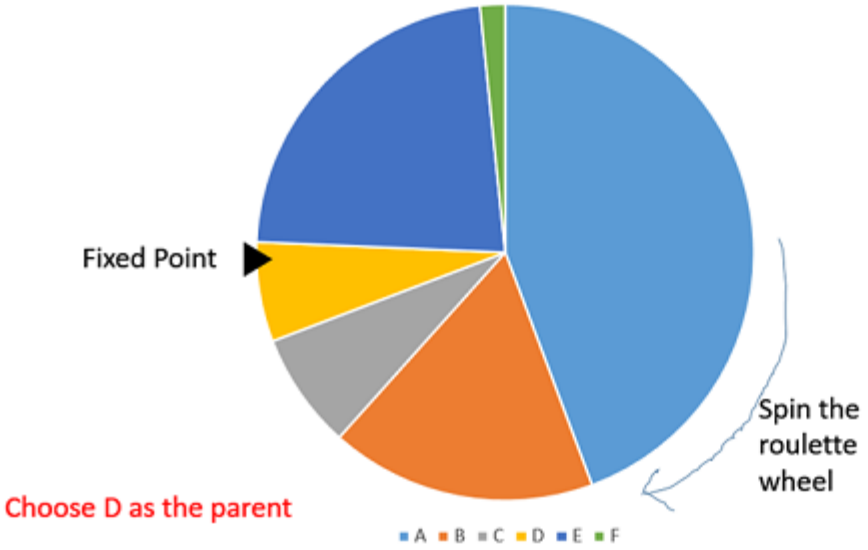
Parent Selection

There are also two types of parent selection, fitness proportional selection (FPS) and ranking selection. FPS is where the probability of an individual being selected for mating depends on its fitness compared to that of the rest of the population. This is the most common method used but has some drawbacks. Individuals with particularly high fitness values, especially early in the algorithm tend to take over the population as they are overly selected. This reduces the chance that the algorithm will search through a large space of possible solutions and is known as premature convergence. There also exists another issue that if fitness values are very close throughout the population, selection pressure is lacking, resulting in almost random selection. This has been shown to cause very slow fitness increasement in later generations of the algorithm. One solution to this issue is called windowing, where the fitness of the least-fit individual is subtracted from the fitness of the rest of the population.

Ranking selection was created due to the drawbacks of FPS. The idea is that it keeps a constant selection pressure by ranking the population based on fitness and then assigning selection probability based on rank, rather than the fitness value itself. This fixes both key concerns with the FPS approach.

Selection Methods

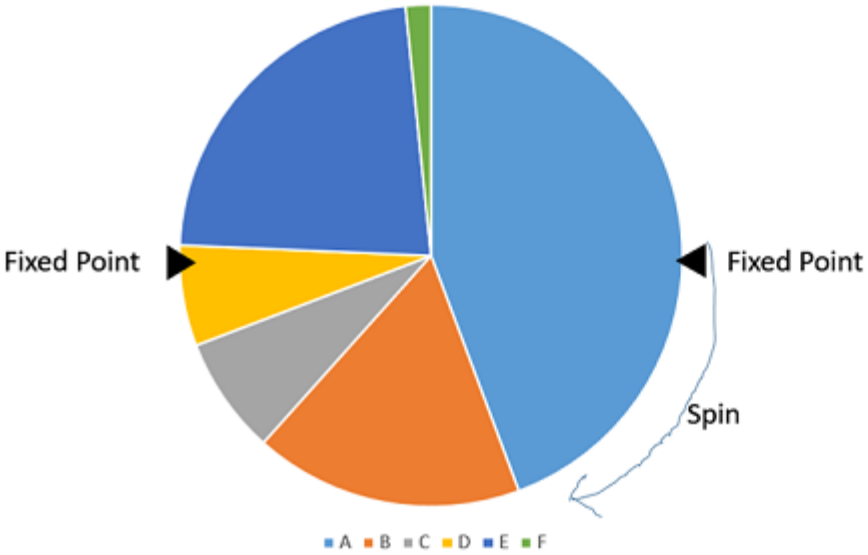
There are many selection methods for genetic algorithms which carry out the actual selecting process. One of these and perhaps the simplest method is roulette wheel sampling. The idea is you spin a wheel with the population on, the size of each individual's section dependent on their fitness or rank. Whatever individual the wheel lands on, will become a parent. Below is a diagram demonstrating this.



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Figure 2.1 – Roulette Wheel Selection [X]

Similar to roulette wheel sampling is stochastic universal sampling (SUS), which is also a wheel but has multiple fixed points, allowing multiple parents to be selected at once and increasing the chance of an individual with high fitness being selected. SUS is demonstrated below.



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Figure 2.2 – Stochastic Universal Sampling [X]

Another commonly used selection method is tournament selection. N number of individuals are chosen from the population randomly. These N individuals are then compared against each other, based on fitness, and the fittest individual is selected to become a parent. This is then repeated for each parent. In an alternative version, there can be multiple tournament survivors.

There is also random selection where parents are selected from the population completely randomly. As there is no selection pressure this method is mostly avoided but still available as an option.

2.2.3 Algorithm Crossover

Crossover is directly related to biological reproduction where two parents are selected and offspring are produced using the genetic material of the parents. Crossover is usually applied with a very high probability and in many cases the probability is one. There are many crossover methods, with some being more suited to this project than others. Two simple operators are single point crossover and multiple point crossover.

One point crossover involves taking a random index within the chromosome and creating a splitting point at that index. From there both parents are split, and the children are created by swapping the tails of the two parents from that index onwards. Multiple point crossover is very similar but this time the parents are split at multiple points along the chromosome. The children are then creating by alternating the segments of the parent chromosomes in each child. The following diagram explains this quite well.

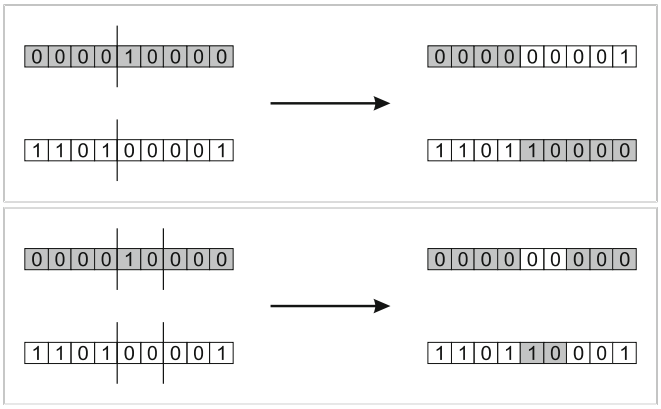


Figure 2.3 – Point Crossovers [X]

Uniform crossover is another method commonly used. For this method, each gene in the first offspring has an equal chance of being inherited from either parent. This happens for each gene in the first offspring and then the second offspring is created from the inverse of the first. Once again here is a diagram.

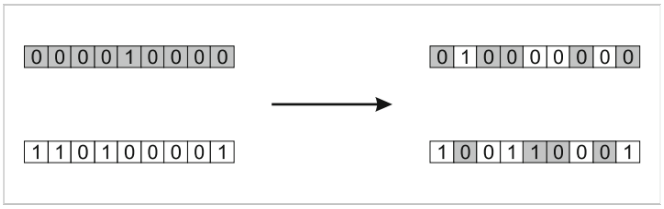


Figure 2.4 – Uniform Crossover [X]

2.2.4 Algorithm Mutation

Mutation in genetic algorithms is used to keep diversity in the population and is essential for convergence. Mutation is also applied with a low probability to ensure that the algorithm does not just become a random search. Once again there are many mutation methods that can be used.

Bit flip mutation is the simplest form of mutation and is used for chromosomes encoded in binary. It is as simple as it is named, with genes being flipped between one and zero. This would not be applicable for this project as the chromosomes are encoded with real values. Random resetting is similar but meant for other encoding methods. In this method, the gene is swapped with a random value created from a set of possible, defined values.

Creep mutation works by adding a small positive or negative value to a gene. The size of the step is dependent on the size of the fitness values as you only want to make small steps. The distribution should be symmetric, centred around zero, to ensure the value can move the same amount positively or negatively.

Scramble mutation and inversion mutation are very similar mutation in the sense that they would be implemented similarly. Scramble mutation takes a subset of the chromosome and shuffles the genes within the subset. Inversion mutation also takes place on a random subset of the chromosome, but instead reverses the order of the subset. Swap mutation involves swapping two genes in the chromosome randomly. The problem with these methods is that they don't introduce new values into the chromosome and if used exclusively, the entire population will keep recycling the same genes. It could be effective to combine one of these methods with random resetting or creep mutation.

2.2.5 Algorithm Survivors

Genetic Algorithm: A Tutorial Review [6]. This short paper outlines the features of a genetic algorithm as well as the basic pseudocode for a functioning algorithm. Each component of a GA is described in detail, such as the selection methods and crossover/mutation methods for reproduction. It also outlines the different GA expressions and explains each one. I used this resource for my early research but still come back to it occasionally as a reminder of certain aspects.

Evolving Simple Organisms using Deep Learning from Scratch [2]. This blog post which I found quite recently tries to solve a similar problem in scratch using a neural network. This was helpful as it showed, at the very least, that the problem is solvable. It also led me to an interesting question. How am I going to make an organism better or worse at navigating to the food? The author used a neural network and used a normalized value representing the direction to the nearest food as an input. The direction the organism turns is then calculated within the neural network. Reading his blog, I ended learning about neural networks and their usefulness in a situation like this. Although I do not plan on using neural networks, I believe I can create a similar result for determining the angle an organism rotates. I can use a simple function to determine the angle from where the organism is facing to the food/poison, then calculate a turning angle based on the values stored in the organism's chromosomes.

2.2 Physics Simulation

As this project is supposed to simulate organisms in an environment, I felt it was important to incorporate real physics into the simulation. This will make the organisms more realistic and make the movement of each one look more natural. One way of achieving this is by implementing, and forcing organism to abide by, Newtonian physics. Newton's three laws of motion are as follows.

2.2.1 First Law

"When viewed in an inertial reference frame, an object either remains at rest or continues to move at a constant velocity, unless acted upon by a force." [X]

This first law introduces the idea that an object moves after a force is applied to it. A force has a direction and a magnitude, which is measured in Newtons. Newton's first law is also known as the law of inertia, inertia being resistance of any physical object to any change in its velocity. A still object resists any force trying to move it and a moving object resists any force trying to change its

velocity. Any object in a simulation displays inertia based in its mass, the higher the mass, the harder it is to move with a force.

2.2.2 Second Law

"The vector sum of the forces *F* on an object is equal to the mass *m* of that object multiplied by the acceleration vector *a* of the object: *F = ma*." [X]

It makes more sense in this simulation to rearrange the equation as (a $= \frac{F}{m}$). This is because each organism will have a known weight value as well as a force to be applied based on its decisions and genes. This means the change in velocity for each organism can be calculated from the force applied divided by the mass of itself.

2.2.3 Third Law

"When one body exerts a force on a second body, the second body simultaneously exerts a force equal in magnitude and opposite in direction on the first body." [X]

Essentially and well known as, for every action, there is an equal and opposite reaction. In the context of this project, it basically means that the objects within the simulation pass force between one another, if two organisms hit each other, they will not just stop but act according to the momentum each one is carrying.

Momentum is represented as $p = mv$. Due to the law of conservation of momentum, the total momentum of a system of objects remains constant. Despite the objects exerting equal and opposite forces onto each other, the velocities do not change by the same amount but instead change proportionally to their mass, conserving their momentum.

2.2.4 Calculating Movement

From here onwards, I will refer to aspects of the simulation in the context of this project. For example, objects are now organisms. The SUVAT equations of motion helped me to understand how motion over time is calculated and how the positions of organisms within the simulation could be manipulated. SUVAT stands for the five variables of linear acceleration:

- S = Displacement (how much an organism's position has changed by).
- U = Initial velocity.
- V = Final velocity.
- A = Acceleration.
- T = Time.

The SUVAT equations allow you to work out any of the five variables based on the values of the other variables. To demonstrate this, the first equation is as follows, $v = u + at$. This means that if you know the constant acceleration (m/s/s) and time then you can work out how much the acceleration affects the overall velocity of an organism. There is a problem with SUVAT equations in that they are dependent on there being constant acceleration or velocity. In this simulation, organisms move in different ways relative to the current conditions of their environment. This means that these factors are never constant. Calculus is needed to work out how the positions of the organisms should change, starting with derivatives.

- Position is noted as s.
- Velocity is the first derivative of position with respect to time, noted as $\dot{s}$.
- Acceleration is the second derivative of position with respect to time, noted as $\ddot{s}$.

Using these derivates and the equation $\dot{s} = \frac{ds}{dt}$, you can work out the velocity of an organism if you know its positions at two points in time. Here is an example:

Organism position at 0.5 seconds = [5,0]

Organism position at 3.5 seconds = [35,0]

$$\dot{s} = \frac{[35,0] - [5,0]}{3.5s - 0.5s} = \frac{[30,0]}{3.0s} = [10,0] = 10\ units/second$$

This equation does not account for whether the velocity varied between 0.5 seconds and 3.5 seconds, only giving the average velocity. This is solved by repeating the equation many times a second, referred to as the hertz (Hz). Many game physics engines update at anywhere from 60Hz to 240Hz. Even those running at 240Hz may not be perfectly accurate but are enough to simulate physics to a respectable degree. Once again there is a problem with this approach in respect to this project. We need to calculate an organism's new position based on its changing velocity, whereas this approach calculates velocity based on positions. The next step to solving the problem is integration.

Integration is used to determine how changes in an organism position's derivatives effect its position over time. The integrals of the position's derivatives are ($\dot{s} = \int \ddot{s}dt$ and $s = \int \dot{s}dt$). dt in this context is the time between each sample (timestep). In a simulation dt is the time between each frame, or $\frac{1}{Hertz\ of\ simulation}$. There are several integration methods that can be used starting with Euler's Integration.

$$v_{n+1} = v_n + a_n dt$$

$$s_{n+1} = s_n + v_n dt$$

These values are calculated each update cycle, taking the values from the current cycle (n) and adding the derivatives, multiplied by the timestep in order to calculate the next cycle's values (n+1). The problem with this method is that velocity is treated as a constant value during a timestep whereas the velocity should be constantly changing due to the integration of acceleration. This ultimately results in a loss of accuracy over time. There is an alternate method, known as Backward Euler, which calculates the derivative at the next timestep rather than the current timestep. This is shown below.

$$v_{n+1} = v_n + a_{n+1} dt$$

$$s_{n+1} = s_n + v_{n+1} dt$$

This method has a glaring problem for a real-time simulation in that it needs to know the velocity and acceleration from the next timestep, which is information you do not have. Fortunately, there is a third option which provides more accuracy over time while not forcing you to predict the future. This is known as 'Symplectic Euler'. Here, acceleration is integrated using the current cycle values and velocity can then be integrated more accurately relative to the current state of the organism's motion.

$$v_{n+1} = v_n + a_n dt$$

$$s_{n+1} = s_n + v_{n+1} dt$$

Having a small timestep is very important when using any of these integration methods. The result of the calculation will never be perfect and having a small timestep can help to minimize the inaccuracy

of the simulation over longer periods of time. The following example, provided by Richard Davison's game physics tutorial [X], demonstrates how different timesteps used in Symplectic integration can affect how positions are calculated:

> We know from the SUVAT calculations that after 2.4 seconds, a constant acceleration of 2 m/s should result in an end position of:
>
> $$s = ut + \frac{1}{2}at^2 \quad \rightarrow \quad s = 0 * t + \left(\frac{1}{2}(2 * 2.4)\right)^2 \quad \rightarrow \quad s = 5.76$$
>
> Using Semi-Implicit Euler (Symplectic) integration results in the following positions:
>
> - Position using timestep 0.01 = 5.784
> - Position using timestep 0.1 = 6
> - Position using timestep 0.2 = 6.24
> - Position using timestep 0.4 = 6.72
> - Position using timestep 0.8 = 7.68

Richard then goes on to describe how a 60Hz or 60 frames per second simulation, which equates to a timestep of 0.0166 seconds would be "fairly accurate" but would still lead to the position being inaccurate after a decent amount of time. This suggests that physics engines should aim to run at as many hertz as possible while still maintaining adequate performance.

2.3 Collision Detection and Resolution

2.3.1 Overview

Collision detection is extremely important in many computer systems, being a "fundamental problem in computer animation, computer graphics, physically-based modelling, and robotics." [X] It is easy to think that if you have two circles on the screen, the computer should know whether or not they are touching or not but instead a series of equations is needed, as well as different types and phases of collision detection for the purposes of accuracy and efficiency. Anytime the mouse is moved in a program or webpage there is most likely a function in the background tracking exactly where the mouse is and whether it is within the bounds of another element.

Just as important is collision resolution. As an example, if you watched a simulation of two cars colliding at high speed you could easily tell at what point the cars collided. However, there is a good chance that you would not be able to tell exactly how the cars would act afterwards. Many factors need to be considered. How fast was each car going? What direction was each car travelling? What material are they driving on? How much does each car weigh? The job of the simulation in this case, would be to take all these factors into consideration and calculate the trajectory of each car after the collision.

There are three types of collision detection needed for the project, rectangle to rectangle, circle to circle and rectangle to circle. There are multiple methods for collision detection and for each I would need to also get the penetration distance between the two objects and the direction of least penetration.

Jeffrey Thompson's collision detection website [4] provided me with an extensive resource for collision detection between different basic shapes. Each example is well explained, readable and easy to understand. He explains that there are "more efficient ways to detect these collisions" [X] and that

the principles are taught with minimal math. This was a great starting point and provided interactive samples with each section.

2.3.1 Circle/Circle

To calculate whether two circles are colliding you need to first get the distance between the centres of both circles. This is achieved by the following calculation, with circle A and circle B.

- DistanceX = A Centre X – B Centre X.
- DistanceY = A Centre Y – B Centre Y.
- Distance = Square Root (DistanceX^2 + DistanceY^2).

You then check whether 'Distance' is less than or equal to the sum of the radii of the circles. If this is true, then they are colliding.

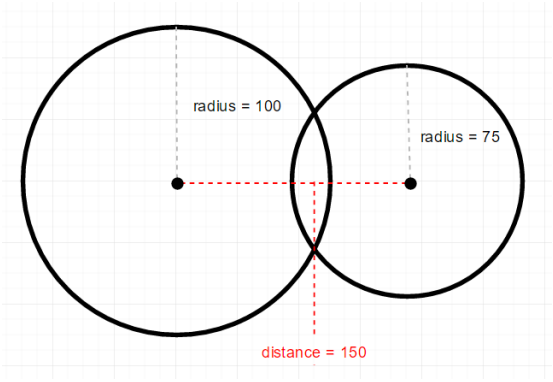


Figure 2.5 – Circle/Circle Collision [X]

2.3.2 Rectangle/Rectangle

If you have two rectangles A and B, here are the checks you must do to detect whether they are colliding.

1. Is the right side of A to the right of the left side of B?
2. Is the left side of A to the left of the right side of B?
3. Is the bottom side of A below the top side of B?
4. Is the top side of A above the bottom side of B?

If all of these are true, then the two shapes are colliding. Although not easy to follow by reading, it would be easy to implement in C++.

2.3.3 Rectangle/Circle

For this collision method, you need to first work out which side of the rectangle is closest to the circle and then you carry out the following tests.

1. If the circle is to the right, check against the right edge of the rectangle.
2. If the circle is to the left, check against the left edge of the rectangle.
3. If the circle is above, check against the top edge of the rectangle.
4. If the circle is below, check against the bottom edge of the rectangle.

Simple enough. Once you know which side you need to check you use the Pythagorean Theorem to calculate the distance between the centre of the circle and the edge of the rectangle closest to the edge. Here is how this value is found:

- DistanceX = Circle Centre X – Left/Right Edge (dependent on previous tests).

- DistanceY = Circle Centre Y – Top/Bottom Edge (dependent on previous tests).
- Distance = Square Root (DistanceX^2 + DistanceY^2).

You then check whether ‘Distance’ is less than or equal to the radius of the circle. If this is true, then they are colliding.

2.1.4 Graphical Libraries in C++

Stanford graphics library manual [3]. Stanford University has a publicly accessible manual for the graphics library in C++. The website contains detailed descriptions on each function in the library.

2.1.5 C++ Development

CSC3221 Programming for Games Lectures [5]. These lectures provided me with solid C++ knowledge and groundwork for me to build upon. I already had quite a good grasp on object-oriented programming fundamentals, but these lectures and notes helped to reinforce this.

2.4 Summary

Section 3: Design & Implementation

3.1 Introduction

3.2 Planning

3.2.1 Genetic Algorithm Planning

Although I wanted the algorithm to be customisable by the user, there were a couple of factors that either could not be changeable or I felt were not necessary for the project. One of these factors was the population initialisation. The two main methods for this were random initialisation and heuristic initialisation.

The problem with the heuristic approach was that there is not a known heuristic for the problem I am trying to solve. I also did not want a fully random approach as I soon found out while experimenting with the early stages of the simulation, extreme values can ruin the entire simulation. For example, if one organism has an incredibly high speed and size it will hit other organisms, in turn, pushing them towards or away from food and poison they otherwise would have found. Although realistically this could happen, this population is specifically designed to ignore each other. The approach I went with was to use a mixture of both. I experimented with a reasonable range of values that a gene could have to begin with and then the population was initialized with a range of these semi-random values.

The other factor I addressed before the project took place was the population model for the algorithm. The two most popular approaches are the steady state model whereby just a couple of organisms are replaced in each generation with new offspring, and the generation model where the entire population is replaced each generation. I found that the difference between the choices was too drastic, with steady state being too slow for a problem where generations can last over a minute long. Meanwhile, generational runs the risk of deviating away from the best solution by removing potentially perfect organisms. Once again, I compromised to a solution, I believed would be better. By default, each generation would keep 50% of its population through the selection process and new offspring will be created from this 50% through crossover and mutation.

I had to also decide on a fitness function suitable for this problem. Initially it would just take the populations current health values when the generation ends and use that, however I ran into an issue. What if the entire population died before the generation ended? Slightly less drastically, what if any 10% of the population died before the generation ended. It would not make sense that two organisms that died at different times would have the same fitness value. So, I decided to track each organism’s lifetime and current health, combining these to create the fitness value. This means that dead organisms will be ranked by the time they stayed alive and living organisms will be ranked by their current health.

3.3 Tools & Technologies

3.3.1 C++17

Before I began this project, I was not sure which language I was going to use. Java was my preferred and most used language, but I had recently started using C++, and there were a few things I liked compared to Java.

C++ is a compiled language, meaning it is much faster than other languages and offers much greater control due to being a strongly-typed unsafe language. As cplusplus.com forum user, Albatross states:

> C++ is a language that expects the programmer to know what he or she is doing, but allows for incredible amounts of control as a result. [x]

The language also supports dynamic type checking, which has helped create the collision classes.

Java's garbage collection system constantly checks whether memory is still in use, whereas C++ trusts the programmer only to allocate and use what they need to. In a simulation where there are many calculations taking place every frame, this is rather important.

SFML, which provides the GUI for the project, was also primarily created for the C and .NET languages, which helped in the decision-making process.

3.3.2 Visual Studio 2019

Visual Studio (VS) was my IDE of choice as I have used it many times for previous projects I have undertaken. The inbuilt IntelliSense feature has saved hours that would have been spent searching through class documentation by providing a robust code-completion aid. The debugger is also the best I have used and has been very useful to me throughout the year. Git integration out of the box has also made source control and backing up the project a swift process.

3.3.3 Eigen

The Eigen header library was used for Vector2 representation and calculations. This was needed for representing positions and the forces used on elements of the simulation. I was tempted to create my own vector class; however, it seemed more sensible to use a simple header library and fill in the blanks where needed as the time would be better spent elsewhere.

3.3.4 SFML

Simple and Fast Multimedia Library provides the ability to apply a simple interface to Windows applications to help with the development of games. There are five modules within the library, of which I used three. These three were the system, window, and graphic modules.

3.4 Application Development Methodology

As I am constantly learning and applying new knowledge to the project, I felt the waterfall approach would not be an effective methodology

3.5 Development

3.5.1 The Algorithm

The algorithm is called at the end of a generation within the main evolution simulation class. It is constructed with multiple arguments, including:

- A vector of all organisms in the population.
- An enumeration to determine the selection type.
- An enumeration to determine the crossover type.
- An enumeration to determine the mutation type.

The constructor of the algorithm class then sets the respective properties from these parameters. The constructor then calls the 'computeAlgorithm()' method which calls each algorithm function in the following order.

sortPopulationByFitness

This method simply sorts the population vector by their fitness values, which are calculated by adding together the lifetime of the organism and its current health when the generation ended.

selectionProcess -> crossoverProcess -> mutationProcess

These methods check the corresponding type properties and call the respective function. For example, if the mutation_ parameter was set to swap mutation, mutationProcess would call the 'mutationSwap()' function.

```
if (mutation_ == MutationType::SWAP) mutationSwap();
```

The crossover process method also randomises the selected population and pairs them, in preparation for mating via crossover.

createNewPopulation

selectionRouletteWheel

This function keeps a variable can contains the total of all fitness values from the population. A random value between zero and the total fitness is then created. Next, a loop goes through the population, adding their fitness value to a temporary variable and comparing this temporary variable against the random value. If the random value is less than or equal to the temporary variable, this organism is added to the list of organisms to be mated and placed in the next generation.

selectionTournament

This function first loops through the number of participants in a tournament (the tournament size) and adds that number of random organisms from the population to a temporary vector. This vector is then sorted by the fitness values of the organisms within. Finally, the N fittest organisms from that vector are added to the list of organisms to be mated and placed in the next generation, where N is the number of survivors. Both the tournament size and survivor amount can be modified.

selectionRandom

This is quite a simple method. It calculates 10 random, unique numbers between 0 and the population size. These are used as indexes for the population vector. N number of organisms from the population are then added to the new list of organisms, based on the random indexes. N in this case would be the pre-determined size of organisms to be selected for crossover and mutation.

crossoverUniform

The chromosomes from both organisms in a mating pair are looped through. For each gene, in both chromosomes, there is a 50% chance they are swapped in the offspring. A diagram detailing from tutorialspoint.com is below. [x] These offspring are then placed, with the parents, in a vector ready to be passed to the mutation function. This process is repeated for each mating pair.

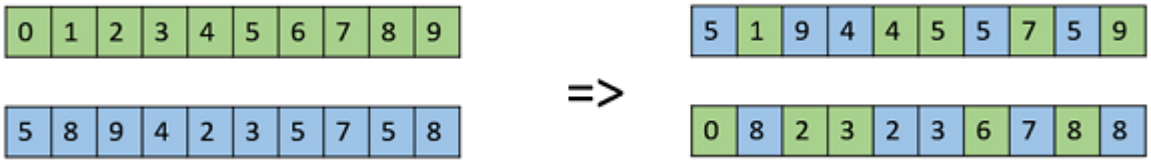


Figure 3.1 – Uniform Crossover [X]

crossoverSinglePoint

For single point crossover, a random point along the chromosomes of the parents is chosen using the program's random engine. The tail end of each chromosome is then swapped in the offspring. Both new organisms and the parents are then moved to a vector to be passed to the mutation function. This process is repeated for each mating pair.

Figure 3.2 – Single Point Crossover [X]

crossoverMultPoint

The multiple point crossover function is very similar to the one above, except this time there are two dividing points on a chromosome. To get these separate points, I created a vector containing all indexes in a chromosome. The vector is then shuffled, and all but the first two values are removed. This remaining vector would now contain the two dividing points.

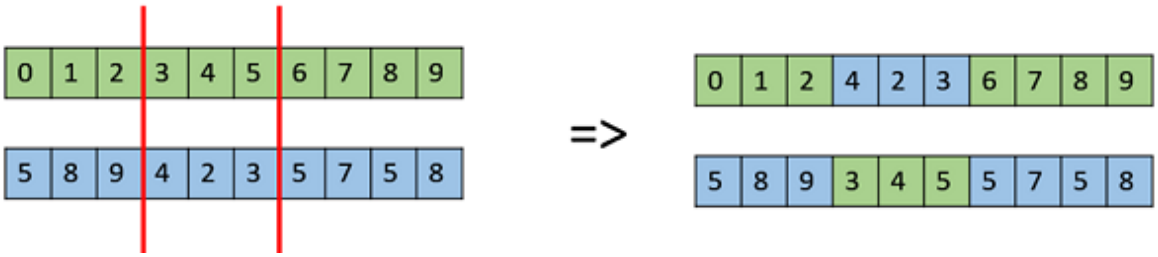


Figure 3.3 Multiple Point Crossover [X]

mutationScramble & mutationInversion

The mutation methods start off by doing a probability check to see if the mutation chance condition is met. For scramble mutation, a vector containing all indexes in a chromosome is created. This vector is shuffled, and the first two values are sorted and placed into a separate vector. These two values act as a subset of the chromosome. The genes within the subset are also randomly shuffled and replace the values previously in the subset of the chromosome. This is then repeated for each organism in the population.

Inversion mutation is the same as scramble in the sense that a random subset of each chromosome is altered. The difference is, instead of scrambling the subset, it is instead inverted.

mutationRandomValue

mutationSwap

Swap mutation is quite simple. If the probability check is successful, then a gene in the chromosome is swapped with another gene. This is then repeated as if it is only done once, the other mutation methods are more likely and more capable of having a larger impact on the population.

3.5.2 The Physics Engine

3.5.3 Collision Detection & Resolution

Each simulation object stores its own collider object. Colliders have a number of properties and methods which aid in detecting collisions between objects. The properties include:

- Collider area.
- Position of collider.
- Radius / Width / Height.
- Colour.
- Collider Type (Organism, food, wall, etc...).
- Collider Shape (Rectangle/Circle).
- minExtentX & maxExtentX (used for calculations)

3.5.4 The Renderer

The renderer class contains all the drawing functions needed for the program as well as holding the window object itself. The four drawing functions included are:

- DrawCircle: used for drawing food and poison elements.
- DrawBox: used for drawing the outer walls of the environment.
- DrawOrganism: used for drawing the population.
- DrawHealth: used for drawing the organism's current health above it.

Main.cpp is where the program begins, creating a new renderer object. Using the SFML library, a window is then created at a resolution of 1280 by 720 and a framerate limit is set. I chose 720p as 16:9 is the most common aspect ratio for modern computer monitors and laptop. Also, even cheaper laptops tend to have a resolution no smaller than 1366 by 768. In the worst case where the resolution is smaller than this, the program can be resized, only altering the simulation for a few frames.

At this point a new EvolutionSimulation object is created, with the window object passed into its constructor, allowing elements to be drawn outside of main.cpp. Next the main game loop begins.

3.5.5 Global Header

This file holds all global variables used throughout the program. Throughout the development of the project, it would store variables that I had not yet decided how to use or where to use them. Some of the variables it holds include:

- Population size.
- Number of food pieces.
- Number of poison pieces.
- The window resolution.
- The framerate limit.
- Default mutation chance.
- Default crossover chance.
- Default tournament size.
- Default survivor amount.
- Default generation time (in seconds).

When the user is prompted for the properties of the program, the answers are stored in the global header for use throughout runtime.

3.7 Summary

Section 4: Evaluation

4.1 Overview

A large portion of the program is non-deterministic and there is no exact expected outcome for a lot of functions. Couple with the stochasticity of the genetic algorithm mean that visual testing is going to an important factor in testing the solution as well as some manual white box testing. Collision detection and resolution is an aspect of the program that is deterministic and for any two shapes in any two positions, the program should know whether they have collided, how far they've collided and where they've collided.

4.2 Testing Solution

4.2.1 Element Creation

Each time the program is ran, the user chooses the number of organisms, food pieces and poison pieces present in the environment. The UI should show the correct number of these elements in different positions across the environment.

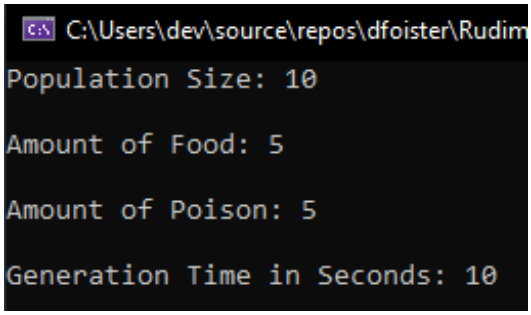


Figure 4.1 – Program Start

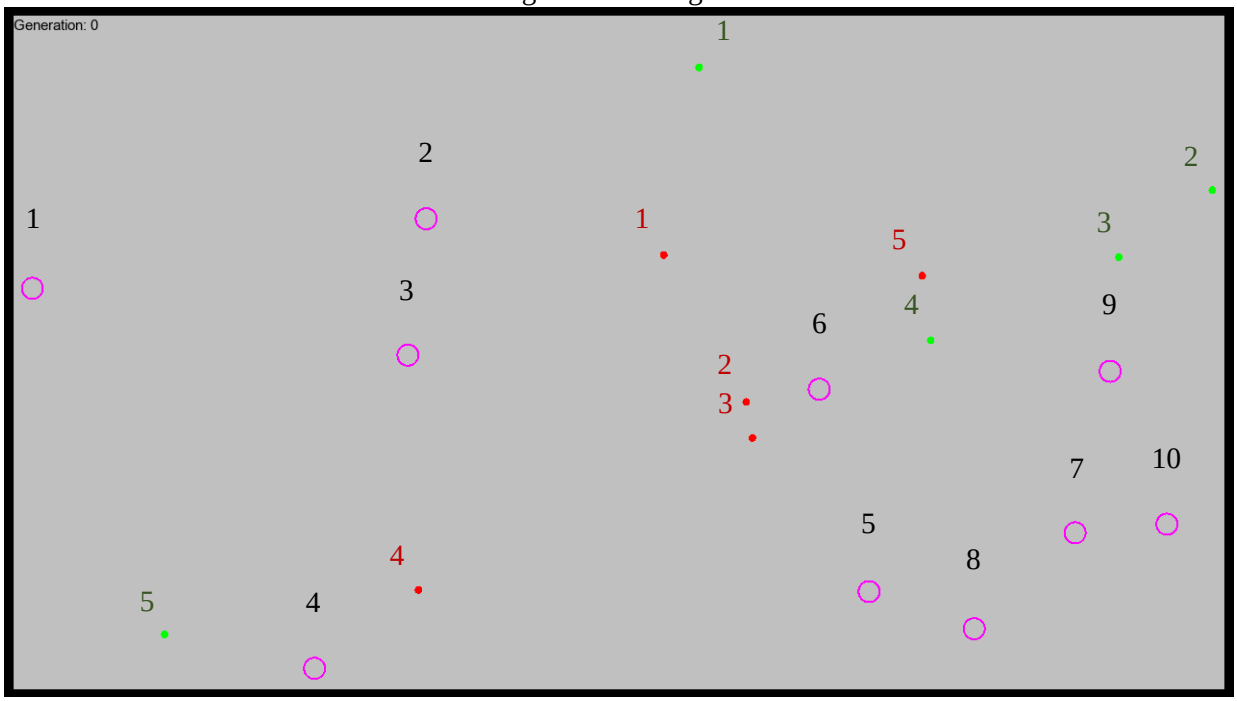


Figure 4.2 – Program Running with Numbered Elements

4.2.2 Selection Methods

Roulette Wheel Selection

Stochastic Universal Sampling

Random Selection

4.2.2 Crossover Methods

For all crossover testing the same parameters have been used:

- Population: 20.
- Food: 1.
- Poison: 1.
- Generation Time: 1 second.
- Selection Method: Roulette Wheel.
- Mutation: Disabled
- Survivors: 10

As crossover does not take fitness or mutation into account, I was able to disable mutation completely and reduce other factors down to nearly nothing. This mean all chromosomes were selected nearly randomly but will not affect the outcome of the crossover procedure.

Uniform Crossover

The figure below is output from the uniform crossover method. Chromosomes 1 and 2 represent the parent chromosomes, whereas three and four represent the children created via uniform crossover. Here you can see that each gene has remained in the same position as it did in the parent chromosome but each child alternates randomly which gene it receives.

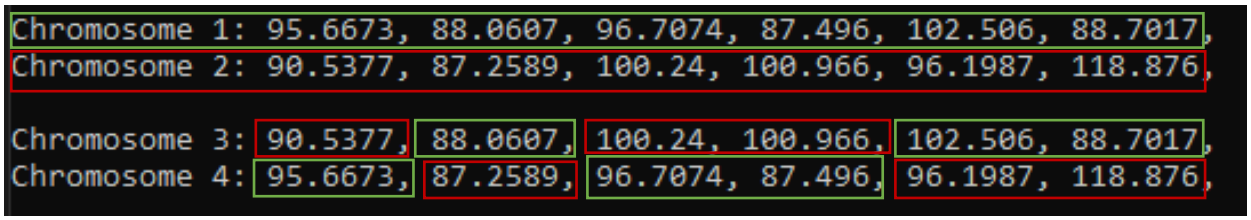


Figure 4.3 – Uniform Crossover Output

Single Point Crossover

Similar to the test above, chromosome 1 and 2 are the parent chromosomes and the bottom two are the children. The output of the function shows that the algorithm is successfully splitting the parent chromosomes at a single point and assigning each split to the correct child chromosomes.

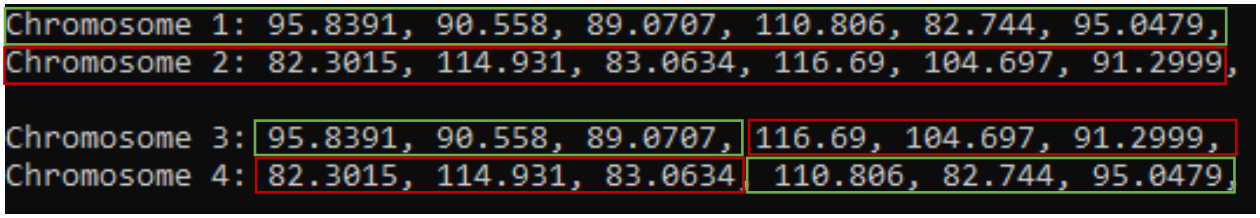


Figure 4.4 – Single Point Crossover Output

Multiple Point Crossover

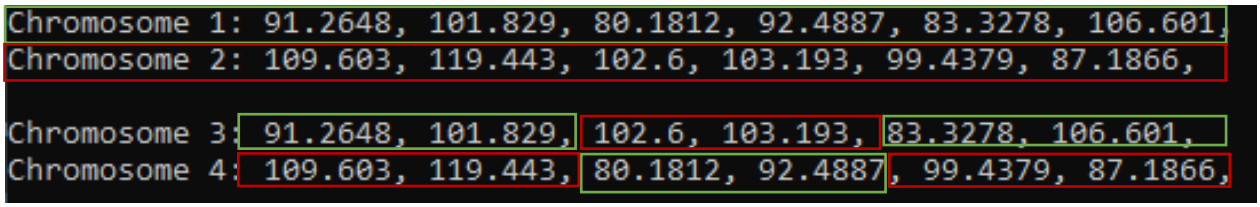


Figure 4.5 – Multiple Point Crossover Output

4.2.3 Mutation Methods

For the mutation tests, I printed 10 random chromosomes to the console and then passed them into the function, along with a mutation chance. I then outputted the same chromosomes after the function had completed to see whether the chromosomes had been mutated correctly.

Swap Mutation

As discussed in section 3, if the swap mutator is called, it will swap one or two random genes on a chromosome with two other random genes on that same chromosome. With a mutation chance of 50%, we should expect around half of the 10 chromosomes to be mutated, with one or two genes swapped. I will highlight the mutated chromosomes in green on the figure below. Once again, the top set of chromosomes are the originals, and the bottom set are the chromosomes after the function has ran.

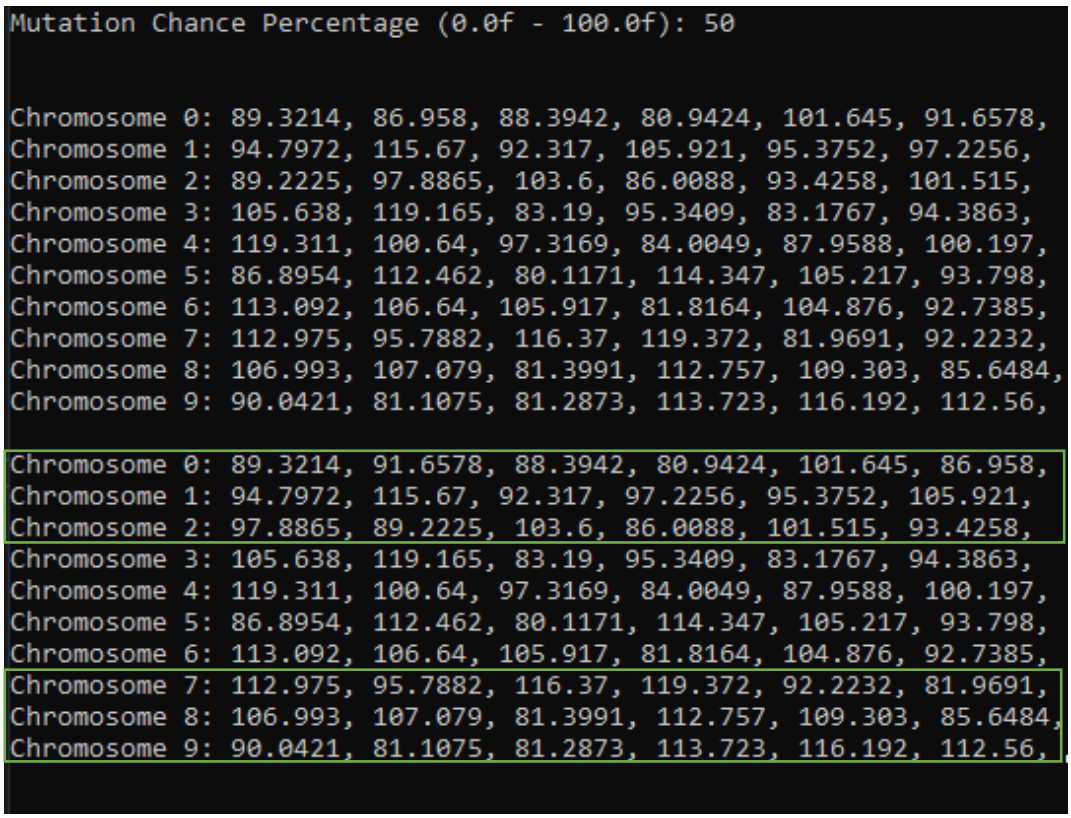


Figure 4.6 – Swap Mutation

As you can see, I have highlighted six out of the ten chromosomes. With a mutation chance of 50% and only 10 chromosomes, 60% is an acceptable result. To show that the random engine I am using is reliable I also tested whether the mutator would have been activated if called 1000 times with a mutation chance of 50%. The results of this are below.

Essentially, a random number is generated between 0 and 100. If this number is 50 or less, then the mutator would have been called and therefore a counter is incremented. Out of 1000 calls, the counter is incremented 513 times, or 51.3% of the time, which is more than acceptable for the simulation.

```
Mutation Chance Percentage (0.0f - 100.0f): 50

Chromosomes Mutated: 513
```

```
int counter = 0;
std::uniform_real_distribution<> distrRandomTest(0.0f, 100.0f);
for (int i = 0; i < 1000; i++) {
    if (distrRandomTest(engine) <= GLOBAL::MUTATION_CHANCE) {
        counter++;
    }
}
std::cout << "Chromosomes Mutated: " << counter;
```

Figure 4.6 – Swap Mutation & Random Engine

Scramble/Shuffle Mutation

If the scramble mutator is called, a random subset of that chromosome has its genes shuffled. Below is output from this function when 10 random organisms have been passed to it. A mutation chance of 50% has been used once again. On the left I will highlight which chromosomes have been mutated and then on the right I will highlight the subsets which has been scrambled, with colours corresponding to the original chromosome.

```
Mutation Chance Percentage (0.0f - 100.0f): 50


Chromosome 0: 119.386, 91.5861, 86.7582, 110.238, 100.325, 97.2376,
Chromosome 1: 109.394, 93.5428, 84.1047, 83.9149, 105.555, 111.62,
Chromosome 2: 90.2998, 119.651, 83.1781, 115.488, 92.2017, 80.7675,
Chromosome 3: 116.893, 94.8536, 102.992, 83.6427, 102.173, 82.4338,
Chromosome 4: 92.3708, 101.108, 102.72, 81.1199, 89.6484, 91.5117,
Chromosome 5: 105.095, 104.46, 109.1, 106.764, 105.331, 87.6438,
Chromosome 6: 119.528, 108.784, 118.531, 86.8721, 89.521, 96.1022,
Chromosome 7: 94.0451, 90.7883, 118.947, 88.3207, 100.739, 86.0061,
Chromosome 8: 95.8658, 86.0427, 109.943, 97.0091, 114.013, 114.685,
Chromosome 9: 103.215, 87.7721, 97.7705, 104.743, 108.248, 112.591,

Chromosome 0: 119.386, 91.5861, 86.7582, 110.238, 100.325, 97.2376,
Chromosome 1: 109.394, 84.1047, 93.5428, 83.9149, 105.555, 111.62,
Chromosome 2: 90.2998, 92.2017, 115.488, 83.1781, 119.651, 80.7675,
Chromosome 3: 116.893, 94.8536, 102.173, 83.6427, 102.992, 82.4338,
Chromosome 4: 92.3708, 102.72, 101.108, 81.1199, 89.6484, 91.5117,
Chromosome 5: 105.095, 104.46, 109.1, 105.331, 106.764, 87.6438,
Chromosome 6: 108.784, 119.528, 118.531, 86.8721, 89.521, 96.1022,
Chromosome 7: 94.0451, 90.7883, 118.947, 88.3207, 100.739, 86.0061,
Chromosome 8: 95.8658, 86.0427, 109.943, 97.0091, 114.013, 114.685,
Chromosome 9: 103.215, 87.7721, 97.7705, 104.743, 108.248, 112.591,
```

Figure 4.7 Scramble Mutation

Inversion Mutation

This test is almost identical to the last test except instead of scrambling the random subset, it is instead inverted. Here you can see exactly 50% of the chromosomes are mutated and each has a randomly inverted subset.

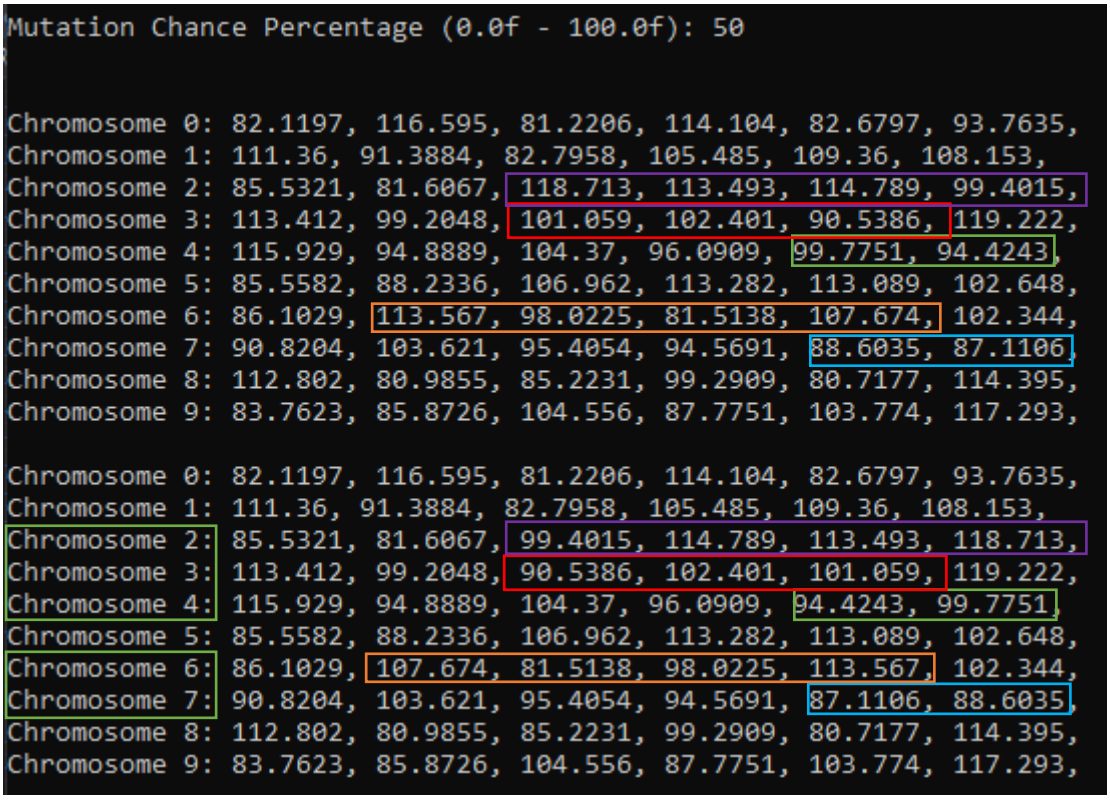
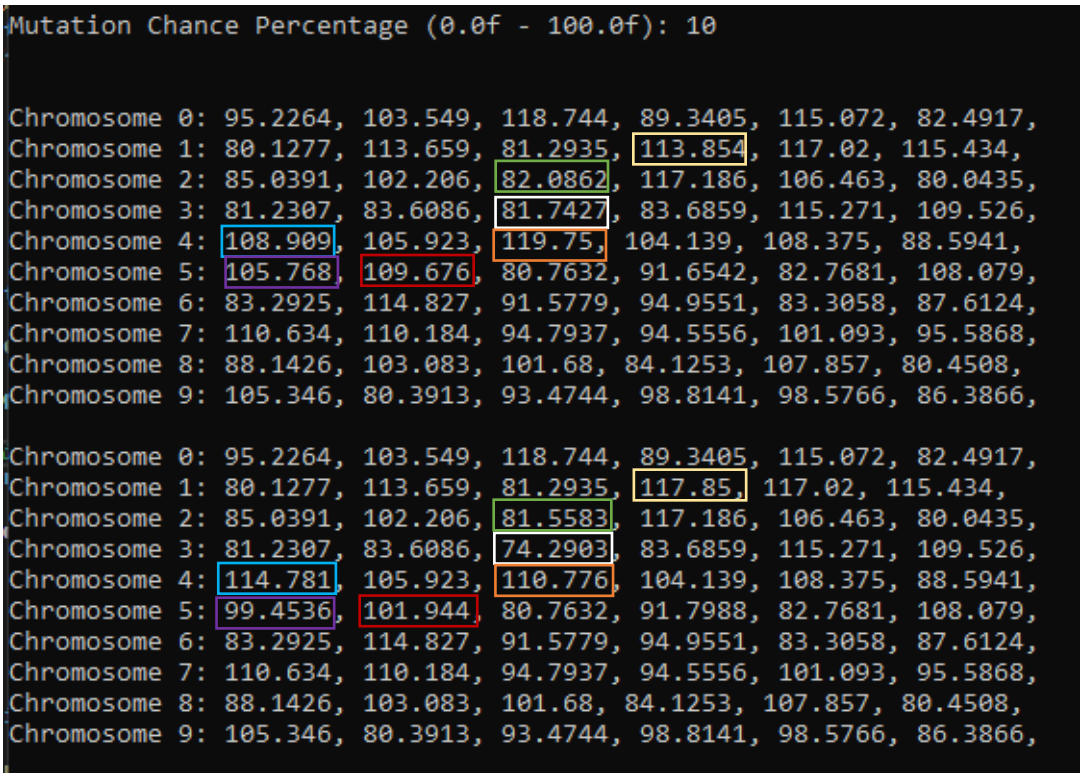


Figure 4.8 Inversion Mutation

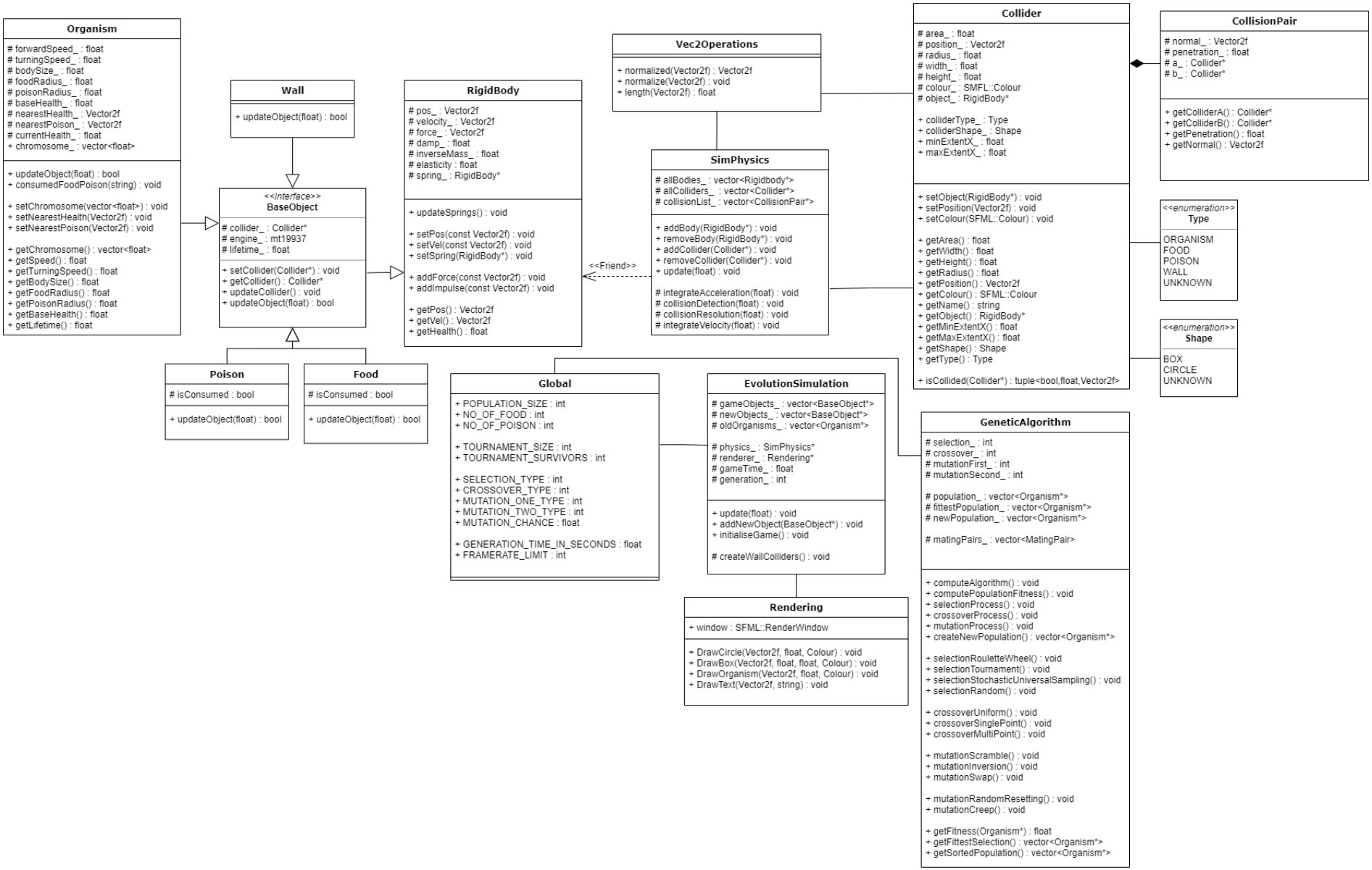
Creep Mutation

Creep mutation carries out mutation on single genes rather than the entire chromosome. For this reason, I lowered the mutation chance to 10% for the test to avoid cluttering the figure below. Once again, I have highlighted the mutated genes and colour coordinated them with the gene from the original chromosome.



For this test, the creep value was set between -15 and +15 and as you can see, all mutated genes fall into this range. The percentage of mutated genes in this example was 11.6%, landing very close to the chosen 10% mutation chance.

Random Resetting Mutation


Organism
forwardSpeed_ : float
turningSpeed_ : float
bodySize_ : float
foodRadius_ : float
poisonRadius_ : float
baseHealth_ : float
nearestHealth_ : Vector2f
nearestPoison_ : Vector2f
currentHealth_ : float
+ chromosome_ : vector<float>
+ updateObject(float) : bool
+ consumedFoodPoison(string) : void
+ setChromosome(vector<float>) : void
+ setNearestHealth(Vector2f) : void
+ setNearestPoison(Vector2f) : void
+ getChromosome() : vector<float>
+ getSpeed() : float
+ getTurningSpeed() : float
+ getBodySize() : float
+ getFoodRadius() : float
+ getPoisonRadius() : float
+ getBaseHealth() : float
+ getLifetime() : float
Wall
+ updateObject(float) : bool
<<Interface>>
BaseObject
collider_ : Collider*
engine_ : mt19937
lifetime_ : float
+ setCollider(Collider*) : void
+ getCollider() : Collider*
+ updateCollider() : void
+ updateObject(float) : bool
Poison
isConsumed : bool
+ updateObject(float) : bool
Food
isConsumed : bool
+ updateObject(float) : bool
RigidBody
pos_ : Vector2f
velocity_ : Vector2f
force_ : Vector2f
damp_ : float
inverseMass_ : float
elasticity : float
spring_ : RigidBody*
+ updateSprings() : void
+ setPos(const Vector2f) : void
+ setVel(const Vector2f) : void
+ setSpring(RigidBody*) : void
+ addForce(const Vector2f) : void
+ addImpulse(const Vector2f) : void
+ getPos() : Vector2f
+ getVel() : Vector2f
+ getHealth() : float
<<Friend>>
Vec2Operations
+ normalized(Vector2f) : Vector2f
+ normalize(Vector2f) : void
+ length(Vector2f) : float
SimPhysics
allBodies_ : vector<Rigidbody*>
allColliders_ : vector<Collider*>
collisionList_ : vector<CollisionPair*>
+ addBody(RigidBody*) : void
+ removeBody(RigidBody*) : void
+ addCollider(Collider*) : void
+ removeCollider(Collider*) : void
+ update(float) : void
# integrateAcceleration(float) : void
# collisionDetection(float) : void
# collisionResolution(float) : void
# integrateVelocity(float) : void
Collider
area_ : float
position_ : Vector2f
radius_ : float
width_ : float
height_ : float
colour_ : SMFL::Colour
object_ : RigidBody*
+ colliderType_ : Type
+ colliderShape_ : Shape
+ minExtentX_ : float
+ maxExtentX_ : float
+ setObject(RigidBody*) : void
+ setPosition(Vector2f) : void
+ setColour(SFML::Colour) : void
+ getArea() : float
+ getWidth() : float
+ getHeight() : float
+ getRadius() : float
+ getPosition() : Vector2f
+ getColour() : SFML::Colour
+ getName() : string
+ getObject() : RigidBody*
+ getMinExtentX() : float
+ getMaxExtentX() : float
+ getShape() : Shape
+ getType() : Type
+ isCollided(Collider*) : tuple<bool,float,Vector2f>
CollisionPair
normal_ : Vector2f
penetration_ : float
a_ : Collider*
b_ : Collider*
+ getColliderA() : Collider*
+ getColliderB() : Collider*
+ getPenetration() : float
+ getNormal() : Vector2f
<<enumeration>>
Type
ORGANISM
FOOD
POISON
WALL
UNKNOWN
<<enumeration>>
Shape
BOX
CIRCLE
UNKNOWN
Global
+ POPULATION_SIZE : int
+ NO_OF_FOOD : int
+ NO_OF_POISON : int
+ TOURNAMENT_SIZE : int
+ TOURNAMENT_SURVIVORS : int
+ SELECTION_TYPE : int
+ CROSSOVER_TYPE : int
+ MUTATION_ONE_TYPE : int
+ MUTATION_TWO_TYPE : int
+ MUTATION_CHANCE : float
+ GENERATION_TIME_IN_SECONDS : float
+ FRAMERATE_LIMIT : int
EvolutionSimulation
gameObjects_ : vector<BaseObject*>
newObjects_ : vector<BaseObject*>
oldOrganisms_ : vector<Organism*>
physics_ : SimPhysics*
renderer_ : Rendering*
gameTime_ : float
generation_ : int
+ update(float) : void
+ addNewObject(BaseObject*) : void
+ initialiseGame() : void
# createWallColliders() : void
Rendering
+ window : SFML::RenderWindow
+ DrawCircle(Vector2f, float, Colour) : void
+ DrawBox(Vector2f, float, float, Colour) : void
+ DrawOrganism(Vector2f, float, Colour) : void
+ DrawText(Vector2f, string) : void
GeneticAlgorithm
selection_ : int
crossover_ : int
mutationFirst_ : int
mutationSecond_ : int
population_ : vector<Organism*>
fittestPopulation_ : vector<Organism*>
newPopulation_ : vector<Organism*>
matingPairs_ : vector<MatingPair>
+ computeAlgorithm() : void
+ computePopulationFitness() : void
+ selectionProcess() : void
+ crossoverProcess() : void
+ mutationProcess() : void
+ createNewPopulation() : vector<Organism*>
+ selectionRouletteWheel() : void
+ selectionTournament() : void
+ selectionStochasticUniversalSampling() : void
+ selectionRandom() : void
+ crossoverUniform() : void
+ crossoverSinglePoint() : void
+ crossoverMultiPoint() : void
+ mutationScramble() : void
+ mutationInversion() : void
+ mutationSwap() : void
+ mutationRandomResetting() : void
+ mutationCreep() : void
+ getFitness(Organism*) : float
+ getFittestSelection() : vector<Organism*>
+ getSortedPopulation() : vector<Organism*>

4.3 Results

4.4 Summary

Section 5: Conclusion

5.1 Satisfaction of Aims & Objectives

5.2 What is established and what can be done further?

5.3 What Went Well?

5.4 Even Better If?

References

[1] – Bacardit, J 2020, *L02 – Genetic Algorithms*, lecture notes, Biologically-Inspired Computing CSC3423, Newcastle University, delivered 18th October 2020.

[2] – Rooy, N 2017, *Evolving Simple Organisms using a Genetic Algorithm and Deep Learning from Scratch with Python*, Nathan Rooy, viewed 20th January 2021, <https://nathanrooy.github.io/posts/2017-11-30/evolving-simple-organisms-using-a-genetic-algorithm-and-deep-learning/>

[3] – Stanford University 2020, *graphics.h*, Stanford University, viewed 12th January 2021, <https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1126/materials/cppdoc/graphics.html>

[4] – Thompson, J 2020, *Collision Detection*, Jeffrey Thompson, viewed 20th December 2020, <http://www.jeffreythompson.org/collision-detection>

[5] – Speirs, N 2020, *C++ Lecture Material*, lecture notes, Programming for Games CSC3221, Newcastle University, delivered 29th September 2020.

[6] – Mukhopadhyay, D, Balitanas, M, Farkhod, A, Jeon, S & Bhattacharyya, D 2009, 'Genetic Algorithm: A Tutorial Review', *International Journal of Grid and Distributed Computing*, vol. 2, no. 3, pp. 25-32.

[X] - TomsPlanner. 2009. *Tom's Planner - Gantt charts for the rest of us.* [online] Available at: https://www.tomsplanner.com/.

[X] - Eiben, A. and Smith, J., 2015. *Introduction to Evolutionary Computing (Natural Computing Series)*. 2nd ed. Heidelberg: Springer, pp.13-14.

[X] - Kumar, U., 2020. Genetic Algorithm. [Blog] *Analytics Vidhya*, Available at: <https://medium.com/analytics-vidhya/genetic-algorithm-5aba4aac48f7>.

[X] - Tutorials Point. n.d. *Genetic Algorithms - Crossover*. [online] Available at: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm.

[X] - Eiben, A. and Smith, J., 2015. *Introduction to Evolutionary Computing (Natural Computing Series)*. 2nd ed. Heidelberg: Springer, pp.53-54.

[X] - Nave, R., 2017. *Newton's Laws*. [online] Hyperphysics.phy-astr.gsu.edu. Available at: http://hyperphysics.phy-astr.gsu.edu/hbase/Newt.html. [Accessed 13 April 2020]

[X] - Davison, R 2020, *Game Physics Part 2*, lecture notes, Computer Science CSC3222, Newcastle University, delivered January 2021.

[X] - Ponamgi, M., Cohen, J. and Lin, M., 1995. *Collision Detection for Virtual Environments and Simulations Using Incremental Computation*. [online] Gamma.cs.unc.edu. Available at: http://gamma.cs.unc.edu/COLLISION_PON/collision.html#:~:text=Collision%20detection%20is%20a%20fundamental,interactions%20in%20the%20simulated%20environments.. [Accessed 21 April 2021].

[X] - Workman, K., n.d. *Processing Collision Detection*. [online] Happy Coding. Available at: https://happycoding.io/tutorials/processing/collision-detection.

[X] - Cplusplus.com. n.d. *A Brief Description - C++ Information*. [online] Available at: https://www.cplusplus.com/info/description/.

[X] -