



**NANYANG
TECHNOLOGICAL
UNIVERSITY**

Genetic Algorithms for Swarm Parameter Tuning

Submitted by: "#CHEE JUN YUAN GLENN#" <CHEE0124@e.ntu.edu.sg>

Matriculation Number: U1621371J

Supervisor: Ast/P Zinovi Rabinovich

School of Computer Science & Engineering

A final year project report presented to the Nanyang Technological University
in partial fulfilment of the requirements of the degree of

Bachelor of Computer Science

2019

Abstract

Swarming behaviour is based on the aggregation of simple drones exhibiting basic instinctive reactions to stimuli. However, to achieve overall balanced/interesting behaviours, the relative importance of these instincts as well their internal parameters must be tuned. This project attempts to achieve a series of non-trivial swarm-level behaviours by applying Genetic Programming as means of such tuning.

To measure the performance of a swarm, a swarm simulator was designed in Python. The simulator receives parameters for a swarm and a scenario, and outputs a score based on the swarm's performance for that scenario. The score is used by the Multi-Objective Genetic Algorithm (MOGA) as fitness to determine the swarm's chance of breeding; higher the fitness, higher the chance of breeding. The Genetic Algorithm (GA) terminates after 1000 generations and evaluations are done thereafter on the resulting population for analysis.

It was observed that the GA did indeed improve individual specialisations substantially, but improved multi-specialisations only marginally better. Uniform crossover was evident to be the better crossover function, possibly owing to the genes being independent. Interesting behaviours, namely swarm cohesion and bait ball, were also observed on the resultant.

In conclusion, MOGA shows promise of optimising multiple objectives together but is negligible thus far. However, MOGA succeeded in manufacturing certain behaviours by tuning swarm parameters. Further study needs to be done to determine the full potential of MOGA.

Acknowledgements

First and foremost, I would like to express my sincerest thanks and greatest gratitude to my parents for their unwavering support, and also to the many other parties that helped see this project through.

I am truly grateful and thankful to my supervisor, Asst. Prof. Zinovi Rabinovich, who supported me throughout the entire project. He allowed many opportunities for creative freedom and learning while still providing good and constructive feedback, and suggestions on improvements.

I would like to thank the Artificial Intelligence Community on web forums for providing suggestions and help regarding Evolutionary & Genetic Algorithms.

I would like to take this opportunity to also thank Irene Goh for helping to lease and provide support for a Supercomputer from the *School of Computer Sciences & Engineering* (SCSE), *Nanyang Technological University* (NTU), and also to Kesavan Asaithambi for helping to lease a workstation from the *Computational Intelligence Lab* (CIL), *NTU*.

I would also like to thank other members of the lab for their help, especially Bector Ridhima for assisting with optimising the swarm simulation and providing good ideas and suggestions while brainstorming.

Chee Jun Yuan Glenn

April 2019

Table of Contents

Abstract	i
Acknowledgements	ii
Table of Contents	iii
List of Tables	v
List of Figures	vii
List of Graphs	ix
List of Symbols	x
Nomenclature.....	xi
Chapter 1 Introduction	1
1.1 Background.....	1
1.2 Objectives and Scope	2
Chapter 2 Literature Review	3
2.1 Emergence	3
2.2 Boid Behavioural Model	5
2.2.1 Limitations of the model	6
2.3 Genetic Algorithm	7
2.4 The Theory	8
Chapter 3 Designing the swarm simulation	9
3.1 Prototype swarm simulation design in Unity	11
3.1.1 3D Boid model.....	11
3.1.2 Boid Algorithm.....	12
3.1.3 Visual inspection	14
3.1.4 Optimisations.....	14
3.1.5 Usability of Unity / Issues	15
3.2 Swarm simulation design in Python	16
3.2.1 Visual inspection of the swarm	16
3.2.2 Data Structure	16
3.2.3 Optimisations.....	17
3.2.4 Parameters of the swarm simulation	18
3.2.5 JSON	19
3.2.6 Predators of the Boids	20
3.2.7 Preys of the Boids	21

3.2.8	Predator and prey relationship	22
3.2.9	Types of predators and preys.....	22
3.2.10	Swarm specialisations	25
3.2.11	Scene and Scenarios.....	26
3.2.12	Measurements	29
3.2.13	Fitness	31
Chapter 4	Designing the Genetic Algorithm (GA)	31
4.1	Initialisation	32
4.1.1	Seeding	33
4.2	Selection	34
4.3	Genetic operations	35
4.3.1	Crossover.....	36
4.3.2	Mutation	39
4.4	Heuristics	42
4.4.1	Alpha (α) normalisation	42
4.4.2	Beta (β) normalisation	43
4.5	Termination	44
Chapter 5	Experiment & Results	45
5.1	Tuning results of Genetic Algorithm (GA)	45
5.2	Cross-specialisation results	57
5.2.1	Comparison against population	61
5.3	Swarm observations	64
5.3.1	Cohesion.....	64
5.3.2	Bait Ball.....	64
Chapter 6	Conclusions and future improvements.....	67
6.1	Recommendations for future work	68
References	69
Appendix	71

List of Tables

Table 2.4-1: Comparison of physics engines.....	10
Table 3.1-1: Code snippet of the Boid Algorithm pseudocode	13
Table 3.2-1: Parameters of a swarm.....	19
Table 3.2-2: Sample of a JSON parameter file.....	20
Table 3.2-3: Code snippet for reading JSON parameter files.....	20
Table 3.2-4: Description of predator types.....	24
Table 3.2-5: Description of prey types.....	25
Table 3.2-6: Description of specialisations	26
Table 3.2-7: Scenario descriptions.....	27
Table 3.2-8: Visualisation of different scenes/scenarios.....	28
Table 3.2-9: Performance measurement descriptions	30
Table 3.2-10: Normalised measurement for performance.....	30
Table 3.2-11: Code snippet to return the evaluation of a swarm simulation.....	31
Table 4.1-1: Code snippet for Genetic Algorithm's initialisation.....	32
Table 4.1-2: Example of seeding the population	33
Table 4.2-1 : Code snippet for population fitness normalisaion.....	35
Table 4.2-2: Code snippet for choosing 2 random specialisations	35
Table 4.2-3: Code snippet for choosing parents from 2 random specialisations.....	35
Table 4.3-1: Code snippet of the combination and mutation of parents' genomes	36
Table 4.3-2: Description of crossover types.....	38
Table 4.3-3: Code snippet of crossover function.....	39
Table 4.3-4: Example of Bit string mutation with 11 genes.....	40
Table 4.3-5: Example of Flip Bit mutation with 8 genes	40
Table 4.3-6: Example of Boundary mutation with 8 genes.....	41
Table 4.3-7: Description of mutation types.....	41
Table 4.3-8: Code snippet of mutation function	41
Table 4.4-1: Probabilities of specialisation inheritance	43
Table 4.4-2: Probabilities of heuristic beta normalisation	44
Table 4.4-3: Code snippet of β normalisation.....	44
Table 5.2-1: Matrix of singular-specialisation fitness.....	58
Table 5.2-2: Matrix of cross-specialisation fitness	58
Table 5.2-3: Average fitness of cross-specialisations	60

Table 5.2-4: Comparison of cross-specialisation against population for specialisation 0 – Swarm’s velocities	61
Table 5.2-5: Comparison of cross-specialisation against population for specialisation 1 – Predators not in Boids’ radius.....	61
Table 5.2-6: Comparison of cross-specialisation against population for specialisation 2 – Preys in Boids’ radius	61
Table 5.2-7: Comparison of cross-specialisation against population for specialisation 3 – Swarm’s cohesion	62
Table 5.2-8: Comparison of cross-specialisation against population for specialisation 4 – Distance from predators in the presence of preys.....	62
Table 5.2-9: Comparison of cross-specialisation against population for specialisation 5 – Distance from preys in the presence of predators.....	62
Table 5.2-10: Comparison of cross-specialisations with population	63

List of Figures

Figure 2.1-1: Flock of birds in a V formation	3
Figure 2.1-2: Swarm behaviour of bird flock	3
Figure 2.1-3: School of fishes shoaling.....	4
Figure 2.1-4: School of fishes swimming in the same coordinated direction	4
Figure 2.1-5: School of fishes swarming about a common center; a bait ball	4
Figure 2.1-6: An edge of a large bait ball.....	4
Figure 2.2-1: Boids' steering behaviours [4].....	5
Figure 2.2-2: A Boid's neighbourhood.....	6
Figure 2.3-1: Depiction of a typical Genetic Algorithm (GA)	8
Figure 3.1-1: Initial setting of a scene for the swarm simulation in Unity.....	11
Figure 3.1-2: 3D model of Boid prototype	12
Figure 3.1-3: Early prototype swarm simulation in Unity	14
Figure 3.2-1: Visualisation of Python's swarm simulation in different dimensions	16
Figure 3.2-2 Boid's predator steering behaviour	21
Figure 3.2-3: A swarm of 256 predators in the swarm simulation	21
Figure 3.2-4 Boid's prey steering behaviour	22
Figure 3.2-5: Randomly scattered 256 preys in the swarm simulation.....	22
Figure 3.2-6: Predator and prey relation to Boids	22
Figure 3.2-7: Predator's hunting behaviour	23
Figure 3.2-8: Predator's patrolling behaviour	23
Figure 3.2-9: Prey's random behaviour.....	24
Figure 3.2-10: Preys' along predator's route behaviour	25
Figure 3.2-11: scene_velocities.json.....	28
Figure 3.2-12: scene_predators.json	28
Figure 3.2-13: scene_preys.json	28
Figure 3.2-14: scene_distances.json.....	28
Figure 3.2-15: scene_distancesPredator.json.....	28
Figure 3.2-16: scene_distancesPrey.json.....	28
Figure 4.1-1: Visualisation of genomes for a population.....	33
Figure 4.1-2: Visualisation of genome for the seeded individual.....	34
Figure 4.2-1: Population's fitness before normalisation.....	34
Figure 4.2-2: Population's fitness after normalisation	34
Figure 4.3-1: Single-point crossover.....	36

Figure 4.3-2: Two-point crossover	37
Figure 4.3-3: Uniform crossover	38
Figure 4.3-4: Bit string mutation	40
Figure 4.4-1: Population's fitness before α normalisation	43
Figure 4.4-2: Population's fitness after α normalisation	43
Figure 5.2-1: Slices of generations for cross-specialisation.....	57
Figure 5.3-1: Swarm cohesion.....	64
Figure 5.3-2: Swarm predator avoidance	65
Figure 5.3-3: Swarm bait ball	66

List of Graphs

Graph 5.1-1: Average fitness over generation for specialisation 0 – Swarm’s velocities	46
Graph 5.1-2: Average fitness over 5000 generations for specialisation 0 – Swarm’s velocities	47
Graph 5.1-3: Average fitness over generation for specialisation 1 – Predators not in Boids’ radius	48
Graph 5.1-4: Average fitness over 5000 generations for specialisation 1 – Predators not in Boids’ radius	49
Graph 5.1-5: Average fitness over generation for specialisation 2 – Preys in Boids’ radius..	50
Graph 5.1-6: Average fitness over generation for specialisation 3 – Swarm’s cohesion	52
Graph 5.1-7: Average fitness over generation for specialisation 4 – Distance from predators in the presence of preys	54
Graph 5.1-8: Average fitness over generation for specialisation 5 – Inversed distance from preys in the presence of predators.....	56
Graph 5.2-1: Cross-specialisation for specialisation 0 – Swarm’s velocities	59
Graph 5.2-2: Cross-specialisation for specialisation 1 – Predators not in Boids’ radius.....	59
Graph 5.2-3: Cross-specialisation for specialisation 2 – Preys in Boids’ radius.....	59
Graph 5.2-4: Cross-specialisation for specialisation 3 – Swarm’s cohesion.....	59
Graph 5.2-5: Cross-specialisation for specialisation 4 – Distance from predators in the presence of preys.....	59
Graph 5.2-6: Cross-specialisation for specialisation 5 – Inversed distance from preys in the presence of predators.....	59

List of Symbols

α	Percentage of fitness influenced during selection; Alpha (α) normalisation.
β	Percentage of child choosing a random specialisation; Beta (β) normalisation.
f	Fitness.
f_j	Fitness for <i>specialisation_j</i> .
$f_{(i,j)}$	Fitness of <i>population_i</i> for <i>specialisation_j</i> .

Nomenclature

Boid	Bird-Like Object
FYP	Final Year Project
GA	Genetic Algorithm
GP	Genetic Programming
MOGA	Multi-Objective Genetic Algorithm
NTU	Nanyang Technological University
SCSE	School of Computer Science & Engineering

Chapter 1 Introduction

In this day and age, autonomous agents are becoming less of an abnormality and more of a prevalent sight for your average consumer. The ability to simulate and/or replicate true to life interactions between entities has been, and still is, one of the holy grails of Artificial Intelligence [1], being able to simulate lifelike behaviours accurately also brings us a step closer to bridging the gap between the fields of Biological, and Computer Sciences [2].

1.1 Background

In 1986, Craig Reynolds made a computer model of coordinated animal motion [3], such as birds flocking and fishes schooling. It consists of steering behaviours which describe how an individual Bird-Like Object (Boid) would manoeuvre based on its surroundings [4]. This gave rise to seemingly realistic-looking flocking behaviours observed during simulations [5] through the implementation of the Boid behavioural model; the Boid Algorithm.

The Boid algorithm was revolutionary and constituted a great leap forward compared to the usual standard classical techniques used [6]. It was later implemented to simulate colonies of bats and penguins in the feature film “*Batman Returns*” (1992), swarms of flying bird-like creatures in the video game “*Half-Life*” (1998), and further extended to be used in autonomous agents such as unmanned aerial vehicles and the likes [7] [8].

Swarming behaviour is based on aggregation of simple drones exhibiting basic instinctive reactions to stimuli, in this case drones being the Boids, instinctive reactions being the aforementioned steering behaviours, and stimuli being other Boids.

Although the Boid algorithm is a reasonable staple for swarming algorithms, and is commonly employed in autonomous agents [7] [8] [9], it itself is not definitive as the

parameters themselves are not perfect [6] and are insufficient to demonstrate certain complex behaviours we can observe in nature.

Existing studies already attempt to manipulate and extend from the original Boid algorithm to achieve varied swarming behaviours, through a wide array of methods ranging from, manipulation through parasitic infection algorithms [10], stochastic and heuristic principles [11], introduction of additional parameters to the Boid algorithm [5], etc. However, to achieve overall balanced/interesting behaviours, the relative importance of steering behaviours as well as their internal parameters must be tuned.

There is still a great deal to be improved upon the original Boid algorithm to provide even greater realistic behaviours and possibly answer the question to the “whys” and “how” flocks of birds truly behave the way they do [6] [12].

1.2 Objectives and Scope

This project attempts to achieve a series of non-trivial swarm-level behaviours by applying Genetic Programming (GP) as means of such tuning. Notwithstanding studies already being done before on integrating GP with the Boid algorithm [13] [14] [15], they focus on a different high-level behaviour instead.

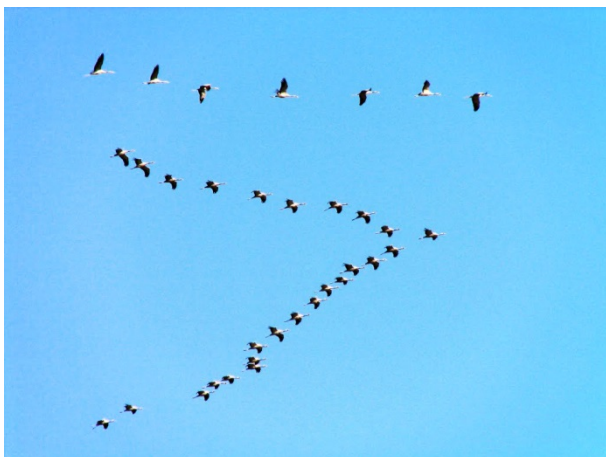
A Genetic Algorithm (GA), Multi-Objective Genetic Algorithm (MOGA), different from other GA's used in previous works will be implemented. The cause and effect of swarm behaviours resulting from the GA will be evaluated. Potential further optimisations will be deduced by “training” the swarm to replicate certain behaviours.

Chapter 2 Literature Review

This chapter provides the literature review on key and important concepts pertaining to the project.

2.1 Emergence

Birds are individually unsophisticated, they do not have much of a brain (hence the term “birdbrain”), nor stratagem and yet collectively, a flock of birds exhibits a higher level of intelligence than itself. A bird flock can manifest certain unique behaviours, such as the “V formation” which improves flight efficiency, and swarming behaviours, etc.



By Hamid Hajihusseini -
<https://www.panoramio.com/photo/43585282>, CC BY 3.0,
<https://commons.wikimedia.org/w/index.php?curid=31237576>

Figure 2.1-1: Flock of birds in a V formation



By Christoffer A Rasmussen - Own work, Public Domain,
<https://commons.wikimedia.org/w/index.php?curid=6736876>

Figure 2.1-2: Swarm behaviour of bird flock

This phenomenon of manifestation is known as emergence, it occurs when smaller entities form a larger entity that exhibits properties that are not present before. For the example of a bird swarm, the smaller entities are the birds, the larger entity is the swarm, and the properties absent in individual birds are the “V formation” and swarming behaviours. The emergence of behaviours in a bird swarm comes as a result of a simple set of rules followed by individual birds. In simpler terms, emergence is complexity arising from simplicity [16].

Unique behaviours emerge from school of fishes as well, namely shoaling and schooling. Schooling represents any group of fishes swimming in the same direction in a coordinated manner and shoaling represents any group of fishes that stays together.



By Uxbona - Own work, CC BY 3.0,
<https://commons.wikimedia.org/w/index.php?curid=5684850>

Figure 2.1-3: School of fishes shoaling



By Jim and Becca Wicks - Snap of Snapper!, CC BY 2.0,
<https://commons.wikimedia.org/w/index.php?curid=2760939>

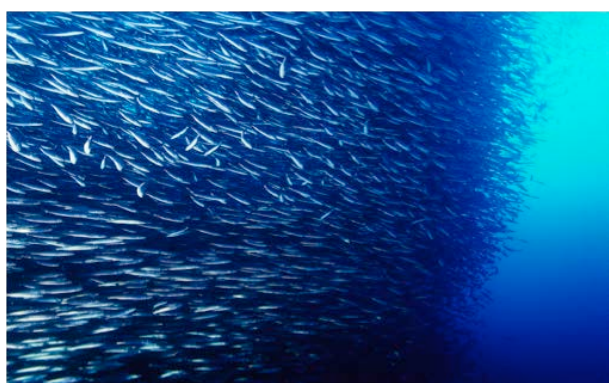
Figure 2.1-4: School of fishes swimming in the same coordinated direction

Another special behaviour is exhibited by school of small fishes in predator avoidance. When threatened by predator(s), certain fishes naturally ball up together defensively and swarm around the predator(s) in a spherical formation, manifesting into what is known as a bait ball.



By Bare Dreamer - Flickr, CC BY 2.0,
<https://commons.wikimedia.org/w/index.php?curid=20234085>

Figure 2.1-5: School of fishes swarming about a common center; a bait ball



By Jim and Becca Wicks - Snap of Snapper!, CC BY 2.0,
<https://commons.wikimedia.org/w/index.php?curid=2760939>

Figure 2.1-6: An edge of a large bait ball

2.2 Boid Behavioural Model

“Birds of a feather flock together”, the proverb connotes that people who share similar characters or interests often spend time with each other. This is mostly true not only for humans but for many animals as well. Animals such as birds, bees, fishes, etc. exhibit coordinated animal motion in the wild, hence the terms: flock of birds, swarm of bees, school of fish, colony of ants, etc. It is precisely the behaviour of such coordinated animal motion that the Boid behavioural model attempts to replicate, i.e. bird flocks and fish schools [4].

The model consists of 3 simple steering behaviours which describe how an individual Boid would manoeuvre based on other Boids in its vicinity [4], the three key steering behaviours of the Boid algorithm are: separation, alignment, and cohesion. These parameters respectively steer an individual Boid to, avoid crowding with other Boids, move towards the average heading of other Boids, and move towards the average position of other Boids.

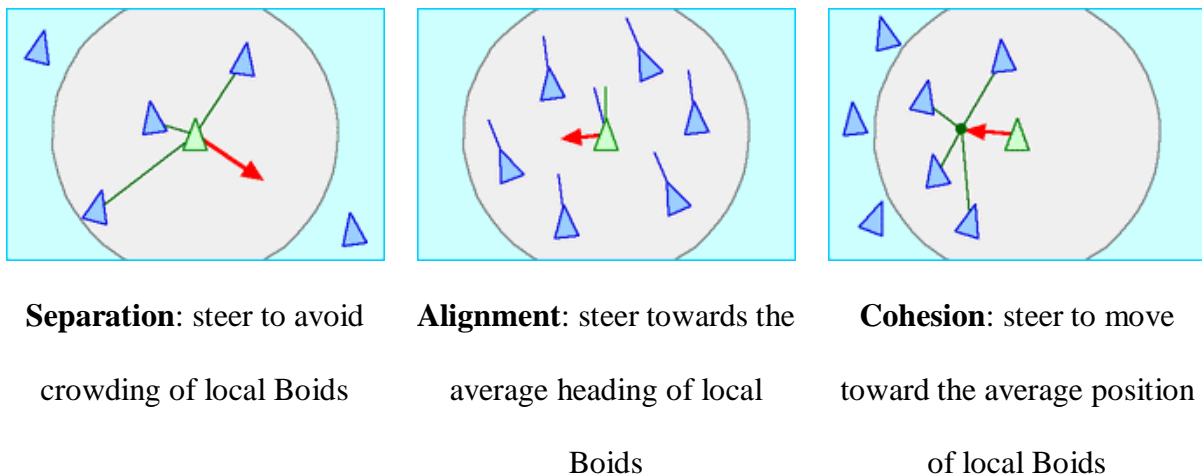


Figure 2.2-1: Boids' steering behaviours [4]

Even though the environment or scene is fully observable to every Boid, flocking necessitates the fact that each individual Boid reacts only to other Boids in its immediate vicinity, i.e. within a small radius around itself; a small neighbourhood. Hence, the neighbourhood of a

Boid is characterised by a certain small radius measured from the center of the Boid, other Boids not present in the neighbourhood are disregarded. The original Boid model comprised of an additional angle (similar to a field of view), but for the purposes of this study, only the radius will be considered.

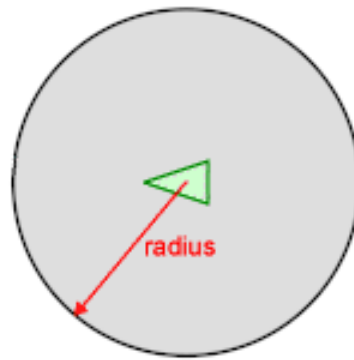


Figure 2.2-2: A Boid's neighbourhood

The neighbourhood is akin to a model of limited perception, e.g. birds in foggy skies or fishes in murky waters, but it is also correct to perceive it as defining the region in which other Boids can influence a Boid's steering [4].

2.2.1 Limitations of the model

There are certain limiting factors that come with implementing the Boid model however. It itself is not definitive as the parameters themselves are not perfect [6] and are insufficient to demonstrate certain complex behaviours observed in nature. For example: it is not explicitly stated how one should “balance” the behaviours, e.g. cohesion directly contradicts separation and the weights of the behaviours are not specified. The radius for a Boid's neighbourhood is not well defined as well, similarly for the angle in the model. These uncertainties make it challenging to properly and accurately simulate lifelike swarm behaviours.

2.3 Genetic Algorithm

A Genetic Algorithm (GA) is a subset of Evolutionary Algorithms (EA) and is a metaheuristic optimisation algorithm. It draws its inspiration from the processes of natural selection in Mother Nature and involves similar biological processes such as selection, crossover, and mutation to solve optimisation problems.

A typical GA encapsulates these processes:

1. Initialisation

- A population of possible solutions to the problem is randomly generated or seeded, or both, varying through the search space.

2. Selection

- Similar to the process of natural selection, a proportion of the population is selected to procreate and generate newer solutions based on their fitness, i.e. fitter individuals have a higher chance of breeding.

3. Genetic operators

- Genetic operators are similar to breeding processes and inheritance. Genes of parents undergo recombination (crossover) to produce offspring with similar traits and then genomes of the offspring are mutated to imitate natural mutation.

4. Heuristics

- Additionally, heuristics may be used to minimise crossover (breeding) from parents that are too similar, this supports diversity and helps prevent premature convergence [17] [18].

5. Termination

- The generational process, aforementioned processes from 2 – 4: selection, genetic operators, and heuristics are repeated through many generations of iterations until certain condition(s) for termination is fulfilled. Some conditions for termination are:
 - Solution(s) satisfying the minimal requirements is found.
 - Fixed number of generations of iterations.
 - Set amount of allocated computational time allowed.

- Newer generations of populations do not generate better solutions.
- Any combination of the above or others...

A visualisation of the GA process is depicted below:

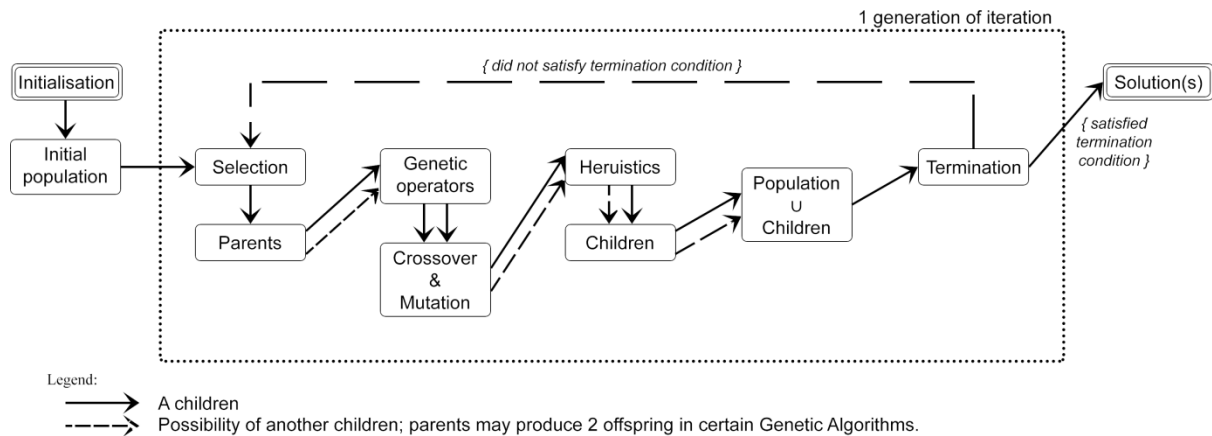


Figure 2.3-1: Depiction of a typical Genetic Algorithm (GA)

A fitness function is also implemented alongside the GA, it determines how an individual is performing in regards to solving the problem. It is used for the selection process where fitter individuals have a greater chance at mating and producing offspring, rendering weaker populations to gradually dwindle through generations.

2.4 The Theory

The idea behind this project is to first design a sound swarm simulation, capable of replicating swarms in real life; true-to-life and lifelike swarms. Measurements will then be defined to determine the performance of a swarm. The Genetic Algorithm (GA) will use the measurements as a form of fitness to select fitter individuals to breed; the higher the fitness of an individual, the more chance of it producing offspring.

Through multiple generations of Genetic Programming, certain behaviours should be emulated and emerge from the swarms as their parameters are being tuned by the GA. The swarms will have different specialisations tailored to the scenarios and there will be multiple

scenarios to tune for specific behaviours. The aim is to produce a swarm that performs relatively well in all specialisations, by combining swarms who perform well in different specialisations together.

Chapter 3 Designing the swarm simulation

A swarm simulation simulator with the ability to judge the performance of the swarm is required for the Genetic Algorithm (GA) to work, visual inspection and visualisation of the swarm in 2D/3D is also required. Geometric analysis and extension to 4D or higher is preferred but not required. The ability to integrate with the GA needs to be taken into consideration also. Built-in physics engine or simple physics would be beneficial.

Comparisons of some available physics engine are as follows:

Engine	Language	Cross-platform	2D / 3D	Closed/ Open Source	Price	Additional notes
Advanced Simulation Library	C++	✓	3D	Open	Free	Barely any detailed documentation, hardware accelerated Multiphysics simulation software.
AGX Dynamics	C++	✓	3D	Closed	?	Pricing unknown.
Bullet	C, C++	✓	2D, 3D	Open	Free	Free and open source subjected to zlib license.
Chipmunk	C	✓	2D	Open	Free	2D real-time rigid body physics engine. Official support for Ruby and 3 rd party support for Python.
Digital Molecular Matter	?	✓	3D	Closed	?	Pricing unknown, middleware physics engine, used in movies such as Fast & Furious 6 (2013), The Amazing Spider-Man 2 (2014), Guardians of the Galaxy (2014), etc.
Havok	?	✓	3D	Closed	?	Pricing unknown.

Microsoft XNA	C#, Visual Basic	✗	2D, 3D	Closed	Free	Freeware, video game development.
MonoGame	C#	✓	2D, 3D	Closed	Free	Evolved from XNA Touch, used in a number of games, last official 2D only version released in June 2012.
Newton Game Dynamics	C++	✓	3D	Open	Free	For simulating realistic rigid bodies in game and other real-time applications.
Open Dynamics Engine	C, C++	✗	3D	Open	Free	Rigid body dynamics simulation and collision detection engine.
Physics Abstraction Layer	C, C++	✓	3D	Open	Free	API that supports multiple physics engines; physics engine wrapper.
PhysX	C	✓	3D	Open	Free	Physics engine middleware by Nvidia, used in a variety of games.
Phyz	DirectX	✗	2.5D	Open	Free	Public domain 2.5D physics engine.
Python	Python	✓	2D, 3D	Open	Free	Open source programming language, multiple libraries for physics available, well documented and widely used, can be used for quick prototyping.
Simulation Open Framework Architecture	C++, Python	✓	3D	Open	Free	Primarily targeted at real-time physics simulation with emphasis on medical simulation, can be used as a prototyping tool or physics engine.
Unity	C#	✓	2D, 3D	Closed	Free	Popular free game engine, well documented and widely used, cross-platform, can be used for quick prototyping.
Unreal Engine	C++	✓	3D	Closed	Free	Popular free game engine, well documented and widely used, cross-platform.
Vortex	C++	✓	3D	Closed	?	Pricing unknown, real-time physics engine for rigid body dynamics with collision detection.

Table 2.4-1: Comparison of physics engines

3.1 Prototype swarm simulation design in Unity

The first simulation prototype was done up in Unity (Video game engine), owing to Unity having most of the requirements already implemented in its core: 2D/3D graphical visualisation, geometric analysis, and physics/motion simulation. It is programmed in C# and is cross-platform.

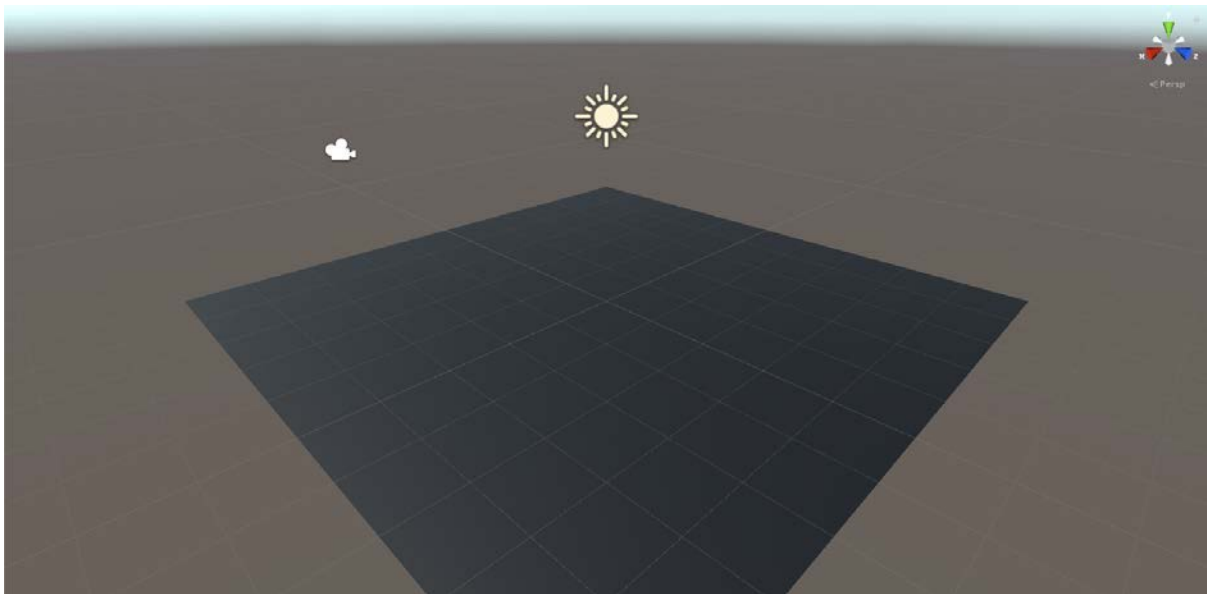


Figure 3.1-1: Initial setting of a scene for the swarm simulation in Unity

The scene features an open 3D world with an impassable plane fixed at origin of 10×10 unit(s). There are no other restrictions on mobility; Boids have free rein on where to move around the world.

3.1.1 3D Boid model

A 3D model of a Boid object was created using Autodesk 3ds Max 2017. It has a tetrahedron shape which allows the direction and rotation to be obvious in 3-dimensional space, with the size of $16 \times 6.9 \times 6$ units which is then scaled by 0.027305 to mimic the size of average birds and fishes $[(16 \times 6.9 \times 6) \times 0.027305] = (0.43688 \times 0.1884045 \times 0.16383)$ units.

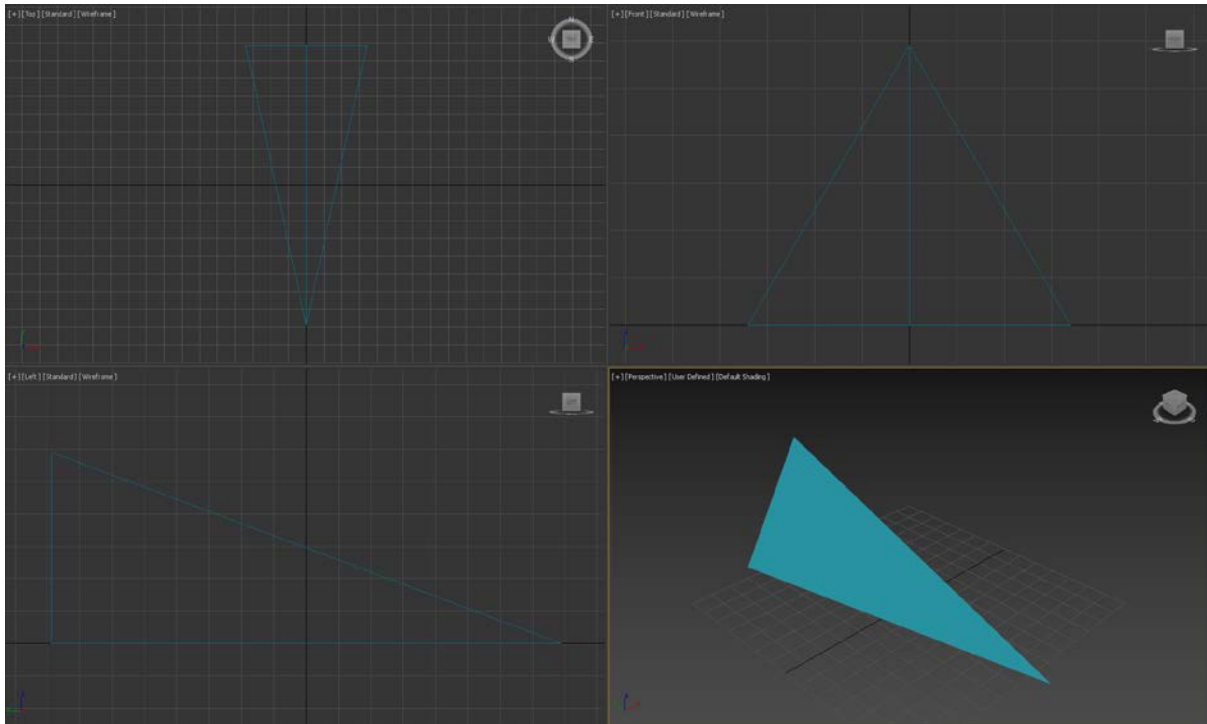


Figure 3.1-2: 3D model of Boid prototype

The model is deliberately made simple to reduce computational workload and improve the speed of simulation(s).

3.1.2 Boid Algorithm

The pseudocode of the actual implementation of the Boid themselves are as follows; all variable units are in vectors. Apart from the 3 key steering behaviours in the original Boid Algorithm: separation, cohesion, and alignment. Other steering behaviours can be introduced too, extending from the Boid Algorithm. Randomness is also introduced to prevent the Boids from becoming permanently stationary when and if all the steering behaviours results to *null*.

Snippet of the Boid Algorithm pseudocode:

```
while(true) {
    foreach(Boid bJ in Boids) {
        position = bJ.position;
        rotation = bJ.rotation;

        separation = Separation(bJ);
        alignment = Alignment(bJ);
        cohesion = Cohesion(bJ);
```



```

        additional_steering_behaviour = AdditionalSteeringBehaviour(bJ);
        ...
        additional_steering_behaviours = AdditionalSteeringBehaviours(bJ);

        target = separation + alignment + cohesion;
        target += additional_steering_behaviour + ... + additional_steering_behaviours;
        target += random;

        bJ.position += target.normalised * speed * Time.deltaTime;
    }
}

Vector Separation(Boid bJ) {
    position = bJ.position;
    c = 0;

    foreach(Boid b in Boids) {
        if(b != bJ) {
            boidPosition = b.position;
            if(distance(boidPosition, position) <= separationDistance) {
                c -= boidPosition - position;
            }
        }
    }

    return c;
}

Vector Alignment(Boid bJ) {
    position = bJ.position;
    pvJ = 0;
    neighbours = 0.0;

    foreach(Boid b in Boids) {
        if(b != bJ) {
            boidPosition = b.position;
            if(distance(boidPosition, position) <= alignmentDistance) {
                pvJ += b.velocity;
                neighbours++;
            }
        }
    }

    return pvJ if neighbours < 1 else (pvJ / neighbours);
}

Vector Cohesion(Boid bJ) {
    position = bJ.position;
    pcJ = 0;
    neighbours = 0.0;

    foreach(Boid b in Boids) {
        if(b != bJ) {
            boidPosition = b.position;
            if(distance(boidPosition, position) <= cohesionDistance) {
                pvJ += boidPosition;
                neighbours++;
            }
        }
    }

    return pcJ if neighbours < 1 else ((pcJ / neighbours) - position);
}

```

Table 3.1-1: Code snippet of the Boid Algorithm pseudocode

3.1.3 Visual inspection

One of the reasons Unity was used for prototyping is the 2D/3D visualisation tools already built-in. The final swarm simulation prototype demonstrated that the code is sane and does exhibit basic and minimal swarming behaviours in the scenarios.

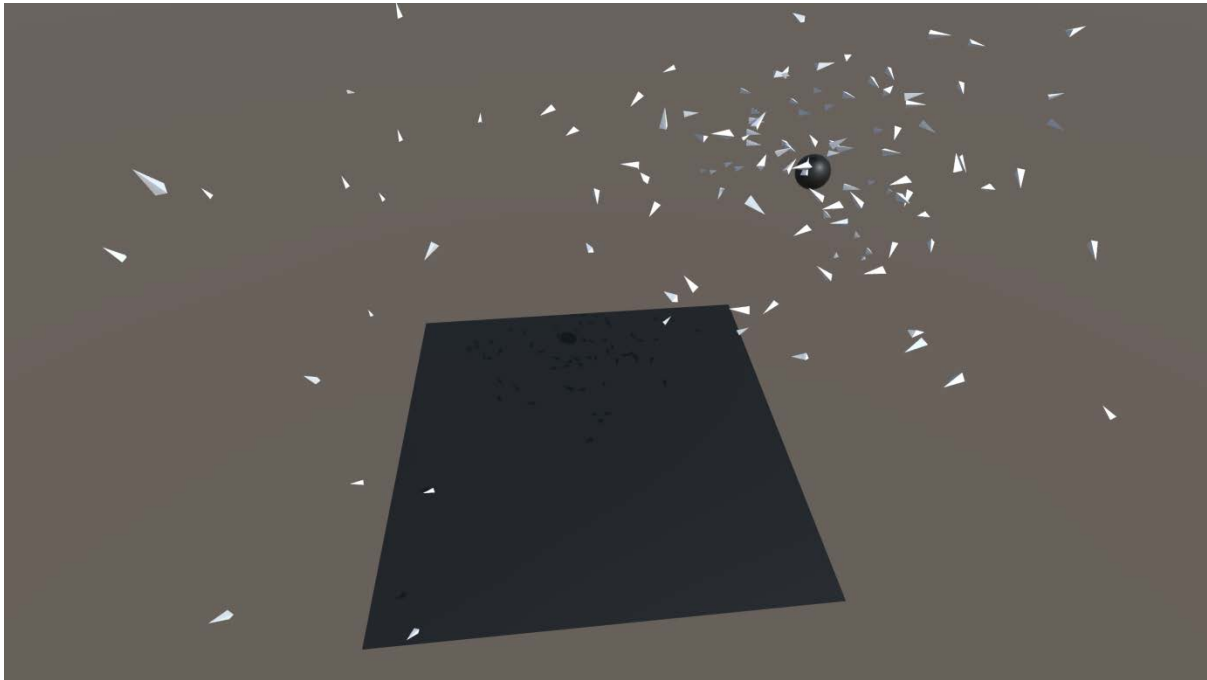


Figure 3.1-3: Early prototype swarm simulation in Unity

Observe the Boids (white tetrahedrons) swarming around a prey (black sphere) while maintaining a reasonable separation and cohesion from each other.

3.1.4 Optimisations

A few key optimisations techniques were discovered during the developmental phase, they are used in tandem to speed up the simulation and reduce the computational time required dramatically.

The optimisations are listed as follows (in descending order of speedup):

1. Unity Job System: Entity Control System (ECS); Multi-processing

- Fully utilising the physical cores and threads with a scheduler resulted in a dramatic increase in speedup.
2. CUDA / Compute Shaders
 - Allowing Graphics Processing Unit (GPU) to compute multiple data with a single instruction, i.e. SIMD (Single Instruction Multiple Data).
 3. Square magnitude comparison
 - Square magnitude calculations to compare distances instead of slower square root function.
 4. Preferring memory over CPU
 - Storing variables in memory/cache to reduce the time to fetch data.
 5. Object Pooling
 - Initialise all objects at the start and visually hide/unhide them as necessary; no additional initialisation or destruction of objects in memory.
 6. Spatial partitioning
 - Spatially partitioning Boids in regions to reduce computation.

3.1.5 Usability of Unity / Issues

Although Unity was a good base for prototyping, it was not a viable option in the end as it could not run numerous simulations quickly within reasonable time with current implementation. This is due to the fact that the optimisations implemented in the algorithm are not sufficient enough to be able to be fully utilised by the GA.

Despite the fact that Unity is actually able to run a swarm simulation astonishingly fast, using multi-processing and ECS (Entity Control System), the technologies of it are out of the scope and time of this study. Hence, alternatives to simulate the swarm are much more recommended. Although Unity was not used, it gave good insight during the prototyping process and led to the discovery of many optimisations techniques that were used later.

3.2 Swarm simulation design in Python

In the end, Python was chosen as a base for simulating the swarm after prototyping in Unity.

Some reasons for settling on Python for the simulation is because Python:

- is able to achieve a high optimisation using vectorisation.
- is easier to use as compared to Unity's ECS (Entity Control System).
- has many libraries for different functions preinstalled, or available to download.
- is able to tightly couple with the Genetic Algorithm (GA), as the GA is to be coded in Python as well.

3.2.1 Visual inspection of the swarm

Undeterred by Python not having a built-in 2D/3D graphical emulation tools, Python's matplotlib and Animation libraries were used instead to emulate and animate 1D, 2D, and 3D to verify if the algorithm is still sane.

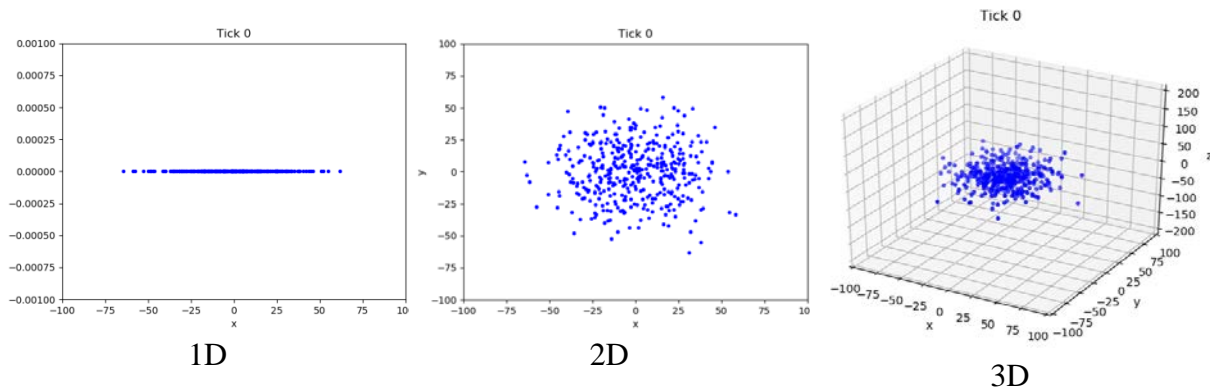


Figure 3.2-1: Visualisation of Python's swarm simulation in different dimensions

3.2.2 Data Structure

A different data structure from the Unity prototype is introduced for Python. Before, in Unity, each Boid was an object that has its own position, rotation, and velocity. For the newer data

structure in Python, every Boid's position, rotation, and velocity is stored separately together based on the principle of separations of concerns, i.e. all the Boids' positions are stored together as an array, same with rotations, and velocities, etc.

```
positions # (n,dimension) array of (n × dimension) containing all n Boids' positions.
rotations # (n,dimension) array of (n × dimension) containing all n Boids' rotations.
velocities # (n,dimension) array of (n × dimension) containing all n Boids' velocities.
```

3.2.3 Optimisations

Further optimisations are required for the simulation and Genetic Algorithm (GA) to run in reasonable time as the GA would run many generations of iterations to provide decent tuning of parameters. A great majority of the time was spent to optimise every single line of code in the simulator.

For instance, having a more memory-friendly data structure enables the use of the highly optimised numpy's function, einsum, to perform vector calculations extremely quickly and shorten computational time. This is because einsum uses compiled code and SIMD (Single Instruction Multiple Data) optimisations. Separation of concerns also allows for extension to fully leverage multiple processors using the multiprocessing library.

A few optimisations that were implemented in the simulator are:

- Reusing variables
 - Simple and effective yet often neglected, storing reusable variables such as Boids' positions in memory allows data to be reused in numerous calculations further on reducing repeated read/writes.
- Vectorisation
 - Storing all Boids' position, rotation, and velocity separately as Lists permits vectorisation. Treating the Lists as arrays enables fast array manipulation(s).
- Numpy's optimised functions

- Predefined numpy functions such as `numpy.array`, `numpy.einsum`, `numpy.sum`, `numpy.mean`, etc. are much more optimised than traditional code with similar logic.
- `numpy.einsum`
 - An integral part of optimisation, providing a speedup of more than 300% compared to traditional array manipulations. `numpy.einsum` employs SSE2 (Streaming SIMD Extensions 2) and SIMD (Single Instruction Multiple Data) which makes it run as fast as compiled code.

3.2.4 Parameters of the swarm simulation

Aside from the parameters in the original Boid model, newer parameters were introduced to add support for predators and preys. There are a total of 11 parameters introduced and a list of the parameters and their description is listed as follows:

Parameter(s)		Description
Radii		
0	<code>radii_separation</code>	The radius from the center of a Boid in which other Boids that are within <code>radii_separation</code> will influence the separation of the Boid.
1	<code>radii_alignment</code>	The radius from the center of a Boid in which other Boids that are within <code>radii_alignment</code> will influence the alignment of the Boid.
2	<code>radii_cohesion</code>	The radius from the center of a Boid in which other Boids that are within <code>radii_cohesion</code> will influence the cohesion of the Boid.
3	<code>radii_predator</code>	The radius from the center of a Boid in which Predators that are within <code>radii_predator</code> will influence the predator reflex of the Boid.

4	radii_prey	The radius from the center of a Boid in which Predators that are within radii_prey will influence the prey reflex of the Boid.
Weights		
5	weights_separation	The percentage of separation reflex contributing to the steering behaviours of a Boid.
6	weights_alignment	The percentage of alignment reflex contributing to the steering behaviours of a Boid.
7	weights_cohesion	The percentage of cohesion reflex contributing to the steering behaviours of a Boid.
8	weights_predator	The percentage of predator reflex contributing to the steering behaviours of a Boid.
9	weights_predatorBoost	A percentage multiplied on top of predator reflex towards the steering behaviours of a Boid.
10	weights_prey	The percentage of prey reflex contributing to the steering behaviours of a Boid.
11	weights_preyBoost	A percentage multiplied on top of prey reflex towards the final target vector of a Boid.

Table 3.2-1: Parameters of a swarm

3.2.5 JSON

To implement modularity, parameters of the swarm are initialised from JSON file(s) read by the swarm simulation. A log of the simulation run can also be saved into a JSON file for high-level debugging purposes.

```
{
  "boidSize": 0.10922,
  "radii": {
    "separation": 2.0,
    "alignment": 4.0,
    "cohesion": 8.0,
    "predator": 4.0,
    "prey": 8.0
  },
  "weights": {
    "separation": 1.0,
    "alignment": 1.0,
    "cohesion": 1.0,
    "predator": 1.0,
    "predatorBoost": 2.0,
    "prey": 1.0,
    "preyBoost": 2.0
  },
  "maximumSpeed": 5.4934993590917414392968320820363
}
```

Table 3.2-2: Sample of a JSON parameter file

```
import json

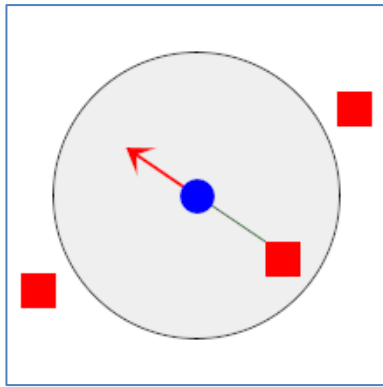
inputFilename = "parameters.json"
with open(inputFilename, "rt") as input_file:
    inputs = json.load(input_file) # parameters dict
```

Table 3.2-3: Code snippet for reading JSON parameter files

3.2.6 Predators of the Boids

Predators were introduced to the simulation, extending from the original Boid Algorithm. The purpose of introducing predators is to coax the Genetic Algorithm (GA) into optimising the swarm for predator avoidance. Doing so, the simulation may emulate behaviours like the bait ball, exhibited by swarms in the presence of predators.

The additional steering behaviour for predator avoidance steers an individual Boid to move away from the nearest predator within the radius.



Predator: steer to move away
from the nearest predator

Figure 3.2-2 Boid's predator steering behaviour

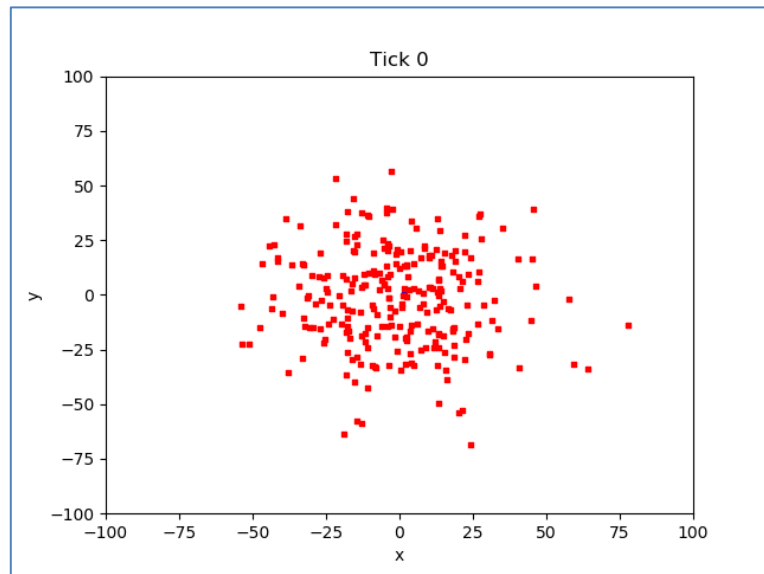
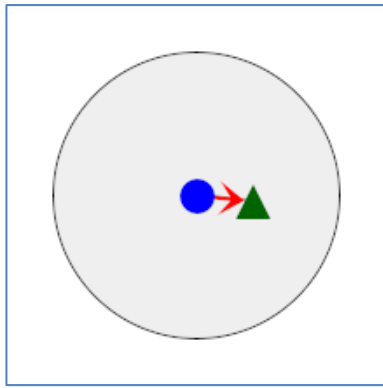


Figure 3.2-3: A swarm of 256 predators in the swarm simulation

3.2.7 Preys of the Boids

Preys were introduced into the mix as well, extending from the algorithm. The purpose of introducing preys is to coax the Genetic Algorithm (GA) into optimising the swarm for prey seeking also. The simulation may emulate behaviours exhibited by swarms in the presence of preys. The additional steering behaviour for prey seeking steers an individual Boid to move towards the nearest predator within the radius.



Prey: steer to move towards the
nearest prey

Figure 3.2-4 Boid's prey steering
behaviour

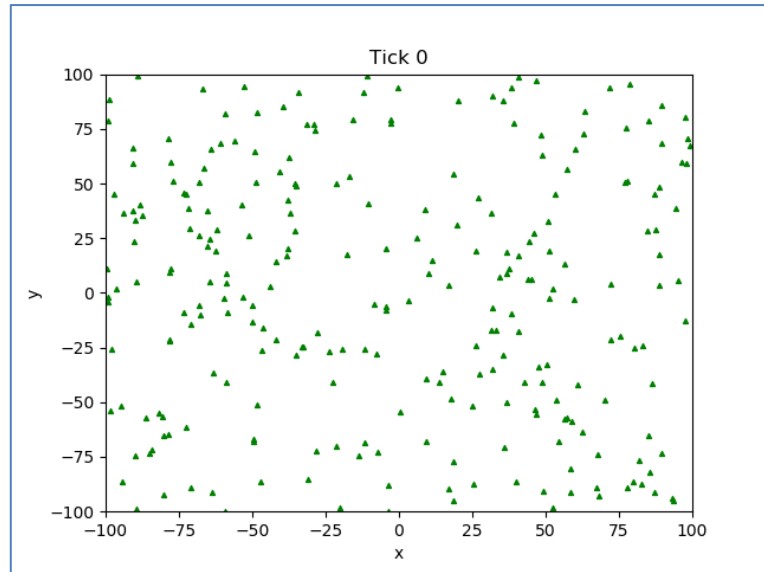


Figure 3.2-5: Randomly scattered 256 preys in the swarm
simulation

3.2.8 Predator and prey relationship

Predators and preys are only relative to Boids and do not influence each other, i.e. predators do not hunt the swarm's prey and preys do not influence the swarm's predators.

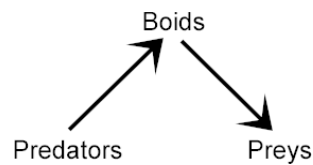


Figure 3.2-6: Predator and prey relation to Boids

Predators solely represent the swarm's predator and preys solely represent the swarm's prey.

3.2.9 Types of predators and preys

In addition to the introduction of predators and preys, different types of predators and preys were also encompassed into the simulation. The 2 types for predators are: hunting (1) or patrolling (2), and the 2 types for preys are: random (1) or along predator's route (2).

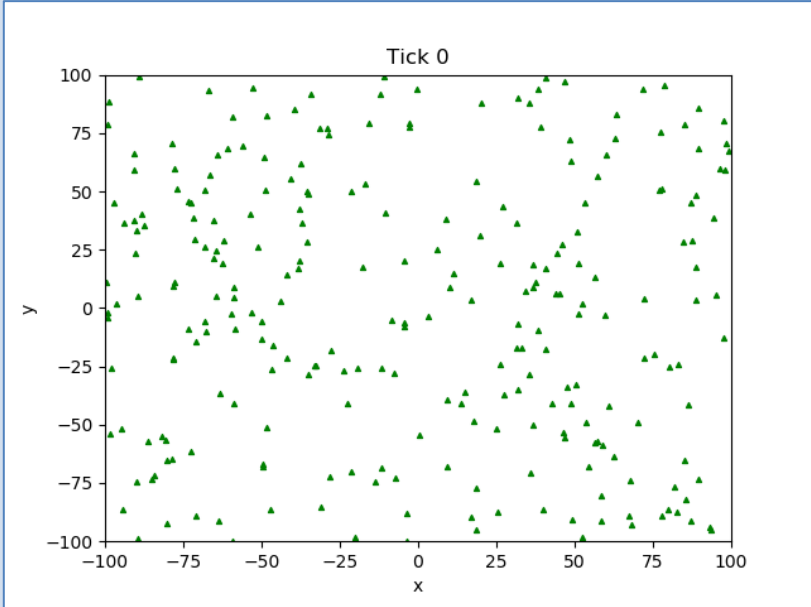
The descriptions of predator types are as follows:

Type	Description
Hunting (1)	<p>Predators move towards nearest Boid.</p> <div data-bbox="360 506 887 1028">A diagram illustrating a predator's hunting behavior. A red square represents the predator, and three blue circles represent Boids. A red arrow points from the predator to the closest Boid, indicating the direction of movement.</div> <p>Figure 3.2-7: Predator's hunting behaviour</p> <p>If a predator is already in the same position as a Boid, it will not move.</p>
Patrolling (2)	<p>Predator follows a defined route, patrolling through waypoints set.</p> <div data-bbox="360 1335 1174 1939"><p>Example of predator type 2: patrolling. Route: $[-75, 75], [75, -75], [75, 75]$, 3 waypoints</p>A graph showing a patrolling route on a 2D plane with x and y axes ranging from -100 to 100. The route is a dashed line forming a triangle with vertices at (-75, 75), (75, -75), and (75, 75). The segments are labeled 1, 2, and 3. Waypoint 1 is at (-75, 75), Waypoint 2 is at (75, -75), and Waypoint 3 is at (75, 75). Arrows indicate the direction of travel along the route.</div> <p>Figure 3.2-8: Predator's patrolling behaviour</p>

	After reaching the final waypoint in the route, the predator will wrap around to the initial waypoint.
--	--

Table 3.2-4: Description of predator types

The descriptions of prey types are as follows:

Type	Description
Random (1)	<p>Preys respawn randomly throughout the scene every 200 ticks.</p> <div></div> <p>Figure 3.2-9: Prey's random behaviour</p>
Along predator's route (2)	<p>Every 200 ticks, prey respawns near the predator's current route.</p>

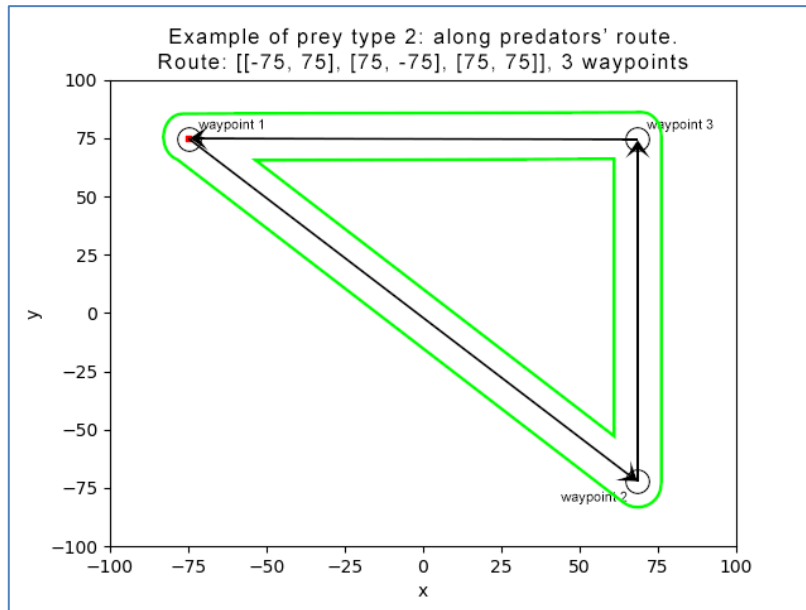


Figure 3.2-10: Preys' along predator's route behaviour

In the example above, for every 200 ticks, preys respawn anywhere within the green outline of the predator's route. The radius of the prey's spawn from the predator's route can be defined, however is defined arbitrarily for this project.

Table 3.2-5: Description of prey types

3.2.10 Swarm specialisations

Because it is not trivial for a typical Genetic Algorithm (GA) to generate a swarm that performs well in everything, different specialisations were introduced into the simulation for the GA. Specialisations represent basic tasks that swarms would perform such as, having a fast moving swarm, avoiding predators, seeking preys, etc. The concept is that with different specialisations, the GA would ideally combine different swarms performing well in their specialisations together, thus producing newer swarms which should perform relatively well in all specialisations.

This is slightly similar to a divide-and-conquer approach, where complex swarm behaviours are broken down into simpler behaviours for tuning and then recombined to reproduce the complex behaviours.

This project features 6 specialisations, described below:

Specialisation		Description
0	“velocities”	Swarm specialises in staying a fast moving swarm; high velocity.
1	“predators”	Swarm specialises in having less predators within the Boids’ radius. Is used as an analog to DBSCAN.
2	“preys”	Swarm specialises in having less preys within the Boids’ radius. Is used as an analog to DBSCAN.
3	“distances”	Swarm specialises in maintain swarm cohesion in the presence of predators.
4	“distancesPredator”	Swarm specialises in staying further from predators in the presence of preys.
5	“distancesPrey”	Swarm specialises in staying closer to preys in the presence of predators.

Table 3.2-6: Description of specialisations

3.2.11 Scene and Scenarios

For different specialisations, different scenarios are presented to the swarm catering to different specialisations. The specialisations represent simpler behaviours of a swarm for the GA to optimise. Every scene has a n number of Boids, and may or may contain predators or preys with different types.

A description of the scenes and its parameters are given as follows:

Scene	n	Predators (Type)	Preys (Type)
scene_velocities.json	400	0	0
scene_predators.json	400	4 (1)	0
scene_preys.json	400	0	4 (1)
scene_distances.json	400	4 (1)	4 (1)
scene_distancesPredator.json	400	1 (2)	1 (1)
scene_distancesPrey.json	400	1 (2)	1 (2)
Predator types: Hunting (1), and Patrolling (2) Prey types: Random (1), and Along Predator's Route (2)			

Table 3.2-7: Scenario descriptions

Visualisations of aforementioned scenarios are illustrated as follows:

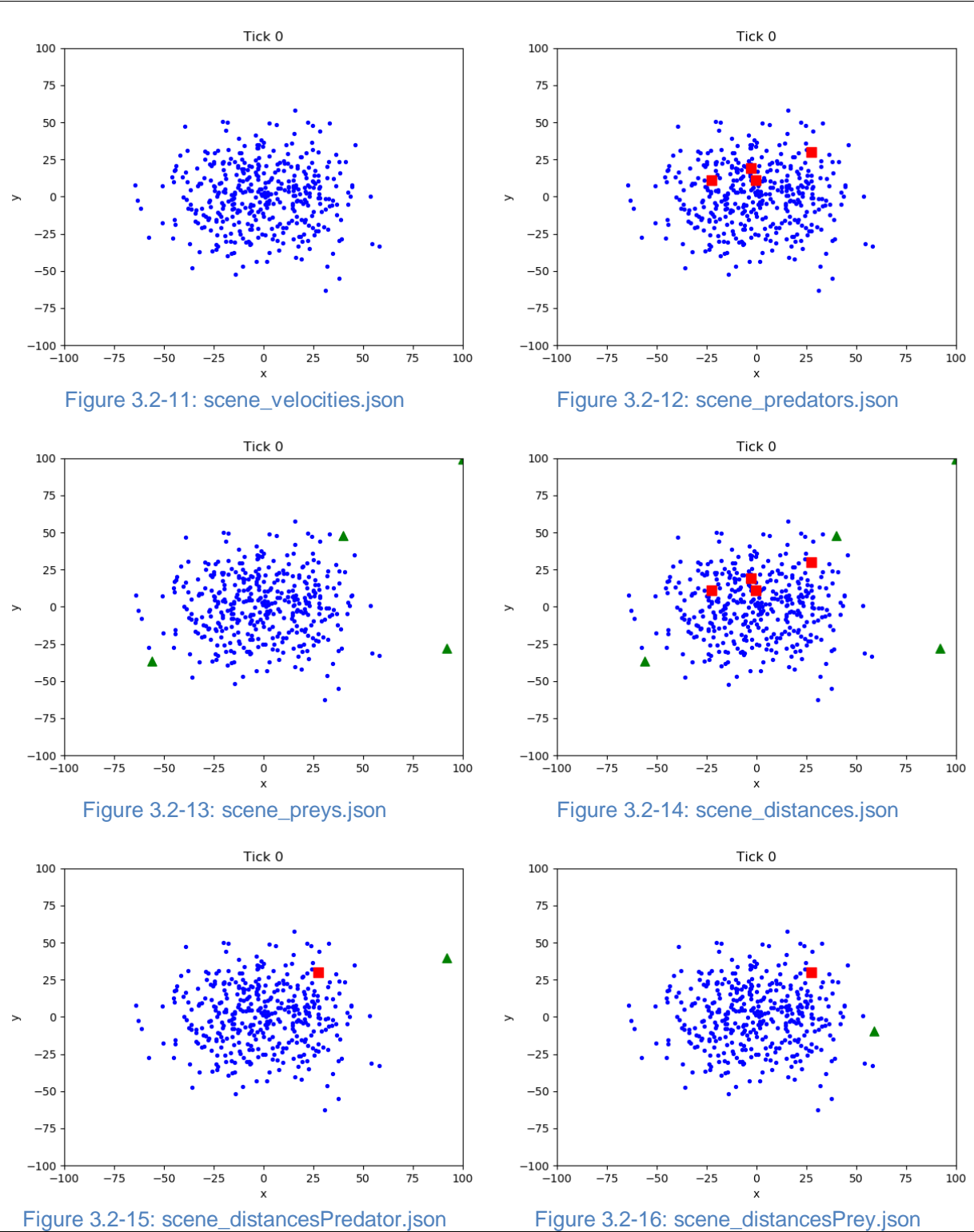


Table 3.2-8: Visualisation of different scenes/scenarios

Predators and preys were enlarged in the figures above for illustration purposes

3.2.12 Measurements

To measure the performance of the swarms, measurements were taken at every tick and averaged together after the final tick. The descriptions and metrics of the specialisations are listed below:

Measurement	Metrics
velocities	<p>The average velocity of the swarm, higher is better</p> $f_0 = \left(\sum_{i=0}^n Boid_i^{velocity} \right) \div n$
predators	<p>The average number of predators in the Boids' radii, lower is better.</p> $f_1 = \left(\sum_{i=0}^n (predators Boid_i^{radius}) \right) \div (n \times nPredators)$
preys	<p>The average number of preys in the Boids' radii, higher is better</p> $f_2 = \left(\sum_{i=0}^n (preys Boid_i^{radius}) \right) \div (n \times nPreys)$
distances	<p>The average distance between each and every Boid, lower is better.</p> $f_3 = \left(\sum_{j=0}^n \sum_{i=0}^n (Boid_j^{position} - Boid_i^{position}) \right) \div n^2$
distancesPredator	<p>The average distance between each and every Boid to the nearest relative predator, higher is better.</p> $f_4 = \left(\sum_{j=0}^{nPredators} \sum_{i=0}^n \min(Predator_j^{position} - Boid_i^{position}) \right) \div (nPredators \times n)$
distancesPrey	<p>The average distance between each and every Boid to the nearest relative prey, lower is better.</p>

	$f_5 = \left(\sum_{j=0}^{nPreys} \sum_{i=0}^n \min(Prey_j^{position} - Boid_i^{position}) \right) \div (nPreys \times n)$
--	--

Table 3.2-9: Performance measurement descriptions

To normalise the metrics, i.e. higher scores being better for all measurements, measurements that were better having lower scores was inverted. Metrics that were inverted are: predator, distances, and distancesPrey.

The new equations for the normalised metrics are given below:

Measurement	Metrics
velocities	$f_0 = \left(\sum_{i=0}^n Boid_i^{velocity} \right) \div n$
predators	$f_1 = \frac{1}{\left(\sum_{i=0}^n (predators Boid_i^{radius}) \right) \div (n \times nPredators)}$
preys	$f_2 = \left(\sum_{i=0}^n (preys Boid_i^{radius}) \right) \div (n \times nPreys)$
distances	$f_3 = \frac{1}{\left(\sum_{j=0}^n \sum_{i=0}^n (Boid_j^{position} - Boid_i^{position}) \right) \div n^2}$
distancesPredator	$f_4 = \left(\sum_{j=0}^{nPredators} \sum_{i=0}^n \min(Predator_j^{position} - Boid_i^{position}) \right) \div (nPredators \times n)$
distancesPrey	$f_5 = \frac{1}{\left(\sum_{j=0}^{nPreys} \sum_{i=0}^n \min(Prey_j^{position} - Boid_i^{position}) \right) \div (nPreys \times n)}$

Table 3.2-10: Normalised measurement for performance

The normalisations, having the same metrics across the board, were done to reduce the complexity and overheads for the GA from a consistent metric.

3.2.13 Fitness

To integrate the swarm simulation with the Genetic Algorithm (GA), a function, `__run__(parameters: dict, scene_file: str)`, is defined. The function takes in parameters of a swarm and a scene file, which initialises the variables in the simulation before the simulation begins. After the simulation is done, the function returns the names of specialisations and the respective fitness scores.

The shortened Python code for this is given below:

```
def __run__(parameters: dict, scene_file: str) -> (tuple, [float]):
    __global__() # initialise parameters

    # run simulation
    for tick in range(ticks):
        __update__(tick)

    # return measurements
    return (
        (
            "velocities",
            "predators",
            "preys",
            "distances",
            "distancesPredator",
            "distancesPrey"
        ),
        output["fitness"]
    )
```

Table 3.2-11: Code snippet to return the evaluation of a swarm simulation

Chapter 4 Designing the Genetic Algorithm (GA)

The difference between typical Genetic Algorithms and the GA implemented is that typical Genetic Algorithms optimises a single task/specialisation, whereas the GA implemented in this project optimises many specialisations, hence the name “Multi-Objective Genetic

Algorithm”. It means that the GA will try to optimise a swarm for many objectives at once and this is a nontrivial task.

As mentioned before, this is slightly similar to a divide-and-conquer approach, where complex swarm behaviours are broken down into simpler behaviours for tuning and then recombined to reproduce the complex behaviours.

4.1 Initialisation

The population is initialised with random values having a search space of 100.0, this limits the value to range from 0.0 to 100.0 for every gene, using the `search_space` variable and `numpy.random.rand`. Each individual has 12 genes, which corresponds to the number of parameters a swarm has. The population’s specialisation is initialised after, with random values between 0 and 5 to represent the specialisations (0 being the 1st specialisation and 5 being the 6th specialisation).

The fitness of the population is then evaluated with respects to the population’s respective specialisation, i.e. a swarm specialising in specialisation 0 will be evaluated using the scene/scenario specific for specialisation 0.

```
search_space = 100

population = numpy.random.rand(n, properties) * search_space
populationSpecialisation = numpy.random.randint(0, nSpecialisations, n)
for x in range(n):
    populationFitness[x] = fitness(population[x], populationSpecialisation[x])
```

Table 4.1-1: Code snippet for Genetic Algorithm’s initialisation

A visualisation of the populations’ genomes is depicted below, rows represent individuals and columns represent genes:

	0	1	2	3	4	5	6	7	8	9	10	11
0	2.0	4.0	8.0	4.0	8.0	1.0	1.0	1.0	1.0	2.0	1.0	2.0
1	70.9651...	90.0142...	53.4115...	24.7293...	67.1806...	56.1729...	54.2559...	89.3447...	84.2779...	30.6012...	63.1169...	68.0238...
2	97.0427...	89.3567...	94.2425...	64.2225...	61.4647...	22.7683...	48.6031...	80.7219...	84.4220...	53.4680...	75.7798...	49.9676...
3	85.0327...	61.9696...	86.1614...	23.1697...	40.2218...	62.4375...	14.3036...	12.2798...	41.6829...	55.6882...	94.1419...	40.9259...
4	73.6751...	99.5450...	91.6664...	0.20232...	97.1331...	88.9048...	69.9488...	9.75246...	57.3429...	82.0037...	56.0891...	35.0762...
5	54.3499...	87.9589...	11.4096...	3.14388...	95.2810...	28.8743...	44.1949...	25.9021...	59.6891...	65.5286...	27.5695...	85.7972...
6	88.8724...	28.5060...	65.9560...	97.2120...	79.6874...	17.9464...	78.4672...	97.0127...	36.2811...	8.78860...	34.3833...	57.1109...
7	16.5512...	63.1615...	6.55655...	91.2874...	8.86539...	19.9387...	47.1952...	44.0997...	76.5551...	1.27435...	68.0662...	27.5602...
8	60.3979...	54.5972...	20.9789...	13.6122...	76.9162...	47.4584...	51.9113...	94.4193...	64.3525...	43.8250...	46.3787...	24.8831...
9	25.6759...	2.29940...	93.2405...	39.2259...	64.8287...	35.6739...	11.0440...	44.8558...	46.8923...	5.81458...	81.6603...	28.9341...
10	65.1091...	15.3705...	52.8271...	68.2148...	71.1799...	74.2616...	57.8734...	38.0499...	76.5049...	5.12362...	91.2336...	43.7043...
11	63.5951...	82.1420...	17.6210...	55.1791...	54.8406...	92.0328...	83.3286...	66.0019...	68.9698...	28.6245...	90.1868...	5.52238...
12	42.1096...	78.6895...	92.7283...	15.7440...	55.8584...	45.1171...	74.8985...	83.9253...	4.55746...	50.5643...	58.8814...	6.17683...
13	2.49223...	59.2589...	73.9377...	0.96839...	98.2292...	4.58936...	56.6805...	9.94523...	35.0336...	39.4463...	38.7282...	40.2699...
14	92.9916...	94.2137...	18.0240...	73.0513...	68.0181...	88.3898...	58.0875...	69.8774...	25.1284...	81.2918...	56.2908...	31.0271...
15	37.4634...	82.1816...	90.5369...	91.8238...	22.9450...	14.9679...	76.4679...	32.9944...	92.1893...	39.9757...	17.8014...	9.80384...
16	12.3688...	62.8657...	47.5658...	70.2289...	10.2262...	9.26771...	72.5037...	37.1146...	28.6394...	87.9426...	3.45965...	1.82051...
17	31.7715...	37.5128...	99.6912...	81.5052...	64.1503...	15.3590...	73.3059...	28.7393...	36.9132...	1.67317...	85.2986...	30.9490...
18	56.2927...	9.09326...	79.2601...	2.92094...	5.57266...	72.0349...	3.11695...	48.2600...	71.1529...	48.0504...	80.1016...	54.8874...
19	60.2903...	75.9279...	1.15620...	92.8798...	61.8509...	80.0932...	1.19527...	56.3375...	20.5771...	27.0741...	38.1123...	99.9151...
20	99.0125...	76.3854...	7.18335...	97.4027...	98.5531...	24.0194...	50.1016...	69.5637...	26.5187...	96.5439...	89.8595...	52.5484...
21	50.7520...	43.8633...	1.00641...	85.2153...	68.1227...	95.5142...	24.6421...	13.8632...	67.7384...	63.5503...	61.3513...	92.4742...
22	5.97972...	72.1803...	97.3743...	6.49425...	59.3962...	15.8315...	99.4671...	86.1181...	36.5023...	99.5060...	32.3569...	42.0257...
23	4.82870...	36.7363...	16.6266...	16.0044...	37.6225...	22.6261...	29.2553...	41.8605...	80.5852...	1.09947...	14.9610...	97.8271...
24	4.09506...	0.02187...	86.8306...	10.6728...	88.5054...	65.2676...	17.0611...	1.62256...	14.4236...	17.0610...	89.2859...	20.0023...
25	46.7160...	15.4922...	19.1128...	35.3103...	12.8117...	40.2754...	85.0856...	27.8612...	42.4903...	72.2466...	65.6307...	31.9278...

Figure 4.1-1: Visualisation of genomes for a population

4.1.1 Seeding

It is also possible to “seed” the population with predetermined known good values as a way to speed up and optimise the GA. An individual in the population was seeded with arbitrary parameters that roughly exhibited some swarming behaviours during simulation development.

The values seeded are as follows:

```
seed = population[0]
seed[0] = 2
seed[1] = 4
seed[2] = 8
seed[3] = 4
seed[4] = 8
seed[5] = 1
seed[6] = 1
seed[7] = 1
seed[8] = 1
seed[9] = 2
seed[10] = 1
seed[11] = 2
```

Table 4.1-2: Example of seeding the population

0	1	2	3	4	5	6	7	8	9	10	11
2.0	4.0	8.0	4.0	8.0	1.0	1.0	1.0	1.0	2.0	1.0	2.0

Figure 4.1-2: Visualisation of genome for the seeded individual

4.2 Selection

Before every selection process, the population's fitness is first normalised in each specialisation. Individuals receive a fitness of 0 for other specialisations that they are not specialising in and the sum of the normalised fitness for each specialisation should add to 1.

$$\sum_{j=0}^{specialisations} \left(\sum_{i=0}^{population} (Population_i^{fitness} | specialisation_j) \right) = 1$$

A visualisation of the normalisation is depicted below, rows represent individuals and columns represent fitness:

	0	1	2	3	4	5
0	83.725343...	377.56421...	0.0177368...	0.0002123...	10565.167...	0.0001081...
1	1418.5482...	143.45449...	0.0580969...	0.0003191...	11843.489...	7.8382100...
2	1387.9108...	0.0	0.1039672...	0.0016518...	0.0	5.7861779...
3	1334.0932...	1.5638107...	0.0424432...	0.0001104...	5158.0405...	7.7975823...
4	1342.9768...	399.64192...	0.2033288...	5.4826027...	13672.937...	5.0474780...
5	1361.2478...	231.68780...	0.0	0.0001025...	8870.6076...	0.0
6	1417.7323...	4.1749809...	0.2710540...	9.8436023...	21178.827...	5.8003712...
7	1366.7648...	125.69876...	0.0	0.0009724...	21231.539...	0.0
8	1422.3000...	0.0	0.0	0.0634959...	0.0	0.0
9	1200.8074...	278.84906...	0.0363317...	0.0125368...	13149.455...	8.1060090...
10	1370.5258...	7.0588340...	0.0782128...	5.7565813...	13373.092...	6.2828203...
11	1428.3636...	1.0040835...	0.1584838...	0.2389422...	102.95937...	0.0002202...
12	1414.8261...	211.62298...	0.3838952...	0.0011882...	4361.2063...	0.0001053...
13	1405.8518...	399.70703...	0.3662634...	0.0004352...	18572.278...	4.6059281...
14	1364.2407...	0.0	0.0	4.5087709...	0.0	0.0
15	1428.84	0.9975086...	0.0431494...	1925.9069...	12.118618...	0.0002613...
16	1428.8399...	0.9976181...	0.0176904...	417.84838...	12.268757...	0.0008545...
17	1403.1277...	406.77412...	0.0	0.1475099...	10190.395...	0.0
18	991.58381...	0.0	0.0007196...	6.5088891...	0.0	6.2886155...
19	1391.2472...	0.0	0.0933010...	4.6541412...	0.0	4.8163547...
20	1428.6869...	1.0308937...	0.2384439...	0.0003350...	2780.2853...	5.2822946...
21	1425.0388...	1.0867991...	0.2273327...	0.0535769...	698.09520...	0.0001299...
22	1428.8188...	1.7830818...	0.1933483...	132.60156...	116.01164...	0.0002550...
23	1399.9815...	387.22263...	0.0156066...	0.0935323...	9021.4411...	0.0002189...
24	1373.0827...	165.17635...	0.3079598...	0.0001063...	15822.181...	6.0930244...

Figure 4.2-1: Population's fitness before normalisation

	0	1	2	3	4	5
0	0.0	0.0	0.0	0.0	0.0	0.0212917...
1	0.0	0.0	0.0	0.0	0.1012266...	0.0
2	0.0	0.0	0.0613894...	0.0	0.0	0.0
3	0.0	0.0	0.0	4.0699001...	0.0	0.0
4	0.0	0.0	0.0	0.0	0.1168629...	0.0
5	0.0	0.1695224...	0.0	0.0	0.0	0.0
6	0.0	0.0	0.0	3.6266675...	0.0	0.0
7	0.0	0.0919719...	0.0	0.0	0.0	0.0
8	0.0689112...	0.0	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0	0.1123887...	0.0
10	0.0	0.0	0.0	0.0	0.0	0.0123710...
11	0.0	0.0	0.0	0.0	0.0008799...	0.0
12	0.0	0.0	0.0	0.0004377...	0.0	0.0
13	0.0	0.0	0.0	0.0	0.0	0.0090691...
14	0.0660982...	0.0	0.0	0.0	0.0	0.0
15	0.0	0.0	0.0	0.0	0.0001035...	0.0
16	0.0	0.0	0.0	0.0	0.0	0.1682625...
17	0.0	0.2976305...	0.0	0.0	0.0	0.0
18	0.0	0.0	0.0004249...	0.0	0.0	0.0
19	0.0	0.0	0.0550913...	0.0	0.0	0.0
20	0.0	0.0	0.0	0.0001234...	0.0	0.0
21	0.0	0.0	0.0	0.0	0.0	0.0255844...
22	0.0	0.0	0.0	0.0	0.0009915...	0.0
23	0.0	0.0	0.0	0.0	0.0771064...	0.0
24	0.0	0.0	0.0	3.9199751...	0.0	0.0

Figure 4.2-2: Population's fitness after normalisation

```

population_fitness = numpy.copy(populationFitness)
population_fitness[numpy.arange(len(population)), populationSpecialisation] = 0
population_fitness = populationFitness - population_fitness
population_fitness = population_fitness / numpy.sum(population_fitness, axis=0) # normalise

```

Table 4.2-1 : Code snippet for population fitness normalisation

Then, to perform selection, 2 random specialisations were chosen first:

```

a_specialisation, b_specialisation = {
    random.randint(0, nSpecialisations - 1), random.randint(0, nSpecialisations - 1)
}

```

Table 4.2-2: Code snippet for choosing 2 random specialisations

This is analogous to choosing parents from 2 random occupations for breeding, e.g. a doctor and a lawyer. The randomly chosen specialisations are then used for choosing the parents where a parent is chosen based on the fitness in their specialisation; s.t. “fitter” individuals with higher fitness have a higher probability of being chosen to breed. If the 2 chosen specialisations are the same, 2 distinct individuals are selected from the specialisation as parents.

```

population_fitness # normalised fitness for each specialisation

if a_specialisation != specialisation:
    a = numpy.random.choice(len(population), 1, p=population_fitness[:, a_specialisation])[0]
    b = numpy.random.choice(len(population), 1, p=population_fitness[:, b_specialisation])[0]
else:
    a, b = numpy.random.choice(len(population), 2, False, p=population_fitness[:, a_specialisation])
a, b = population[a], population[b]

```

Table 4.2-3: Code snippet for choosing parents from 2 random specialisations

4.3 Genetic operations

Selection results in 2 parents which are distinct individuals from the population. The genomes of the parents then go through crossover and mutation to produce 2 children to be added back into the population later.

```
a, b = crossover(a, b)
a, b = mutation(a), mutation(b)
```

Table 4.3-1: Code snippet of the combination and mutation of parents' genomes

4.3.1 Crossover

Crossover is a genetic operator to combine genetic information (genomes) of parents to produce new offspring.

There are many types of crossovers but the ones typically used are described in the following:

Crossover	Description
Single-point	<div><p>A random point is chosen on both the parents' genetic information and becomes a crossover point.</p><div><div><div>Parent A</div><div>Parent B</div><div>Children a</div><div>Children b</div></div><div><div><div>Parent A</div><div>Parent B</div><div>Parent A</div><div>Parent B</div></div><div>Crossover point</div></div></div><p>Figure 4.3-1: Single-point crossover</p><p>The genes of the parents are swapped (crossover) between parents after the crossover point, resulting in children that contain genetic information from both parents.</p></div>

	<p>It is possible, albeit with a slim chance, that single-point crossover does nothing at all if the crossover point chosen is at the starting or ending points.</p>																				
Two-point and <i>k</i> -point	<p>Similar to single-point crossover, 2 or <i>k</i> random point(s) are chosen on both the parents' genetic information and becomes crossover point(s)</p> <div><table><tr><td>Parent A</td><td>Parent A</td><td>Parent A</td><td>Parent A</td></tr><tr><td>Parent B</td><td>Parent B</td><td>Parent B</td><td>Parent B</td></tr><tr><td></td><td>Crossover point</td><td></td><td>Crossover point</td></tr><tr><td>Children a</td><td>Parent A</td><td>Parent B</td><td>Parent A</td></tr><tr><td>Children b</td><td>Parent B</td><td>Parent A</td><td>Parent B</td></tr></table></div> <p>Figure 4.3-2: Two-point crossover</p> <p>The genes of the parents are swarmed (crossover) between parents after each crossover point, resulting in children containing genetic information of both parents.</p> <p>It is possible, albeit with a slim chance, that the two-point crossover acts as a single-one crossover if the 2 crossover points are the same or does nothing at all if all the crossover points are the same.</p>	Parent A	Parent A	Parent A	Parent A	Parent B	Parent B	Parent B	Parent B		Crossover point		Crossover point	Children a	Parent A	Parent B	Parent A	Children b	Parent B	Parent A	Parent B
Parent A	Parent A	Parent A	Parent A																		
Parent B	Parent B	Parent B	Parent B																		
	Crossover point		Crossover point																		
Children a	Parent A	Parent B	Parent A																		
Children b	Parent B	Parent A	Parent B																		

Uniform	Each gene of the children is selected from the parents randomly and independently with equal probability, i.e. no bias for genomes with closer proximity to be together.		
	Parent A	Parent B	
	Parent A[0]	...	Parent A[n]
	Parent B[0]	...	Parent B[n]
Children a	$P(\text{Parent A}[0]) = \frac{1}{2}$ $P(\text{Parent B}[0]) = \frac{1}{2}$...	$P(\text{Parent A}[n]) = \frac{1}{2}$ $P(\text{Parent B}[n]) = \frac{1}{2}$
Children b	$P(\text{Parent A}[0]) = \frac{1}{2}$ $P(\text{Parent B}[0]) = \frac{1}{2}$...	$P(\text{Parent A}[n]) = \frac{1}{2}$ $P(\text{Parent B}[n]) = \frac{1}{2}$

Figure 4.3-3: Uniform crossover

Table 4.3-2: Description of crossover types

Parents can either produce 1 or 2 offspring, parents in this project produces 2 offspring.

The code snippet of the crossover functions is given below:

```
def crossover(a: numpy.ndarray, b: numpy.ndarray) -> (numpy.ndarray, numpy.ndarray):
    size = min(len(a), len(b))

    # Single-point
    if crossover_type < 1:
        lhs = numpy.random.randint(1 + 1, size=size) > 0
        rhs = lhs < 1

        children = (a * lhs + b * rhs, b * lhs + a * rhs)

    # Two-point
    elif crossover_type == 1:
        start_point = random.randint(0, size)

        children = (
            numpy.concatenate((a[:start_point], b[start_point:])), numpy.concatenate((b[:start_point],
a[start_point:])))
        )

    # Uniform
    elif crossover_type == 2:
        start_point = random.randint(0, size)
        mid_point = random.randint(start_point, size)

        children = (
            numpy.concatenate((a[0:start_point], b[start_point:mid_point], a[mid_point:])),
            numpy.concatenate((b[0:start_point], a[start_point:mid_point], b[mid_point:])))
        )
    else:
        return None

    return children
```

Table 4.3-3: Code snippet of crossover function

4.3.2 Mutation

Mutation is a genetic operator to sporadically alter or more genetic information (genes) information of an individual, it is similar to biological mutation.

There are many types of mutations but the ones commonly used are:

- Bit string mutation
- Flip bit
- Boundary
- Non-uniform
- Uniform

- Gaussian
- Shrink

This project features only the Bit string, Flip Bit, and Boundary mutation as described below:

Mutation	Description																											
Bit string mutation	<p>Each gene has a $\frac{1}{n}$ probability of mutating to a new gene, where $n =$ <i>number of genomes</i>.</p> <div><table><tr><td>$\frac{1}{n}$</td><td>$\frac{1}{n}$</td><td>...</td><td>$\frac{1}{n}$</td></tr></table><p>Figure 4.3-4: Bit string mutation</p></div> <p>An illustration of this is shown below, an individual with 11 genes and each gene has a $\frac{1}{11}$ chance of mutating.</p> <table><tr><td>$\frac{1}{11}$</td><td>$\frac{1}{11}$</td><td>$\frac{1}{11}$</td><td>$\frac{1}{11}$</td><td>$\frac{1}{11}$</td><td>$\frac{1}{11}$</td><td>$\frac{1}{11}$</td><td>$\frac{1}{11}$</td><td>$\frac{1}{11}$</td><td>$\frac{1}{11}$</td><td>$\frac{1}{11}$</td></tr></table> <p>Table 4.3-4: Example of Bit string mutation with 11 genes</p>	$\frac{1}{n}$	$\frac{1}{n}$...	$\frac{1}{n}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$												
$\frac{1}{n}$	$\frac{1}{n}$...	$\frac{1}{n}$																									
$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$	$\frac{1}{11}$																		
Flip Bit	<p>Each gene is inverted with the search space, e.g. <i>search space</i> = 100.0, <i>gene</i> = 69, <i>flipped gene</i> = 100 – 69 ≡ 31.</p> <p>E.g.</p> <p><i>Search space</i> = 100.0</p> <table><tr><td>Genome</td><td>96.00</td><td>69.95</td><td>99.99</td><td>22.01</td><td>36.11</td><td>73.98</td><td>99.65</td><td>31.63</td></tr><tr><td>Flipped Bit mutation</td><td>100.00 – 96.00</td><td>100.00 – 69.95</td><td>100.00 – 99.99</td><td>100.00 – 22.01</td><td>100.00 – 36.11</td><td>100.00 – 73.98</td><td>100.00 – 99.65</td><td>100.00 – 31.63</td></tr><tr><td>=</td><td>4.00</td><td>30.05</td><td>0.01</td><td>77.99</td><td>63.89</td><td>26.02</td><td>0.35</td><td>68.37</td></tr></table> <p>Table 4.3-5: Example of Flip Bit mutation with 8 genes</p>	Genome	96.00	69.95	99.99	22.01	36.11	73.98	99.65	31.63	Flipped Bit mutation	100.00 – 96.00	100.00 – 69.95	100.00 – 99.99	100.00 – 22.01	100.00 – 36.11	100.00 – 73.98	100.00 – 99.65	100.00 – 31.63	=	4.00	30.05	0.01	77.99	63.89	26.02	0.35	68.37
Genome	96.00	69.95	99.99	22.01	36.11	73.98	99.65	31.63																				
Flipped Bit mutation	100.00 – 96.00	100.00 – 69.95	100.00 – 99.99	100.00 – 22.01	100.00 – 36.11	100.00 – 73.98	100.00 – 99.65	100.00 – 31.63																				
=	4.00	30.05	0.01	77.99	63.89	26.02	0.35	68.37																				

Boundary	<p>Each gene is clipped with either the lower or upper boundary randomly.</p> <p>E.g.</p> <p><i>Lower boundary</i> = 31.0</p> <p><i>Upper boundary</i> = 69.0</p> <table border="1"> <tr> <td>Genome</td><td>96.00</td><td>69.95</td><td>99.99</td><td>22.01</td><td>36.11</td><td>73.98</td><td>99.65</td><td>31.63</td></tr> <tr> <td colspan="9"></td></tr> <tr> <td rowspan="2">Boundary mutation</td><td><i>Upper</i></td><td><i>Upper</i></td><td><i>Lower</i></td><td><i>Lower</i></td><td><i>Upper</i></td><td><i>Lower</i></td><td><i>Upper</i></td><td><i>Upper</i></td></tr> <tr> <td>69.0</td><td>69.0</td><td>99.99</td><td>31.00</td><td>36.11</td><td>73.92</td><td>69.00</td><td>31.63</td></tr> </table> <p>Table 4.3-6: Example of Boundary mutation with 8 genes</p>								Genome	96.00	69.95	99.99	22.01	36.11	73.98	99.65	31.63										Boundary mutation	<i>Upper</i>	<i>Upper</i>	<i>Lower</i>	<i>Lower</i>	<i>Upper</i>	<i>Lower</i>	<i>Upper</i>	<i>Upper</i>	69.0	69.0	99.99	31.00	36.11	73.92	69.00	31.63
Genome	96.00	69.95	99.99	22.01	36.11	73.98	99.65	31.63																																			
Boundary mutation	<i>Upper</i>	<i>Upper</i>	<i>Lower</i>	<i>Lower</i>	<i>Upper</i>	<i>Lower</i>	<i>Upper</i>	<i>Upper</i>																																			
	69.0	69.0	99.99	31.00	36.11	73.92	69.00	31.63																																			

Table 4.3-7: Description of mutation types

The code snippet of the mutation function is given below:

```

search_space = 100

def mutation(a: numpy.ndarray) -> numpy.ndarray:
    size = len(a)

    # Bit string mutation
    if mutation_type == 0:
        bit = numpy.random.rand(size) < (1 / size)

        child = (numpy.random.rand(size) * search_space) * (bit > 0) + a * (bit < 1))

    # Flip Bit
    elif mutation_type == 1:
        child = numpy.array(search_space) - a

    # Boundary
    elif mutation_type == 2:
        boundary = numpy.random.rand()

        if boundary < (1 / 3):
            child = numpy.clip(a, random.random() * search_space, None) # lower bound
        elif boundary < (2 / 3):
            child = numpy.clip(a, None, random.random() * search_space) # upper bound
        else:
            child = numpy.clip(a, random.random() * search_space, random.random() * search_space) #
lower and upper bound
    else:
        child = numpy.random.rand(size) * search_space

    return child

```

Table 4.3-8: Code snippet of mutation function

4.4 Heuristics

Heuristics can be utilised to increase the GA's robustness, parents selected for crossover that are very similar may be penalised. This promotes diversity by encouraging dissimilar parents to breed. There are 2 heuristic normalisation techniques that were fabricated and applied in this project: alpha (α) and beta (β) normalisation.

4.4.1 Alpha (α) normalisation

α normalisation is applied to the fitnesses before every selection process, it is intended to give weaker individuals a fighting chance to breed by balancing the weights of fitness with equal probabilities.

$$fitness = (\alpha \times equal_probabilities) + (1 - \alpha) \times fitness$$

The code snippet of α normalisation is given below:

```
alpha_normalisation = numpy.zeros(population_fitness.shape, dtype=float)
for specialisation in range(nSpecialisations):
    specialisation_population = numpy.where(populationSpecialisation == specialisation)[0]
    alpha_normalisation[specialisation_population, specialisation] += 1 /
        len(specialisation_population)
population_fitness = (alpha * alpha_normalisation) + (1 - alpha) * population_fitness
```

	0	1	2	3	4	5
0	0.0	0.0	0.0	0.0	0.0	0.0212917...
1	0.0	0.0	0.0	0.0	0.1012266...	0.0
2	0.0	0.0	0.0613894...	0.0	0.0	0.0
3	0.0	0.0	0.0	4.0699001...	0.0	0.0
4	0.0	0.0	0.0	0.0	0.1168629...	0.0
5	0.0	0.1695224...	0.0	0.0	0.0	0.0
6	0.0	0.0	0.0	3.6266675...	0.0	0.0
7	0.0	0.0919719...	0.0	0.0	0.0	0.0
8	0.0689112...	0.0	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0	0.1123887...	0.0
10	0.0	0.0	0.0	0.0	0.0	0.0123710...
11	0.0	0.0	0.0	0.0	0.0008799...	0.0
12	0.0	0.0	0.0	0.0004377...	0.0	0.0
13	0.0	0.0	0.0	0.0	0.0	0.0090691...
14	0.0660982...	0.0	0.0	0.0	0.0	0.0
15	0.0	0.0	0.0	0.0	0.0001035...	0.0
16	0.0	0.0	0.0	0.0	0.0	0.1682625...
17	0.0	0.2976305...	0.0	0.0	0.0	0.0
18	0.0	0.0	0.0004249...	0.0	0.0	0.0
19	0.0	0.0	0.0550913...	0.0	0.0	0.0
20	0.0	0.0	0.0	0.0001234...	0.0	0.0
21	0.0	0.0	0.0	0.0	0.0	0.0255844...
22	0.0	0.0	0.0	0.0	0.0009915...	0.0
23	0.0	0.0	0.0	0.0	0.0771064...	0.0
24	0.0	0.0	0.0	3.9199751...	0.0	0.0

Figure 4.4-1: Population's fitness before α normalisation

	0	1	2	3	4	5
0	0.0	0.0	0.0	0.0	0.0	0.0356458...
1	0.0	0.0	0.0	0.0	0.0769290...	0.0
2	0.0	0.0	0.0664090...	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0416870...	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0847472...	0.0
5	0.0	0.1110770...	0.0	0.0	0.0	0.0
6	0.0	0.0	0.0	0.0416848...	0.0	0.0
7	0.0	0.0723017...	0.0	0.0	0.0	0.0
8	0.0657056...	0.0	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0	0.0825101...	0.0
10	0.0	0.0	0.0	0.0	0.0	0.0311855...
11	0.0	0.0	0.0	0.0	0.0267557...	0.0
12	0.0	0.0	0.0	0.0418855...	0.0	0.0
13	0.0	0.0	0.0	0.0	0.0	0.0295345...
14	0.0642991...	0.0	0.0	0.0	0.0	0.0
15	0.0	0.0	0.0	0.0	0.0263675...	0.0
16	0.0	0.0	0.0	0.0	0.0	0.1091312...
17	0.0	0.1751310...	0.0	0.0	0.0	0.0
18	0.0	0.0	0.0359267...	0.0	0.0	0.0
19	0.0	0.0	0.0632599...	0.0	0.0	0.0
20	0.0	0.0	0.0	0.0417283...	0.0	0.0
21	0.0	0.0	0.0	0.0	0.0	0.0377922...
22	0.0	0.0	0.0	0.0	0.0268115...	0.0
23	0.0	0.0	0.0	0.0	0.0648690...	0.0
24	0.0	0.0	0.0	0.0416862...	0.0	0.0

Figure 4.4-2: Population's fitness after α normalisation

4.4.2 Beta (β) normalisation

β normalisation is applied when offspring are choosing their specialisation, it is intended to give offspring a chance to switch specialisation. Without β normalisation, offspring have equal probabilities to inherit specialisations from their parents.

Probability	Description
$\frac{1}{2}$	Child inherits specialisation of Parent A.
$\frac{1}{2}$	Child inherits specialisation of Parent B.

Table 4.4-1: Probabilities of specialisation inheritance

A small probability, β , is defined where β is the probability that a child specialises in a different specialisation from its parents. $\left(\frac{1-\beta}{2}\right)$ probability of a child inhering parent A's specialisation, and another $\left(\frac{1-\beta}{2}\right)$ probability of inhering parent B's specialisation.

Probability	Description
β	Child specialises in a random specialisation.
$\left(\frac{1-\beta}{2}\right)$	Child inherits specialisation of Parent A.
$\left(\frac{1-\beta}{2}\right)$	Child inherits specialisation of Parent B.

Table 4.4-2: Probabilities of heuristic beta normalisation

The code snippet of beta β normalisation is given below:

```
a_specialisation, b_specialisation = numpy.random.choice(
    numpy.array([random.randint(0, nSpecialisations - 1), a_specialisation, b_specialisation]), 2,
    True, p=[ $\beta$ , (1 -  $\beta$ )/2, (1 -  $\beta$ )/2]
)
```

Table 4.4-3: Code snippet of β normalisation

4.5 Termination

It is nontrivial to determine when to terminate the Genetic Algorithm (GA), the population might face a local maximum and may take many more generations of crossover and mutation to escape. Typically, the indication for termination is when a population is not improving after a certain amount of generations; fitness plateauing.

Severely limited by time for this project, the GA will simply brute-force and run for 1,000 generations of iteration. Notwithstanding, we can still say with proliferating confidence that a population has already peaked if the population still does not improve after k generations, where k is the last generation showing substantial improvement.

Chapter 5 Experiment & Results

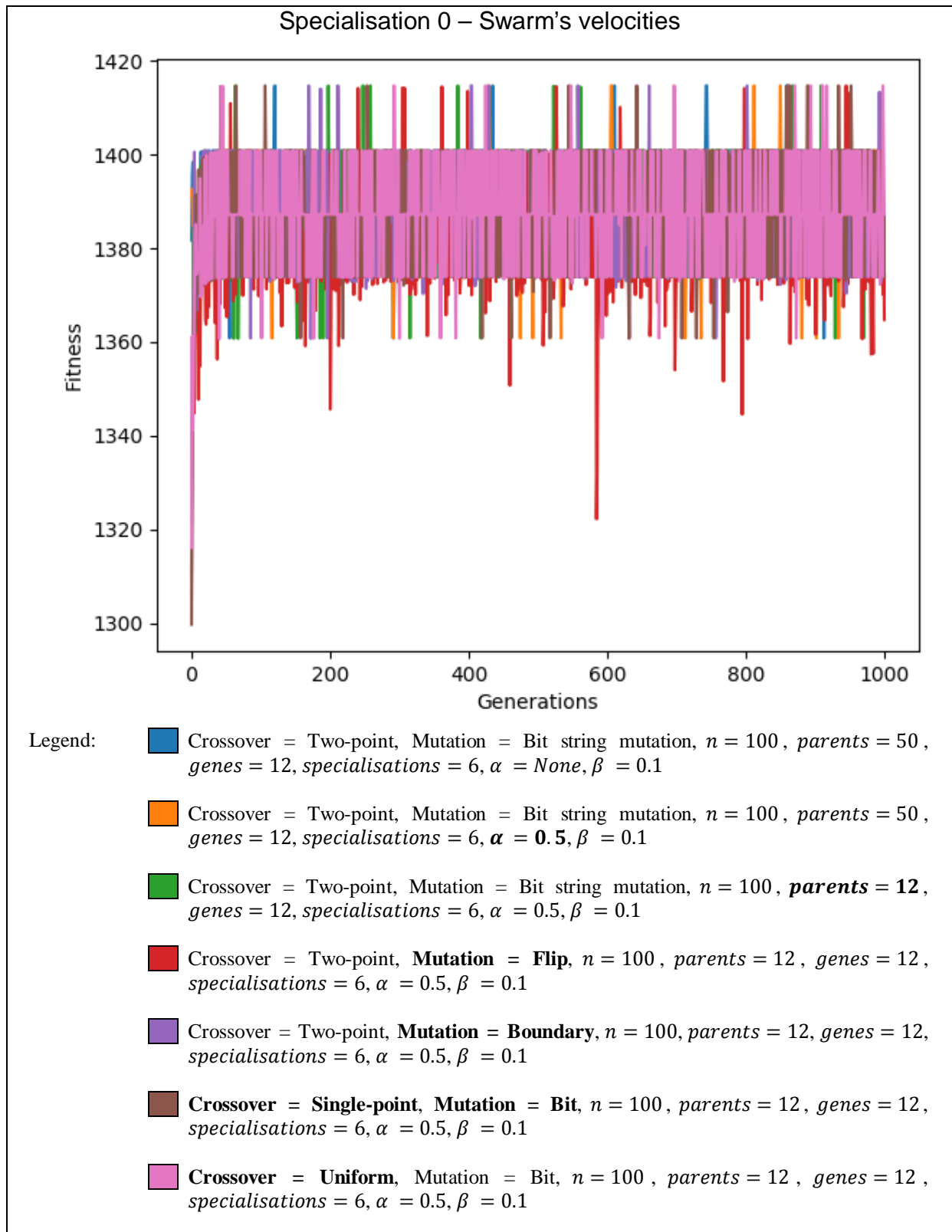
Initial experiments were conducted on a Windows 7 Ultimate 64-bit machine with ASUS Sabertooth Z77 motherboard, Intel® Core™ i7-3770K processor (overclocked to 4.2GHz), 16 GB of Corsair Vengeance (4GB × 4) 1600MHz RAM, Corsair Force Series™ GT 480GB SATA 3 SSHD/SSD, and a NVIDIA GTX 1080 Ti GPU (overclocked to 2.1GHz) of the ASUS ROG STRIX GTX 1080 Ti 11GB OC variant.

However, iterating through many generations is too cumbersome for the aforementioned setup. Hence, subsequent runs were conducted on an Ubuntu 18.04-1LTS Supercomputer with 352 cores (16 x Intel® Xeon® Processor E7-8880 v4), and 12 TB of system memory graciously provided by the School of Computer Science and Engineering, Nanyang Technological University.

A nominal room for error should be taken into consideration due to uncontrollable environment(s), unpredictable event(s), other running program(s)/workload(s), and unmanageable memory management, etc.

5.1 Tuning results of Genetic Algorithm (GA)

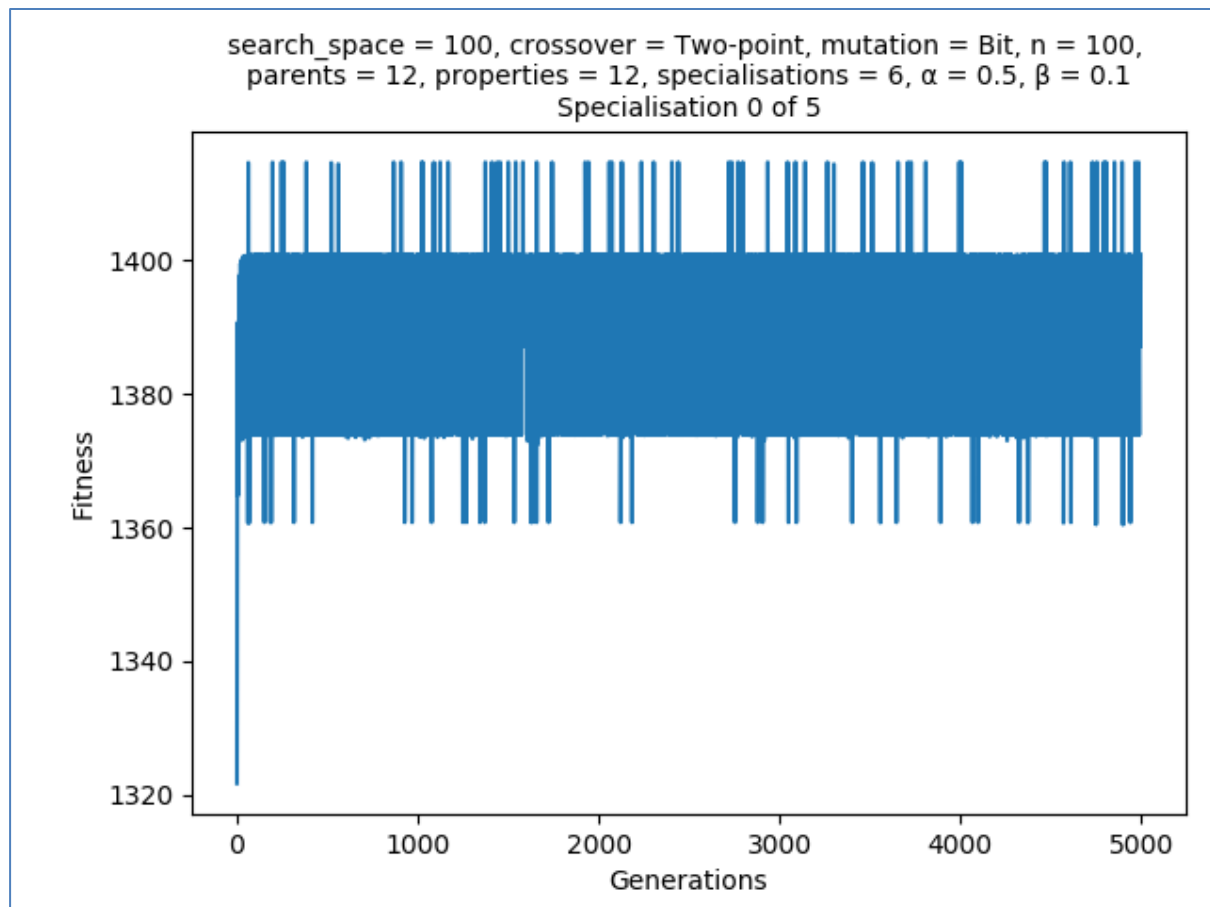
The average fitness of the individual swarms is plotted against generation for each specialisation in the following graphs. The hyper-parameters used by the GA are also stated in the legend.



Graph 5.1-1: Average fitness over generation for specialisation 0 – Swarm's velocities

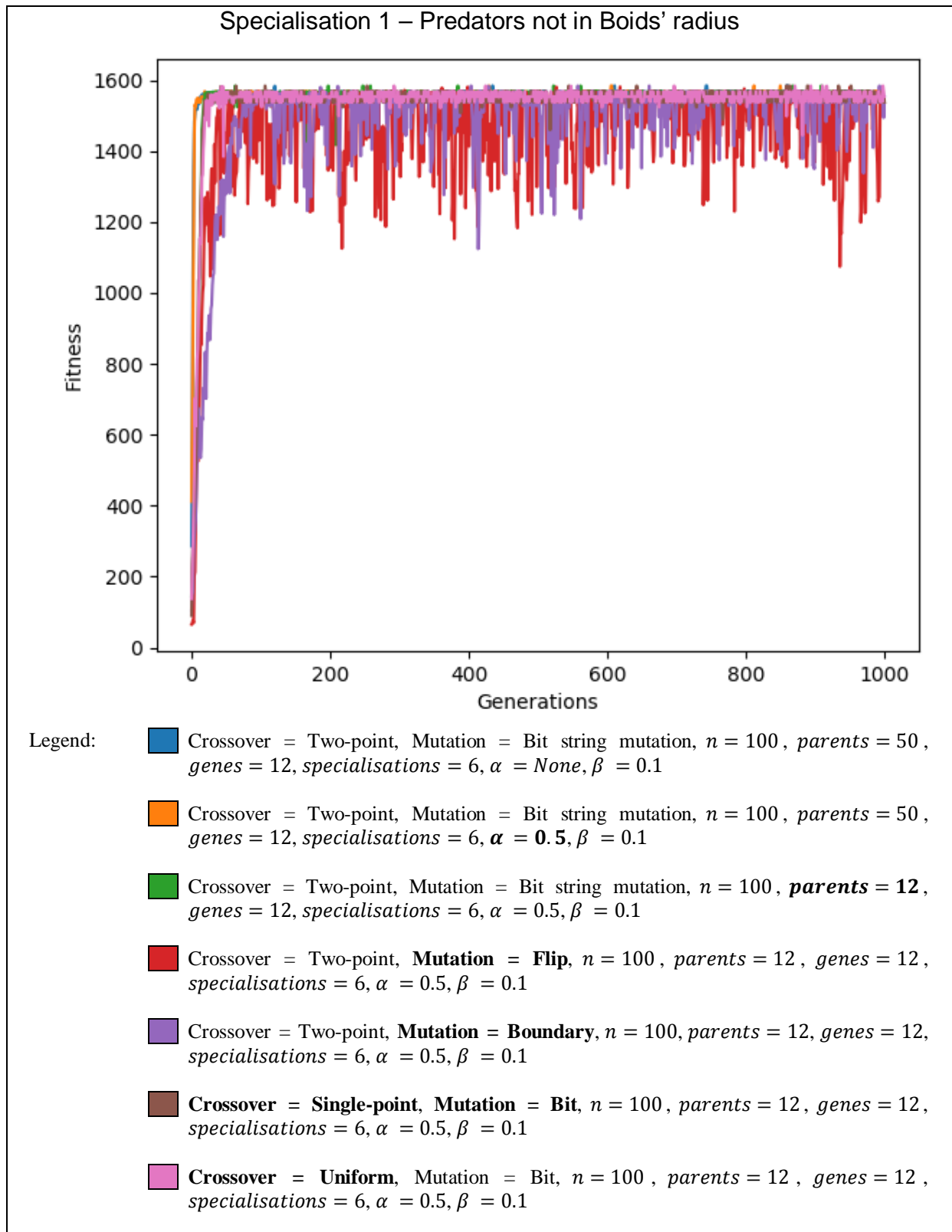
Specialisation 0, or the swarm's velocities, reaches to maxima almost instantly. The idea was to tune the swarm in staying a fast moving swarm, maintaining a relatively high velocity that

is not bounded to the maximumSeed parameter. The “velocities” seems to have reached global maxima, showing no improvement(s) even after 1000 generations or 5000 generations.



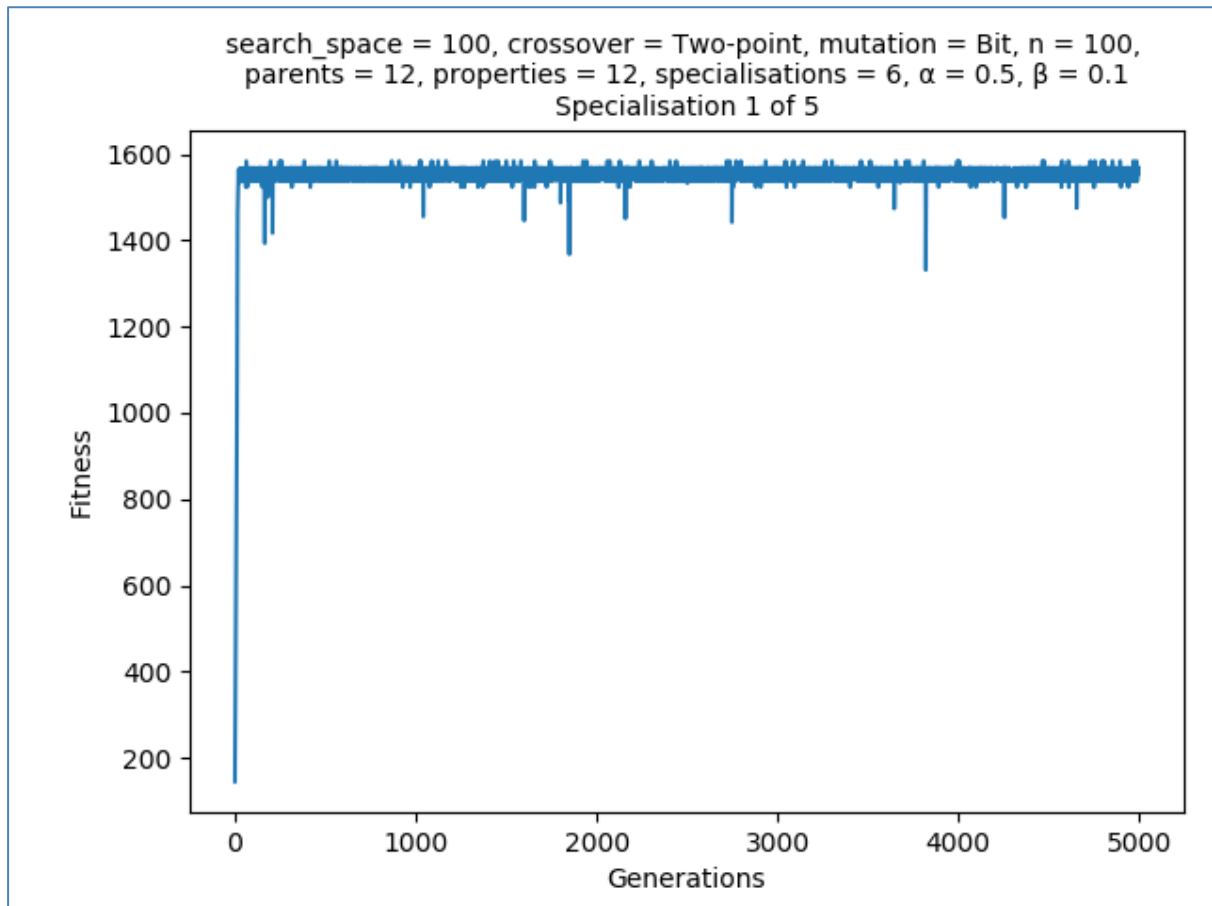
Graph 5.1-2: Average fitness over 5000 generations for specialisation 0 – Swarm's velocities

This could indicate that this measurement is too simplistic for optimisations.



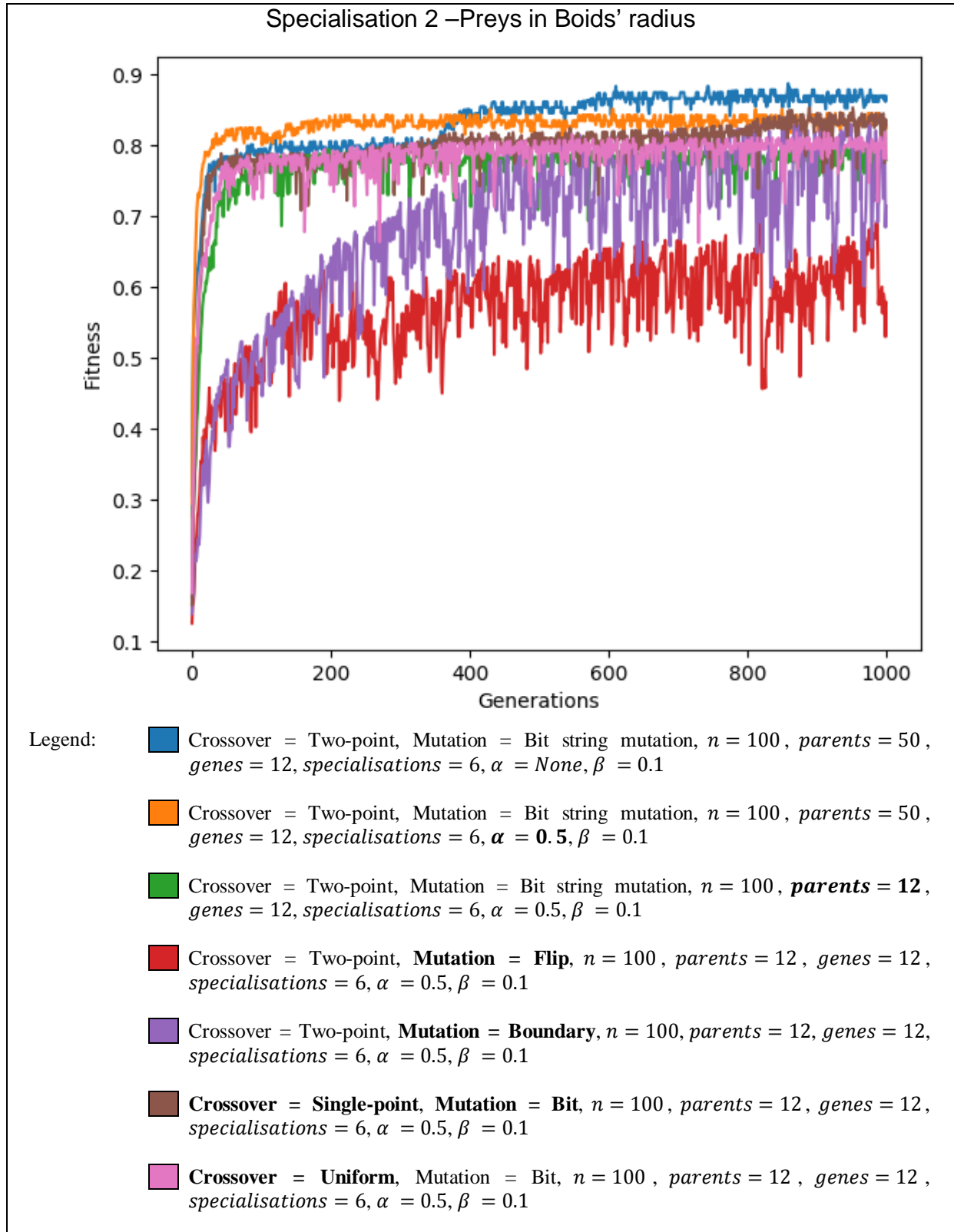
Graph 5.1-3: Average fitness over generation for specialisation 1 – Predators not in Boids' radius

Specialisation 1, or predators not in Boids' radius, also seems to reach to maxima. The idea was to tune the swarm in having fewer predators within the Boids' radius, and is used as an analog to DBSCAN. The “predators” seems to have reached global maxima, showing no improvement(s) even after 1000 generations or 5000 generations.



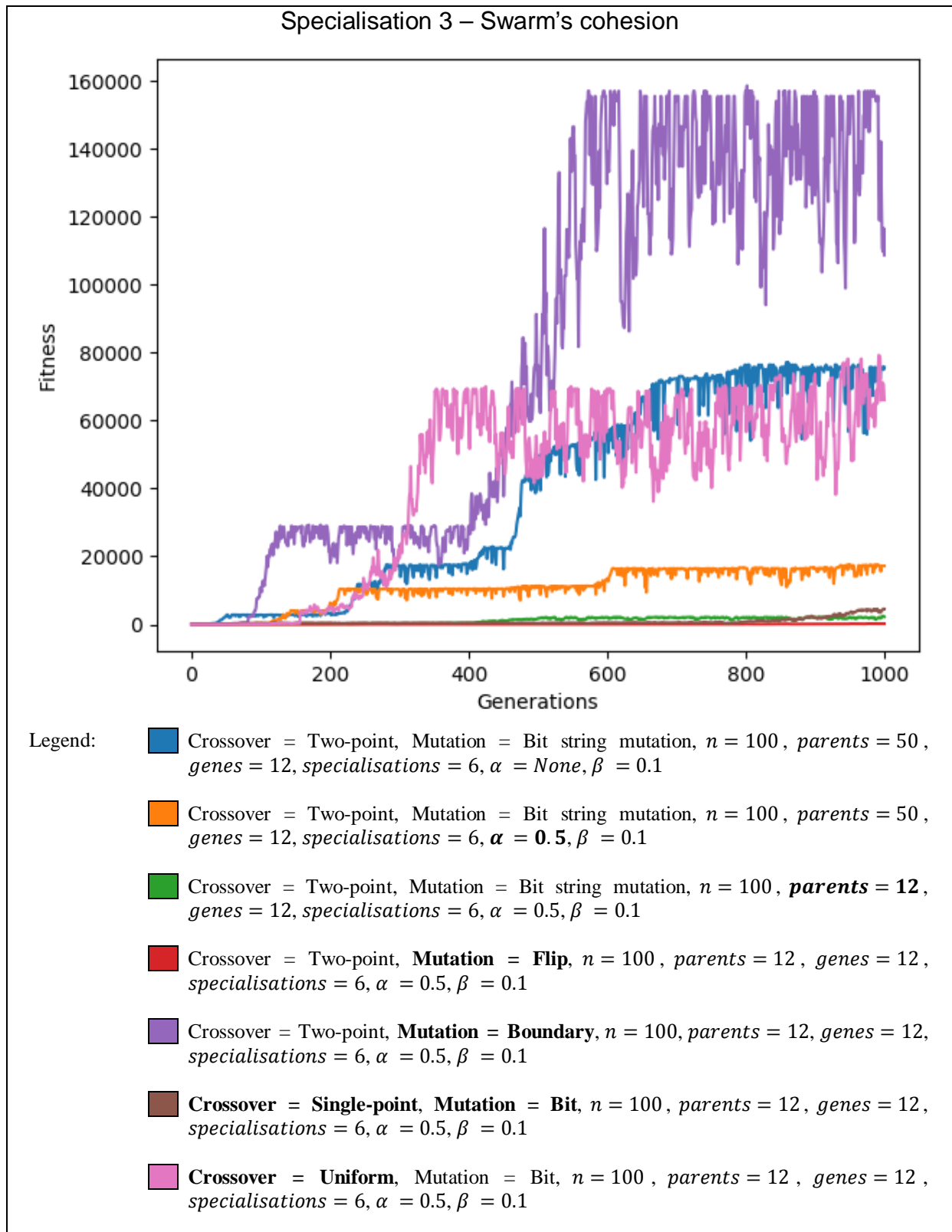
Graph 5.1-4: Average fitness over 5000 generations for specialisation 1 – Predators not in Boids' radius

This could indicate that either this measurement is too simplistic for optimisations or the hunting behaviour for predators needs to be tuned.



Graph 5.1-5: Average fitness over generation for specialisation 2 – Preys in Boids' radius

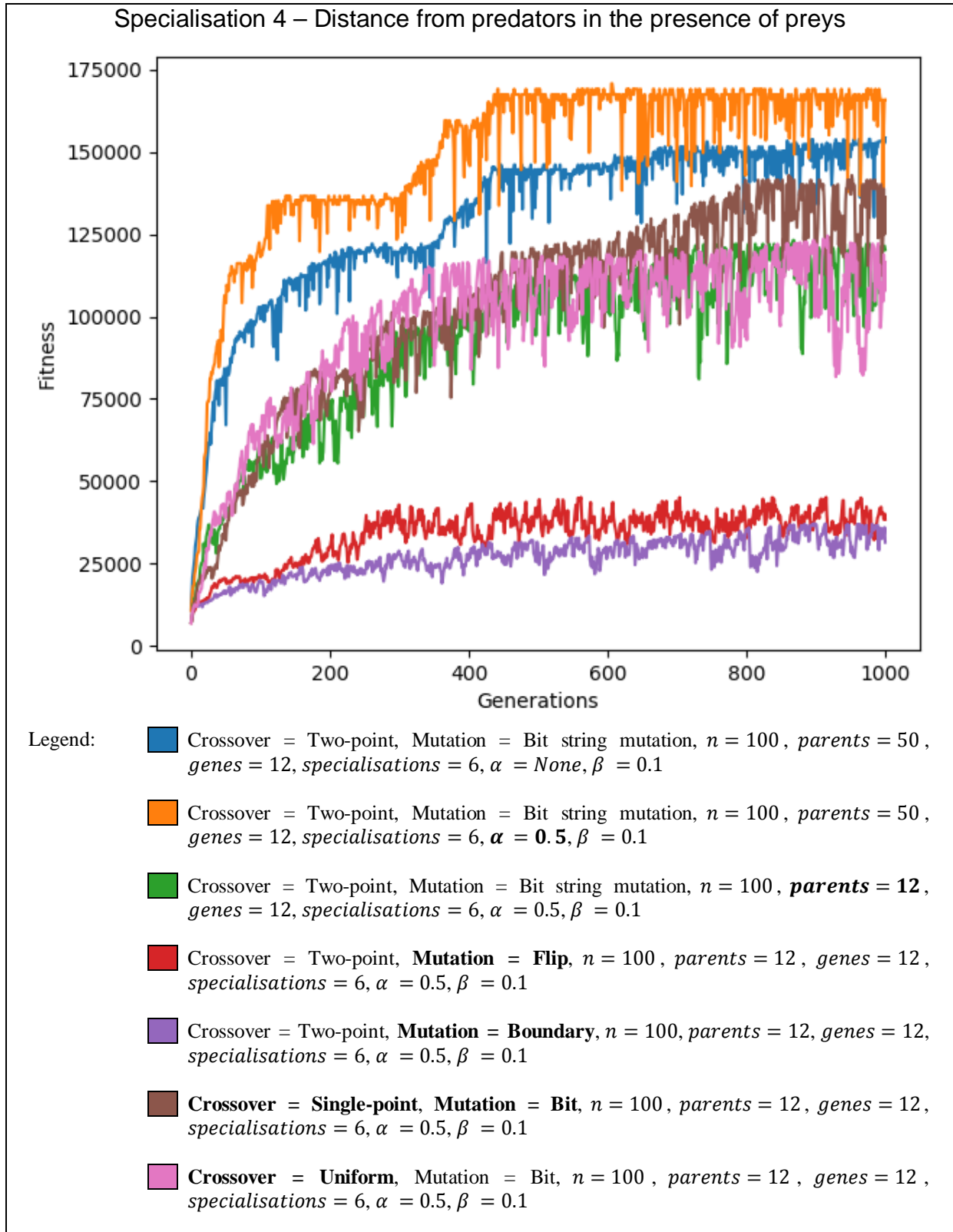
Specialisation 2, or preys in Boids' radius, shows some improvement. The idea is to tune the swarm in having more preys within the Boids' radius, and is used as an analog to DBSCAN. A lower learning rate was observed for Flip and Boundary mutation, Flip mutation being worse. The other hyper-parameters seem to yield similar results.



Graph 5.1-6: Average fitness over generation for specialisation 3 – Swarm's cohesion

Specialisation 3, or swarm's cohesion, shows varying improvements. The idea is to tune the swarm in maintaining cohesion despite the presence of predators. The hyper-parameter showing the best improvements is with Boundary mutation.

Uniform and Two-point crossover performs similarly and is half as good as using Boundary mutation. The worst hyper-parameters showing little to no improvements are: 12 parents, Flip mutation, and Single-point crossover and Bit mutation, they should not be used to tune this specialisation.



Graph 5.1-7: Average fitness over generation for specialisation 4 – Distance from predators in the presence of preys

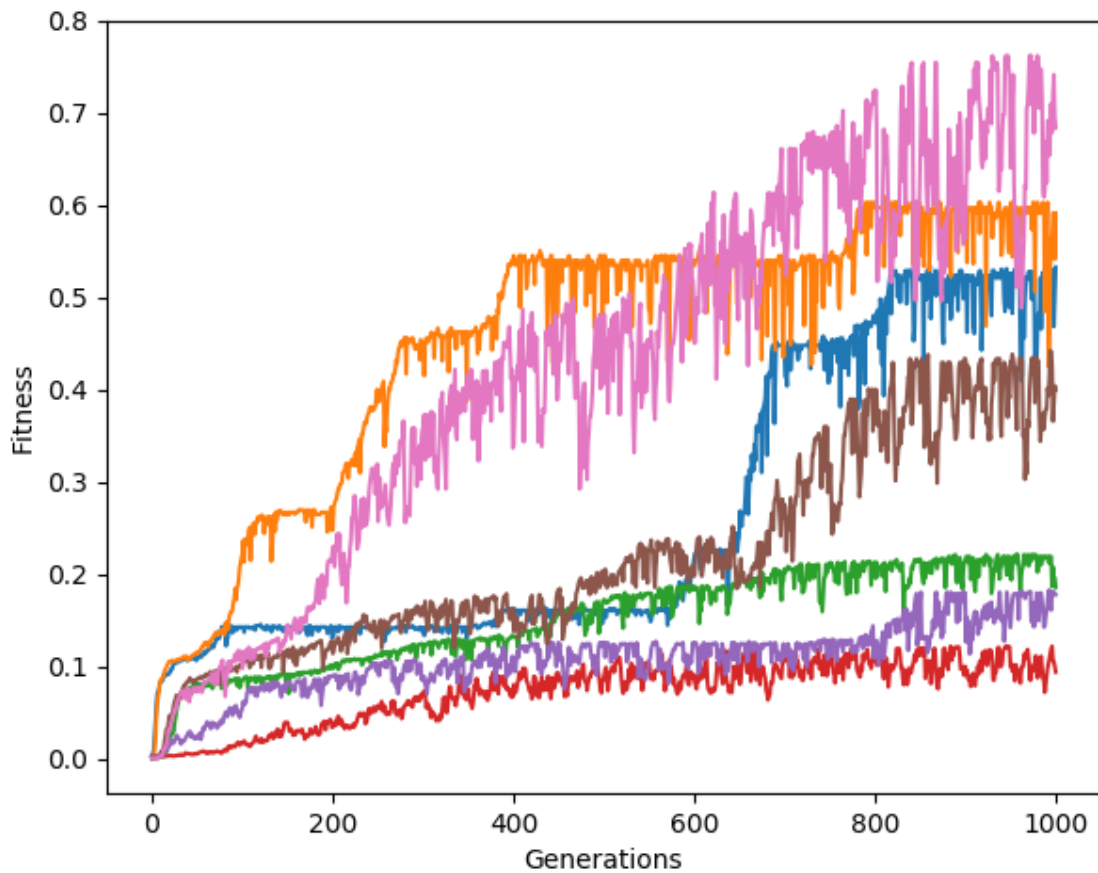
Specialisation 4, or distance from predators in the presence of preys, shows varying but clear distinction on the performance of each hyper-parameter.

The idea is to tune the swarm in staying further from predators in the presence of preys.

The list of hyper-parameters used, in order of best performers to the worse is as follows:

1. Crossover = Two-point, Mutation = Bit string mutation, $n = 100$, $parents = 50$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$
2. Crossover = Two-point, Mutation = Bit string mutation, $n = 100$, $parents = 50$, $genes = 12$, $specialisations = 6$, $\alpha = \text{None}$, $\beta = 0.1$
3. Crossover = **Single-point**, Mutation = **Bit string mutation**, $n = 100$, $parents = 50$, $genes = 12$, $specialisations = 6$, $\alpha = \text{None}$, $\beta = 0.1$
4. **Crossover = Uniform**, Mutation = Bit, $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$
5. Crossover = Two-point, Mutation = Bit string mutation, $n = 100$, **$parents = 12$** , $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$
6. Crossover = Two-point, **Mutation = Flip**, $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$
7. Crossover = Two-point, **Mutation = Boundary**, $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$

Specialisation 5 – Inversed distance from preys in the presence of predators



Legend:

- Crossover = Two-point, Mutation = Bit string mutation, $n = 100$, $parents = 50$, $genes = 12$, $specialisations = 6$, $\alpha = None$, $\beta = 0.1$
- Crossover = Two-point, Mutation = Bit string mutation, $n = 100$, $parents = 50$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$
- Crossover = Two-point, Mutation = Bit string mutation, $n = 100$, **$parents = 12$** , $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$
- Crossover = Two-point, **Mutation = Flip**, $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$
- Crossover = Two-point, **Mutation = Boundary**, $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$
- Crossover = Single-point**, **Mutation = Bit**, $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$
- Crossover = Uniform**, Mutation = Bit, $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$

Graph 5.1-8: Average fitness over generation for specialisation 5 – Inversed distance from preys in the presence of predators

Specialisation 5, or inversed distance from preys in the presence of predators, shows varying performance of each hyper-parameter. The idea is to tune the swarm in staying closer to preys in the presence of predators. Uniform crossover appears to perform better in the long run, outperforming α normalisation. This could mean that the parameters of the swarm (gene) are independent of one another as there is no bias of the proximity of genes for Uniform crossovers compared to Single-point or Two-point crossovers.

5.2 Cross-specialisation results

Aside from each specialisation, additional investigations were done on cross-specialisations. This evaluates how well the Genetic Algorithm (GA) is doing to optimise the specialisations as a whole, i.e. evaluation for a fully optimised swarm, good performance throughout all specialisations. The purpose is to observe if the overall performance of all specialisations is improved by breeding individuals performing well in their own specialisation.

The population's state (population's genomes) is saved every 200 generations of iteration.

Generations
	0	200	400	600	800
					1000

Figure 5.2-1: Slices of generations for cross-specialisation

Each individual in population is then evaluated again but on all specialisations.

Previously, for singular-specialisation matrix, an individual receives a fitness of 0 for other specialisations that they are not specialising in (as shown below). Rows represent individuals and columns represent fitness.

	0	1	2	3	4	5
0	0.0	0.0	0.0	0.0	0.0	0.0356458...
1	0.0	0.0	0.0	0.0	0.0769290...	0.0
2	0.0	0.0	0.0664090...	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0416870...	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0847472...	0.0
5	0.0	0.1110770...	0.0	0.0	0.0	0.0
6	0.0	0.0	0.0	0.0416848...	0.0	0.0
7	0.0	0.0723017...	0.0	0.0	0.0	0.0
8	0.0657056...	0.0	0.0	0.0	0.0	0.0
9	0.0	0.0	0.0	0.0	0.0825101...	0.0
10	0.0	0.0	0.0	0.0	0.0	0.0311855...
11	0.0	0.0	0.0	0.0	0.0267557...	0.0
12	0.0	0.0	0.0	0.0418855...	0.0	0.0
13	0.0	0.0	0.0	0.0	0.0	0.0295345...
14	0.0642991...	0.0	0.0	0.0	0.0	0.0
15	0.0	0.0	0.0	0.0	0.0263675...	0.0
16	0.0	0.0	0.0	0.0	0.0	0.1091312...
17	0.0	0.1751310...	0.0	0.0	0.0	0.0
18	0.0	0.0	0.0359267...	0.0	0.0	0.0
19	0.0	0.0	0.0632599...	0.0	0.0	0.0
20	0.0	0.0	0.0	0.0417283...	0.0	0.0
21	0.0	0.0	0.0	0.0	0.0	0.0377922...
22	0.0	0.0	0.0	0.0	0.0268115...	0.0
23	0.0	0.0	0.0	0.0	0.0648690...	0.0
24	0.0	0.0	0.0	0.0416862...	0.0	0.0

Table 5.2-1: Matrix of singular-specialisation fitness

With cross-specialisation, all of the rows and columns are used to represent overall performance.

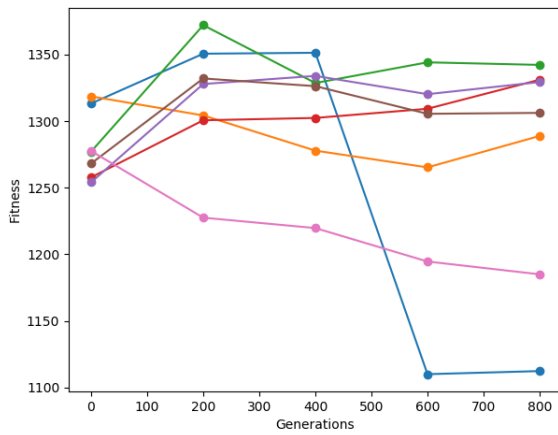
$$f_{(i,j)} = \text{fitness of population}_i \text{ for specialisation}_j$$

	Specialisation					
Population	0	1	2	3	4	5
0	$f_{(0,0)}$	$f_{(0,1)}$	$f_{(0,2)}$	$f_{(0,3)}$	$f_{(0,4)}$	$f_{(0,5)}$
1	$f_{(1,0)}$	$f_{(1,1)}$	$f_{(1,2)}$	$f_{(1,3)}$	$f_{(1,4)}$	$f_{(1,5)}$
...	...					
n	$f_{(n,0)}$	$f_{(n,1)}$	$f_{(n,2)}$	$f_{(n,3)}$	$f_{(n,4)}$	$f_{(n,5)}$

Table 5.2-2: Matrix of cross-specialisation fitness

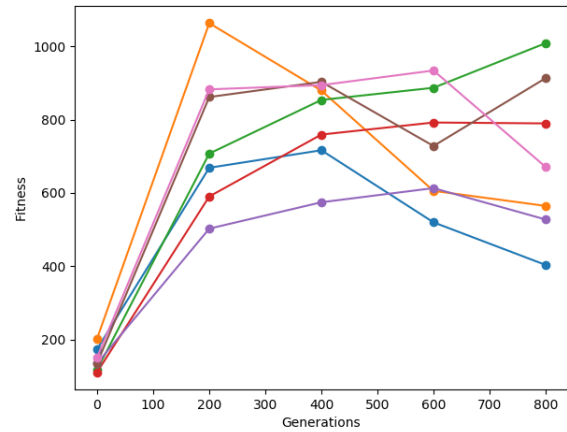
The fitness of each specialisation then is averaged over all individuals in the population at the 200 generations of iteration slices. The results of the investigations are shown below.

Cross-specialisation 0 – Swarm's velocities



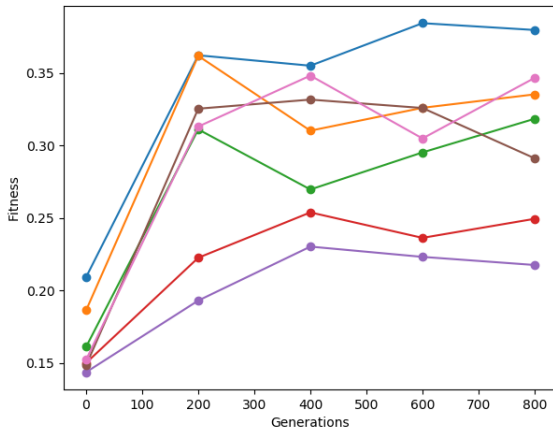
Graph 5.2-1: Cross-specialisation for specialisation 0 – Swarm's velocities

Cross-specialisation 1 – Predators not in Boids' radius



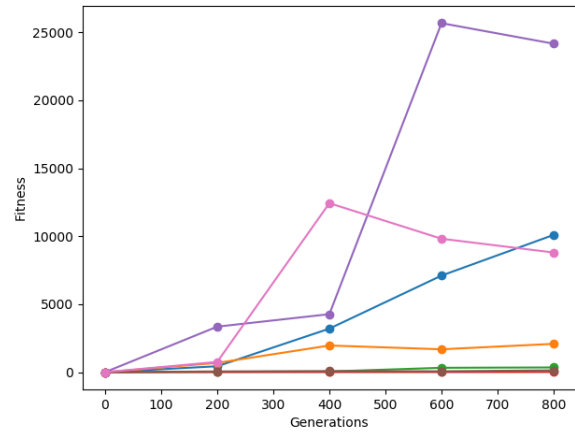
Graph 5.2-2: Cross-specialisation for specialisation 1 – Predators not in Boids' radius

Cross-specialisation 2 –Preys in Boids' radius



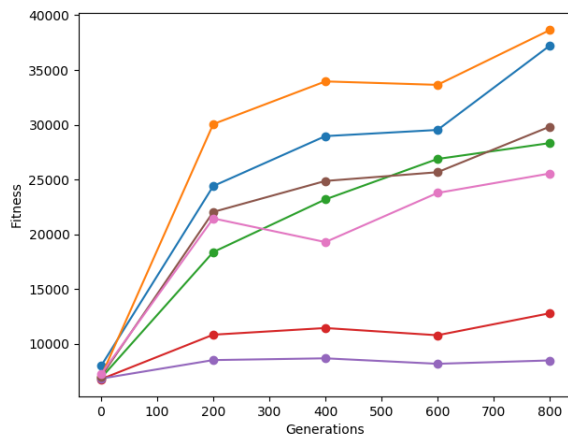
Graph 5.2-3: Cross-specialisation for specialisation 2 – Preys in Boids' radius

Cross-specialisation 3 – Swarm's cohesion



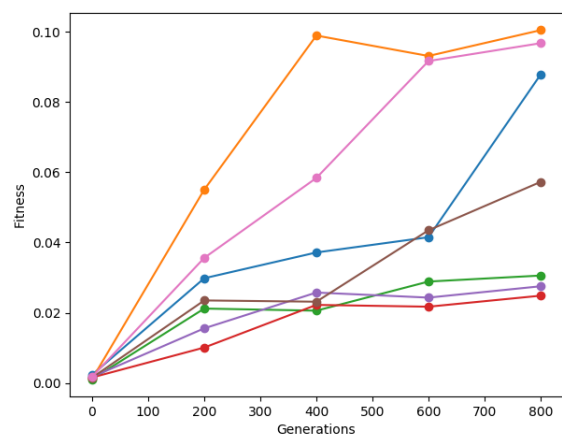
Graph 5.2-4: Cross-specialisation for specialisation 3 – Swarm's cohesion

Cross-specialisation 4 – Distance from predators in the presence of preys



Graph 5.2-5: Cross-specialisation for specialisation 4 – Distance from predators in the presence of preys

Cross-specialisation 5 – Inversed distance from preys in the presence of predators



Graph 5.2-6: Cross-specialisation for specialisation 5 – Inversed distance from preys in the presence of predators

Legend: Crossover = Two-point, Mutation = Bit string mutation, $n = 100$, $parents = 50$, $genes = 12$, $specialisations = 6$, $\alpha = None$, $\beta = 0.1$







	Crossover = Two-point, Mutation = Bit string mutation, $n = 100$, $parents = 50$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$
	Crossover = Two-point, Mutation = Bit string mutation, $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$
	Crossover = Two-point, Mutation = Flip , $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$
	Crossover = Two-point, Mutation = Boundary , $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$
	Crossover = Single-point , Mutation = Bit , $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$
	Crossover = Uniform , Mutation = Bit, $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$

Table 5.2-3: Average fitness of cross-specialisations

Investigations show evidence of improvement across the board, throughout all specialisations, with the exception of specialisation 0. Cross-specialisation evaluation for specialisation 0 appears to reach maxima almost instantaneously. This correlates with the results obtained when evaluating specialisation 0 separately, this supports the indication that the measurement of swarms' velocities is too simplistic for GA optimisations.

The hyper-parameter that produces the better overall performance is:

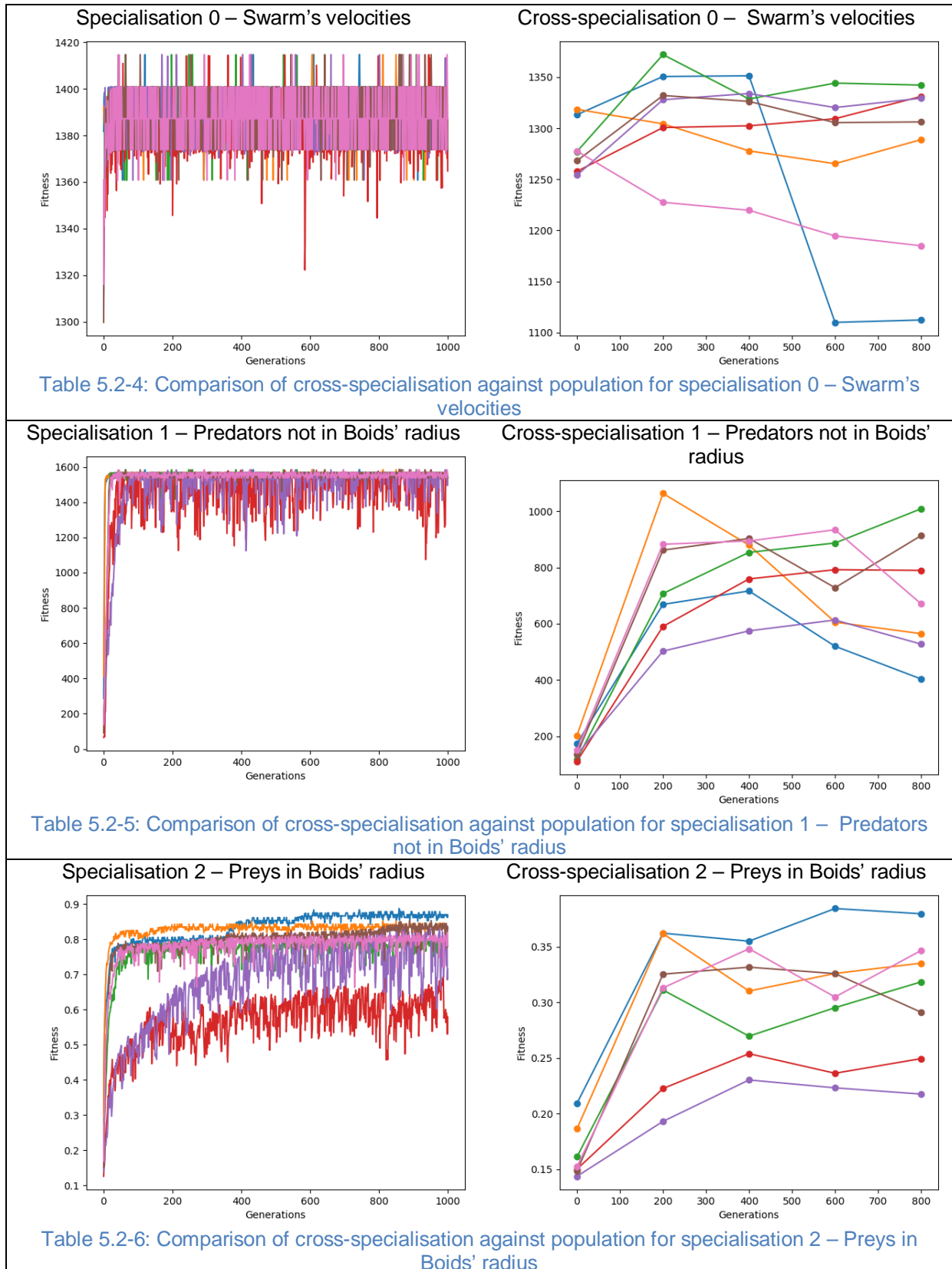
- **Crossover = Uniform**, Mutation = Bit, $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6$, $\alpha = 0.5$, $\beta = 0.1$

As the positions of genes (parameters of the swarm) are arbitrary, Uniform crossover could be performing better because there is no bias of the proximity of genes; Uniform crossover treats parameters of the swarm as independent.

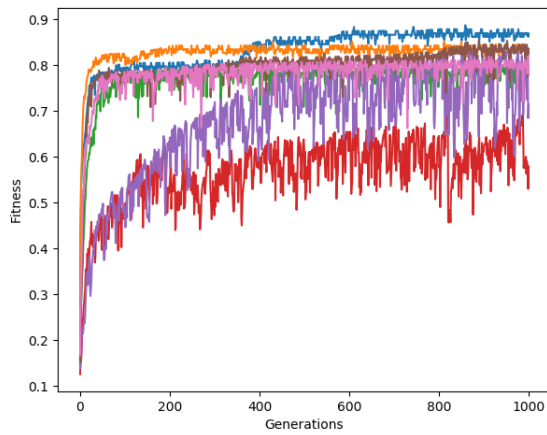
It should be noted that these results alone are not absolute, as they are dependent on randomness. Many runs using the same hyper-parameters should be averaged together to achieve a better gauge, this was not done however due to restrictions of time.

5.2.1 Comparison against population

The cross-specialisations are compared against the separated specialisations on the population.



Specialisation 3 – Swarm's cohesion



Cross-specialisation 3 – Swarm's cohesion

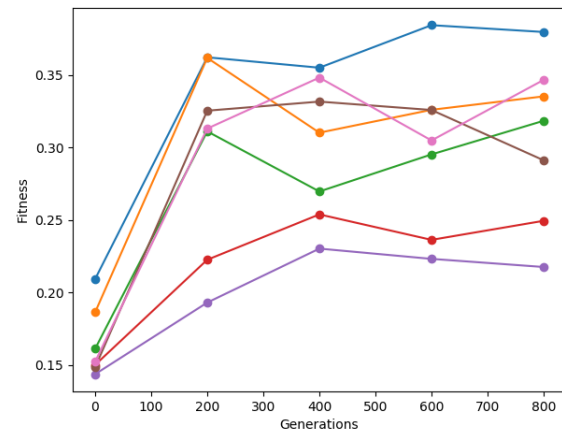
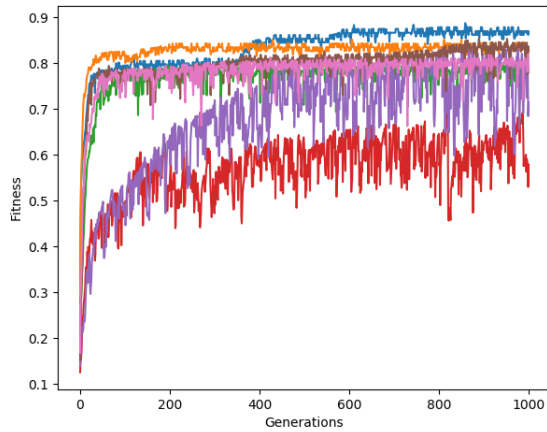


Table 5.2-7: Comparison of cross-specialisation against population for specialisation 3 – Swarm's cohesion

Specialisation 4 – Distance from predators in the presence of preys



Cross-specialisation 4 – Distance from predators in the presence of preys

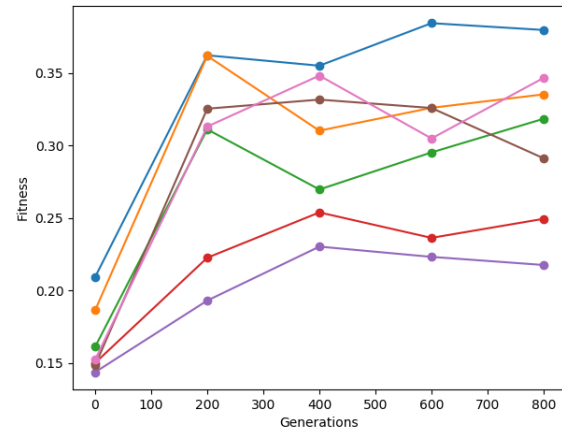
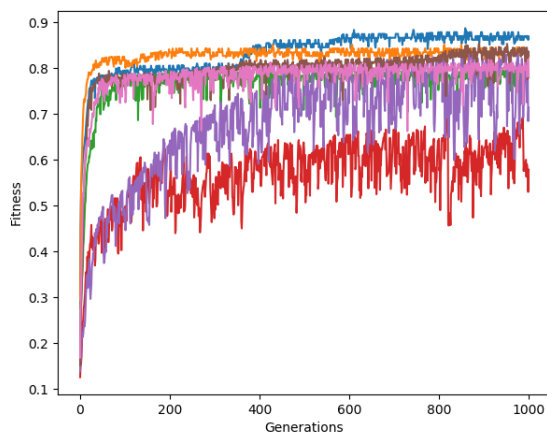


Table 5.2-8: Comparison of cross-specialisation against population for specialisation 4 – Distance from predators in the presence of preys

Specialisation 5 – Distance from preys in the presence of predators



Cross-specialisation 5 – Distance from preys in the presence of predators

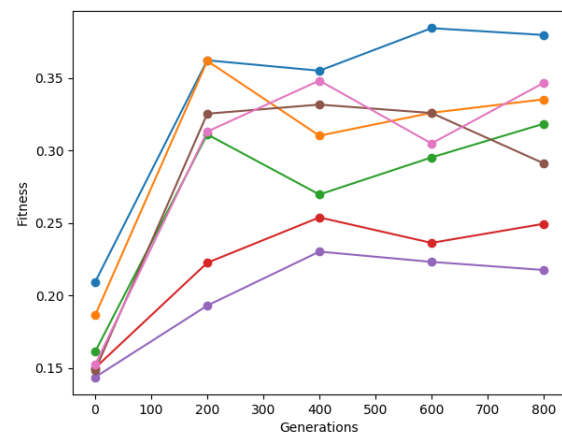


Table 5.2-9: Comparison of cross-specialisation against population for specialisation 5 – Distance from preys in the presence of predators

Legend: ■ Crossover = Two-point, Mutation = Bit string mutation, $n = 100$, $parents = 50$, $genes = 12$, $specialisations = 6$, $\alpha = None$, $\beta = 0.1$

■ Crossover = Two-point, Mutation = Bit string mutation, $n = 100$, $parents = 50$,






—	$genes = 12, specialisations = 6, \alpha = 0.5, \beta = 0.1$
	Crossover = Two-point, Mutation = Bit string mutation, $n = 100$, parents = 12, $genes = 12, specialisations = 6, \alpha = 0.5, \beta = 0.1$
	Crossover = Two-point, Mutation = Flip , $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6, \alpha = 0.5, \beta = 0.1$
	Crossover = Two-point, Mutation = Boundary , $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6, \alpha = 0.5, \beta = 0.1$
	Crossover = Single-point , Mutation = Bit , $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6, \alpha = 0.5, \beta = 0.1$
	Crossover = Uniform , Mutation = Bit, $n = 100$, $parents = 12$, $genes = 12$, $specialisations = 6, \alpha = 0.5, \beta = 0.1$

Table 5.2-10: Comparison of cross-specialisations with population

Observe that the maximum fitness for the cross-specialisation pales in comparison to the populations' specialisations; individuals perform much better in their own specialisation than in the combined specialisations. Also, the improvements of cross-specialisations are minuscule as compared to improvements in individual specialisations.

Although the GA is successful in optimising each specialisation, the GA is only negligibly successful in optimising the overall performance of the swarms, i.e. a swarm that performs well in all specialisation. This could be attributed to conflicts between the steering behaviours, e.g. Separation steering behaviour is counter-intuitive to Cohesion steering behaviour. The stagnant improvement could indicate that the GA has already found a balance between the steering behaviours.

Notwithstanding, it is still possible that with greater iterations, more improvements can be achieved. However due to time restrictions, that was not possible to investigate.

5.3 Swarm observations

Some individuals from the resulting population were provided with the swarm simulation to observe for interesting behaviours.

5.3.1 Cohesion

All the individuals observed were able to exhibit swarm cohesion.

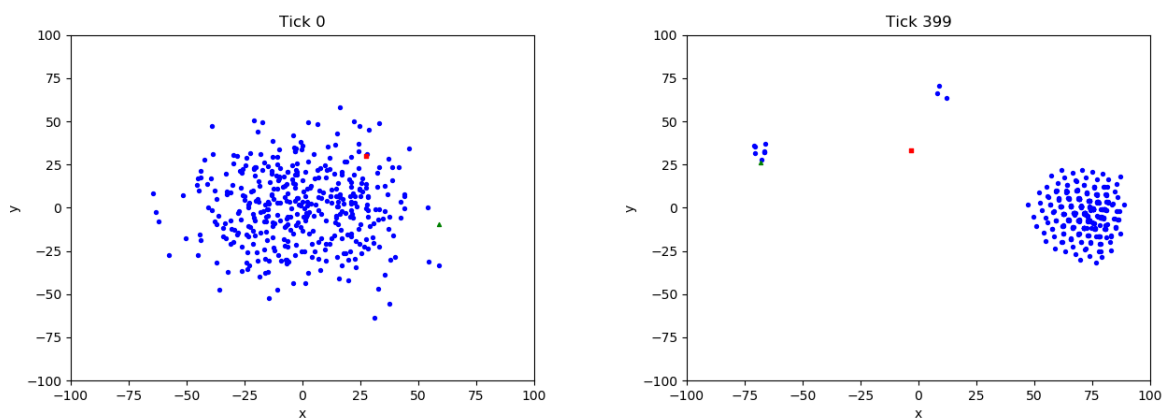


Figure 5.3-1: Swarm cohesion

Starting at random positions (top left), swarms are able to achieve relative swarm cohesion (top right).

5.3.2 Bait Ball

A bait ball is manifested when certain swarms of fishes, threatened by predator(s), naturally ball up together defensively and swarm around the predator(s) in a spherical formation. Some individuals were observed to exhibit bait balls.

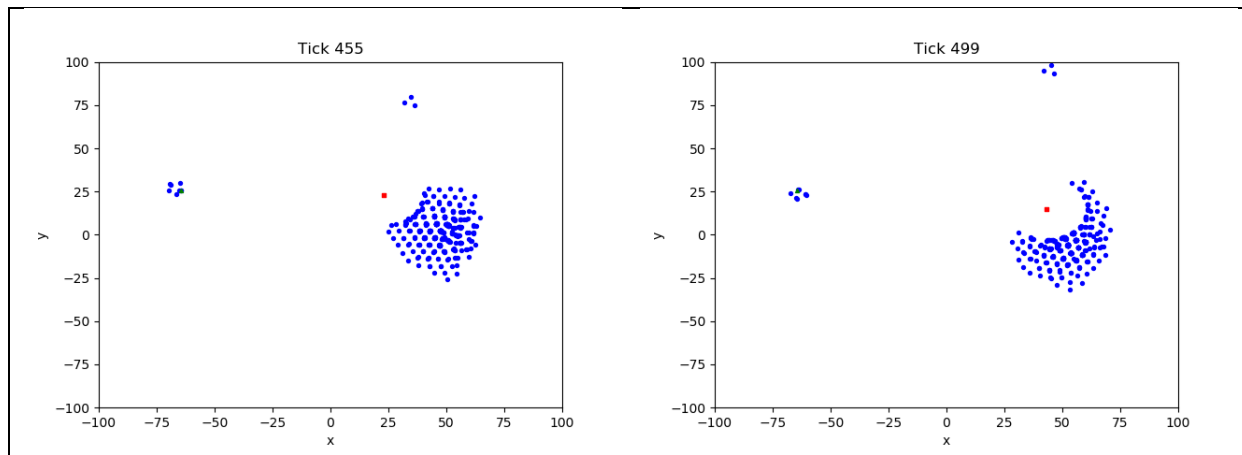


Figure 5.3-2: Swarm predator avoidance

Predator avoidance was first observed in preparation of a bait ball, mimicking the beginnings of a miniature bait ball (top right).

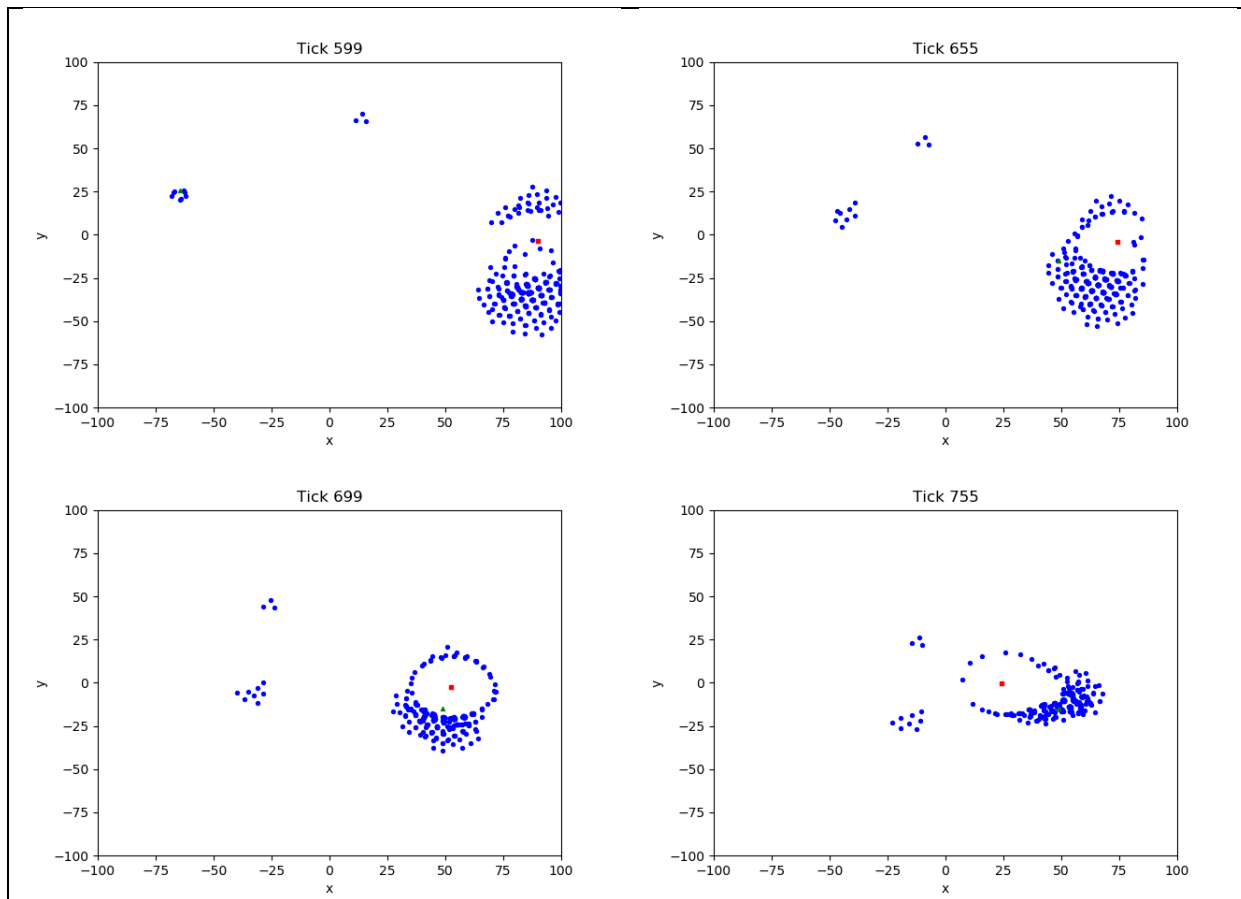


Figure 5.3-3: Swarm bait ball

A predator manages to get into the middle of a swarm due to the swarm being trapped in a corner (top left), predatory steering behaviours kicks in and the swarm starts to avoid the predator (top right). Predator continues to follow its route and goes the next waypoint to the left (bottom left), predator exits the bait ball and swarm exhibits cohesion once again (bottom right).

Chapter 6 Conclusions and future improvements

In conclusion, the Boid Behavioural Model is limited and only marginal at best for exhibiting interesting swarm behaviours. Extending and/or augmenting from the original Boid model is necessary to produce interesting behaviours.

The tuning of multiple objectives for a Genetic Algorithm (GA) is nontrivial and optimal results cannot be expected from simple tunings. By applying the “Multi-Objective Genetic Algorithm” (MOGA), multiple objectives are divided and conquered (optimised) at lower levels to be recombined to achieve overall optimisations. Results show evidence that MOGA is capable of optimising multiple objectives separately, but is negligible in optimising all the objectives together. However, it is still possible for MOGA to optimise the objectives as a whole, given enough generations of iteration; like nature, the advent of human evolution didn’t occur until many generations later.

Certain behaviours emerging from the resulting swarm of MOGA were observed, such as swarm cohesion and bait ball. Therefore, this project is successful in achieving the objectives set out; behaviours can be manufactured with proper measurements/objectives and specialised GA for multiple objectives.

6.1 Recommendations for future work

Throughout the project, several possible improvements were discovered, but were not executed however due to time restrictions. Some recommendations for future work include:

1. Swarm simulation:
 - 1.1. Speed up simulation using faster simulation software vs. Python which uses interpreted code which is relatively slower. E.g. Unity's ECS, etc.
 - 1.2. Optimise code for speedup by exploring additional optimisation(s) available for Python.
 - 1.3. Include additional measurements to allow Genetic Algorithms to tune more parameters with the possibility of achieving better genetic tuning. E.g. providing the full fat DBSCAN measurements, etc.
2. Genetic Algorithm (GA)
 - 2.1. Different GAs could be used to tune the swarm, not necessarily using Multi-Objective Genetic Algorithm. E.g. A different GA meant for optimising multiple objectives could be used.
3. Multi-Objective Genetic Algorithm (MOGA)
 - 3.1. Further tuning of hyper-parameters could result in more interesting behaviours observed. E.g. alleviating local maxima by increasing the probabilities of mutations, testing different crossover types, mutation types, etc.
 - 3.2. Run MOGA for more generations of iteration, this project only ran for 1000 generations. Additional behaviours could emerge from further tuned swarms.

References

- [1] C. G. Langton, *Artificial Life: An Overview*, 1st ed., C. G. Langton, Ed., Cambridge, Massachusetts: The MIT Press, 1997.
- [2] J. Wander and J. Macro, "The Need for and Development of Behaviourally Realistic Agents," *International Workshop on Multi-Agent Systems and Agent-Based Simulation*, pp. 36-49, July 2003.
- [3] C. Reynolds, "Flocks, herds and schools: A distributed behavioral model," *ACM SIGGRAPH Computer Graphics*, vol. XXI, no. 4, pp. 25-34, 1 August 1987.
- [4] C. Reynolds, "Boids (Flocks, Herds, and Schools: a Distributed Behavioral Model)," Reynolds Engineering & Design, 1986. [Online]. Available: www.red3d.com/cwr/boids. [Accessed 2018].
- [5] M. Larsson and S. Lundgren, "Perception of Realistic Flocking Behavior in the Boid Algorithm," Blekinge Institute of Technology, Karlskrona, 2017.
- [6] I. L. Bajec, N. Zimic and M. Mraz, "The computational beauty of flocking: boids revisited," *Mathematical and Computer Modelling of Dynamical Systems*, vol. XIII, no. 4, pp. 331-347, August 2007.
- [7] B. Crowther, "Flocking of autonomous unmanned air vehicles," *The Aeronautical Journal*, vol. CVII, no. 1068, pp. 99-109, February 2003.
- [8] N. R. Watson, N. W. John and W. J. Crowther, "Simulation of unmanned air vehicle flocking," in *Proceedings of Theory and Practice of Computer Graphics, 2003*, Birmingham, 2003.
- [9] C. Hartman and B. Beneš, "Autonomous boids," *Computer Animation and Virtual Worlds (CASA 2006)*, vol. XVII, no. 3-4, pp. 199-206, 14 July 2006.
- [10] A. Shabtay, Z. Rabinovich and J. S. Rosenschein, "Behaviosites: Manipulation of Multiagent System Behavior," *The Twenty-First National Conference on Artificial Intelligence*, pp. 709-715, 16-20 July 2006.
- [11] X. Cui, J. Gao and T. E. Potok, "A flocking based algorithm for document clustering analysis," *Journal of Systems Architecture*, vol. LII, no. 8-9, pp. 505-515, August-September 2006.
- [12] F. Heppner, "Three-dimensional structure and dynamics of bird flocks," *Animal Groups in Three Dimensions: How Species Aggregate*, pp. 68-89, 1997.
- [13] Y. Chen, K. Kobayashi, X. Huang and Z. Nakao, "Genetic algorithms for optimization of boids model," *10th International Conference on Knowledge-Based and Intelligent*

Information and Engineering Systems (KES 2006), pp. 55-62, October 2006.

- [14] Y. Chen, K. Kobayashi, H. Kawabayashi and X. Huang, "Application of Interactive Genetic Algorithms to Boid Model Based Artificial Fish Schools," *12th International Conference on Knowledge-Based and Intelligent Information and Engineering Systems (KES 2008)*, pp. 141-148, September 2008.
- [15] S. Alaliyat, H. Yndestad and F. Sanfilippo, "Optimisation of Boids Swarm Model Based on Genetic Algorithm and Particle Swarm Optimisation Algorithm (Comparative Study)," *28th European Conference on Modelling and Simulation (ECMS 2014)*, pp. 643-650, May 2014.
- [16] Kurzgesagt, "Emergence – How Stupid Things Become Smart Together," Kurzgesagt, 16 November 2017. [Online]. Available: <https://www.youtube.com/watch?v=16W7c0mb-rE>. [Accessed March 2019].
- [17] K. Deb and W. M. Spears, "C6.2: Speciation method," in *Handbook of Evolutionary Computation*, Institute of Physics Publishing, 1997.
- [18] O. M. Shir, "Niching in Evolutionary Algorithms," *Handbook of Natural Computing*, pp. 1035-1069, 2012.
- [19] A. V. Husselmann and K. A. Hawick, "Simulating Species Interactions and Complex Emergence in Multiple Flocks of Boids with GPUs," *The 23rd IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2011)*, pp. 100-107, 2011.

Appendix

Code for Boid Algorithm:

```
//move_all_boids_to_new_positions()
void Boids() {
    position = transform.position;
    rotation = transform.rotation;

    Vector3 cohesion, separation, alignment;
    Vector3 finish;

    separation = Separation(this.gameObject);
    alignment = Alignment(this.gameObject);
    cohesion = Cohesion(this.gameObject);
    finish = Finish(this.gameObject);

    Vector3 target = separation + alignment + cohesion + finish;
    Vector3 targetPosition = position + target;

    transform.position += target.normalized * speed * 0.025f * Time.deltaTime;
    /*transform.Translate(target * Time.deltaTime);
    rigidbody.AddForce(target, ForceMode.Force);
    rigidbody.MovePosition(targetPosition * Time.deltaTime);
    rigidbody.velocity = target * Time.deltaTime;*/

    if (target != Vector3.zero) {
        transform.rotation = Quaternion.Slerp(rotation, Quaternion.LookRotation(targetPosition - position, transform.up), Time.deltaTime * rotationSpeed);
        //transform.rotation = Quaternion.LookRotation(targetPosition - position, transform.up);
    }

    //rigidbody.velocity = transform.forward * speed * Time.deltaTime;
}

//Separation: steer to avoid crowding local flockmates
private Vector3 Separation(GameObject bJ) {
    position = bJ.transform.position;
    Vector3 boidPosition, c = Vector3.zero;

    foreach(GameObject boid in GameController.boidPooler.objectPool) {
        if (boid != bJ) {
            boidPosition = boid.transform.position;
            if (/*Vector3.Distance(boidPosition, position) <= avoidDistance*/ (boidPosition - position).sqrMagnitude <= sqrAvoidDistance) {
                c -= boidPosition - position;
            }
        }
    }

    return c;
}

//Alignment: steer towards the average heading of local flockmates
private Vector3 Alignment(GameObject bJ) {
    position = bJ.transform.position;
    Vector3 pvJ = Vector3.zero;
    int neighbours = 0;

    foreach(GameObject boid in GameController.boidPooler.objectPool) {
        if (boid != bJ) {
            if (/*Vector3.Distance(boid.transform.position, position) <= neighbourDistance*/ (boid.transform.position - position).sqrMagnitude <= sqrNeighbourDistance / 2f) {
                pvJ += boid.GetComponent<Rigidbody>().velocity;
                neighbours++;
            }
        }
    }

    return pvJ / neighbours;
}
```

```

        }
    }
}

return neighbours < 1 ? pvJ : (pvJ / (float)neighbours);
}

//Cohesion: steer to move toward the average position of local flockmates
private Vector3 Cohesion(GameObject bJ) {
    position = bJ.transform.position;
    Vector3 boidPosition, pcJ = Vector3.zero;
    int neighbours = 0;

    foreach(GameObject boid in GameController.boidPooler.objectPool) {
        if (boid != bJ) {
            boidPosition = boid.transform.position;
            if (/*Vector3.Distance(boidPosition, position) <= neighbourDistance*/ (boidPosition -
position).sqrMagnitude <= sqrNeighbourDistance) {
                pcJ += boidPosition;
                neighbours++;
            }
        }
    }

    return neighbours < 1 ? pcJ : (pcJ / (float)neighbours) - position;
}

```

FINAL

This page is intentionally left blank.