

Tunning the Blender Physics Engine via a Genetic Algorithm

A Thesis

Presented to

The Faculty of the

Department of Computer and Information Science

Brooklyn College

The City University of New York

In Partial Fulfillment
of the Requirements for the Degree
Master of Arts

by

David Lettier

April 7, 2014

Acknowledgments

Dr. Elizabeth Sklar's advisement was outstanding and pivotal to the work outlined below. Kudos to the computer science department for providing such an enriching master's program and enlisting exceptionally knowledgeable faculty. Additional kudos to the Blender and Bullet teams for developing a very robust platform for one to build off of.

Abstract

Using a physics engine for robotic simulation provides fidelity with the added benefit of lower research costs, minimal space requirements, and even less time as the physical simulation can be run at faster than real-time speeds [1]. However, physics engines have numerous parameters to tune such as friction coefficients, rigid-body constraints, body mass, gravitational acceleration, material elasticity, collision shapes, etcetera [web][web]. Tuning these either programmatically or as in the case of *Blender*¹—via the graphical user interface—is a daunting task considering the multidimensional state space. Simulation fidelity is compromised as some parameters are perfected while others are now maladjusted. Adjusting some parameters only to malign others leads to a “herding cats” type of effect. This ultimately results in a “reality gap” between the simulated environment and the physical environment. However, by using a genetic algorithm to optimize or rather fine tune the physics-engine parameters, the hypothesis was that the physics engine will produce a robot motion model more closely reassembling that of the real-world robot kinematics. This thesis documents a structured approach to tuning the Blender physics engine via a genetic algorithm thereby closing the reality gap between simulation and reality.

¹Blender, released under the GNU General Public License, is a 3D creation suite allowing artists and programmers alike to produce immersive audio/visual works ranging from animation shorts to real-time 3D interactive games. Features include 3D modeling, 3D sculpting, texturing, sound editing, film editing, motion tracking, rigging, rendering, animation, physical simulation, a real-time 3D engine, and extensibility via the integrated Python API [2].

Contents

1	Introduction	1
1.1	Genetic Algorithms	1
1.2	SimPL	1
1.3	BBAutoTune	2
1.4	BlenderSim	2
2	Genetic Algorithms	3
2.1	Overview	3
2.2	Genomes	4
2.2.1	Genes	4
2.3	Evaluation	5
2.3.1	Criteria	5
2.3.2	Fitness Function	5
2.4	Operators	6
2.4.1	Selection	6
2.4.2	Elitism	6
2.4.3	Crossover	6
2.4.4	Mutation	7
2.5	Population	7
2.5.1	Initialization	7
2.5.2	Reproduction	8
2.6	Termination	8
3	SimPL	9
3.1	Overview	9
3.2	Implementation	9
3.2.1	Arena	9
3.2.2	Ball	10
3.2.3	Paddle	12
3.2.4	Physics Engine	13
3.2.5	Neural Network	13
3.2.6	Genetic Algorithm	15
3.2.7	Database Manager	19
3.3	Platform	19
3.4	Experimental Designs	20
3.4.1	Experiment One	22
3.4.2	Experiment Two	23
3.4.3	Experiment Three	27
3.4.4	Experiment Four	30
3.4.5	Experiment Five	30

3.4.6	Experiment Six	30
3.4.7	Experiment Seven	31
3.5	Experimental Results	32
3.5.1	Experiment One	32
3.5.2	Experiment Two	32
3.5.3	Experiment Three	36
3.5.4	Experiment Four	36
3.5.5	Experiment Five	40
3.5.6	Experiment Six	41
3.5.7	Experiment Seven	43
3.5.8	Comparative Results	44
4	BBAutoTune	51
4.1	Overview	51
4.2	Implementation	51
4.2.1	Surveyor SRV-1 Blackfin 3D Model	51
4.2.2	GUI	52
4.2.3	Physics Engine API	52
4.2.4	Database Manager	54
4.2.5	Robot Monitor	54
4.2.6	Progress Monitor	54
4.2.7	Genetic Algorithm	54
4.3	Platform	67
4.4	Experimental Designs	68
4.4.1	Experiment One	68
4.4.2	Experiment Two	69
4.4.3	Experiment Three	70
4.4.4	Experiment Four	70
4.4.5	Experiment Five	70
4.5	Experimental Results	70
4.5.1	Experiment One	70
4.5.2	Experiment Two	75
4.5.3	Experiment Three	77
4.5.4	Experiment Four	79
4.5.5	Experiment Five	81
4.6	Simulated versus Real Motion	83
5	BlenderSim	87
5.1	Overview	87
5.2	Problems Faced	87
5.3	Implementation	90
5.3.1	Arena	90
5.3.2	Surveyor SRV-1 Blackfin 3D Models	90
5.3.3	Robot Path Planner	90
5.3.4	Robot Controller	90
5.3.5	Task Points Manager	93
5.4	Platform	95
5.5	Experimental Design	96
5.6	Experimental Result	98

6 Conclusion	101
6.1 Genetic Algorithms	101
6.2 SimPL	101
6.3 BBAutoTune	102
6.4 BlenderSim	102
A Software Code	103
A.1 SimPL	103
A.2 BBAutoTune	133
A.3 BlenderSim	159

List of Figures

3.1	SimPL Arena	10
3.2	SimPL Arena Ball	11
3.3	SimPL Angle of Incidence	11
3.4	SimPL Paddle	12
3.5	SimPL NN	14
3.6	SimPL NN Input Vectors	15
3.7	SimPL NN Input Tuple	16
3.8	Basic GA	17
3.9	Simplified Fitness Function	24
3.10	Roulette Wheel Selection	26
3.11	Self-adaptation Algorithm	29
3.12	Experiment One Average Fitness	33
3.13	Experiment One Top Performers	33
3.14	Experiment One Top Performers Tournament	34
3.15	Experiment Two Average Fitness	34
3.16	Experiment Two Top Performers	35
3.17	Experiment Two Top Performers Tournament	35
3.18	Experiment Three Average Fitness	36
3.19	Experiment Three Self-adaptation	37
3.20	Experiment Three Top Performers	37
3.21	Experiment Three Top Performers Tournament	38
3.22	Experiment Four Average Fitness	38
3.23	Experiment Four Top Performers	39
3.24	Experiment Four Top Performers Tournament	39
3.25	Experiment Five Average Fitness	40
3.26	Experiment Five Top Performers	40
3.27	Experiment Five Top Performers Tournament	41
3.28	Experiment Six Average Fitness	41
3.29	Experiment Six Top Performers	42
3.30	Experiment Six Top Performers Tournament	42
3.31	Experiment Seven Average Fitness	43
3.32	Experiment Seven Top Performers	43
3.33	Experiment Seven Top Performers Tournament	44
3.34	Average Fitness Composite	45
3.35	Top Performers Composite	45
3.36	Top Performers Tournament Composite	46
4.1	BBAutoTune GUI Panel	52
4.2	GA Progress Monitor	55
4.3	Rank Fitness Selection Algorithm	57
4.4	Real Robot Forward Motion Data Translated and Rotated Example	59

4.5	Real Robot Forward Motion	60
4.6	Real Robot Forward Motion Distributions	61
4.7	Real Robot Forward Motion 3D Scatter Plot	62
4.8	Real Robot Forward Motion Fast-MCD Support Samples	64
4.9	Real Robot Forward Motion MD versus RD	65
4.10	Simulated Robot Axis Aligned	67
4.11	Physics Engine Parameter Influence Racquetball Environment	68
4.12	Physics Engine Racquetball Path Dissimilarity	72
4.13	Physics Engine Racquetball Path Similarity	73
4.14	Lettier Distance Algorithm	75
4.15	Experiment Two GA Metrics	76
4.16	Experiment Three GA Metrics	78
4.17	Experiment Four GA Metrics	80
4.18	Experiment Five GA Metrics	82
4.19	Simulated versus Real Motion Line Plot	84
4.20	Simulated versus Real Motion 3D Plot	85
5.1	Simulated Treads	88
5.2	Constant Velocity Locomotion Model	88
5.3	Comparative Path Plots	89
5.4	BlenderSim Arena	91
5.5	SRV-1 3D Model	92
5.6	BlenderSim Components	94
5.7	Simulated Task Points	94
5.8	HRTeam Experiment Waypoint and Task Points	97
5.9	Simulated vs. Real Robot Motion	99

List of Tables

3.1	Genetic Algorithm Parameters Overview	21
3.2	Experiment One GA Parameters	47
3.3	Experiment Two GA Parameters	48
3.4	Experiment Three GA parameters	49
3.5	Experiment Four GA Parameters	49
3.6	Experiment Five GA Parameters	50
3.7	Experiment Six GA Parameters	50
3.8	Experiment Seven GA Parameters	50
4.1	Blender Physics Parameters and Ranges	53
4.2	Real Robot Forward Motion Distribution Metrics	58
4.3	Physics Engine Parameter Influences	74
4.4	Experiment Two Best and Worst Physics Parameters Found	77
4.5	Experiment Three Best and Worst Physics Parameters Found	77
4.6	Experiment Four Best and Worst Physics Parameters Found	79
4.7	Experiment Five Best and Worst Physics Parameters Found	81
4.8	Comparison of the Top Phenotypes' Final States to the Real Robot Motion	83
5.1	Simulated vs. Real: Discrete Fréchet Distance	98
5.2	Simulated vs. Real: Hausdorff Distance	98

Chapter 1

Introduction

1.1 Genetic Algorithms

Inspired by the theory of evolutionary, genetic algorithms (GAs) attempt to solve problems that involve multidimensional state spaces [3][4]. GAs overcome problems experienced by other algorithms, such as hillclimbing, random search, or simulated annealing, by using a mixture of exploitation and exploration [4]. Running in a loop, a GA generates proposed solutions to a problem, evaluates each proposed solution's value, and then capitalizes on what it learned to produce further potential solutions. Applications of GAs include numerical function optimization, image processing, combinatorial optimization, machine learning, and civil engineering [4]. What problems are appropriate for GAs and why GAs work when they do are highly debated topics with no dominate answers [3]. With so many variations and parameters values possible, no two GA implementations are guaranteed to be identical, "...the number of variations that have been suggested is enormous. Probably everybody's GA is unique!" [3].

1.2 SimPL

Instead of diving right into the difficult problem targeted by the thesis, SimPL was an intermediate step to developing a genetic algorithm capable tuning a 3D physics engine. The problem for SimPL involved

a neural network (NN) and an on-screen paddle and ball. With the NN controlling the movement of the paddle, the GA developed for SimPL continuously tuned the weights of the NN. With each tuning of the weights, the NN became progressively better at controlling the paddle to hit the ball consistently. This simpler but still related problem provided a tractable context to conduct the needed background research of genetic algorithms. The trials and tribulations experienced while researching the problem of SimPL eased the research conducted for the thesis problem. With the knowledge gained by solving the problem for SimPL, the thesis project was setup for success.

1.3 BBAutoTune

The core work of the thesis was BBAutoTune. Using the SimPL GA as a base, the goal of BBAutoTune was to tune the Blender physics engine via a genetic algorithm such that the physics engine would closely model the motion of a real robot used in HRTeam experiments. With a few modifications, the SimPL GA was easily converted to handle the thesis problem. After collecting data on the motion of the real robot, BBAutoTune learned the necessary physics parameters needed to make the physics engine closely model the motion of the real robot. The motion of the real robot learned was its forward motion given a forward command.

1.4 BlenderSim

BlenderSim was a Blender based, 3D simulator developed for HRTeam during the summer of 2013. Initially, the physics engine in Blender was used to simulate the motion of the robots used in HRTeam experiments. However the physics engine was largely unusable and no amount of parameter tweaking provided consistent and stable results. Thus the physics engine was abandoned as the motion model and a constant velocity model was used in its place. As a proof-of-concept of the thesis, BlenderSim (and specifically the physics engine itself) was revisited using the tuned parameters found by BBAutoTune. By rerunning a previously run HRTeam experiment, the motion of the simulated, physics-based robot was compared with the motion of the real robot.

Chapter 2

Genetic Algorithms

2.1 Overview

Genetic algorithms (GAs) have been applied to various problem domains since they were made popular by John Holland in 1975 [3]. There is no guarantee that any two GAs are the same but the central idea is to use a GA to find the globally optimal solution to some multivariate state space. All of the possible states in the state space make up a fitness landscape where one state is more optimal or fitter than another. The goal of the GA is to find the global maximum or minimum point on the fitness landscape. Intuitively, one can think of the fitness landscape as a foggy mountainousness terrain. As the GA progresses, more and more of this landscape is explored as the GA searches for the highest or maybe the lowest point in the terrain depending on the particular problem the GA has been tasked with. Another central idea to GAs is the concept of mutation and crossover. Using both crossover and mutation, the GA can either expand or narrow its search of the fitness landscape. All GAs have the concept of a population where each member in the population represents a point on the fitness landscape. Initially, a GA's population is usually randomized but could be seeded with some known-to-be *good* solutions by the GA's designer [5]. As the GA runs, new populations are formed from previous populations where each population instance is known as a generation. When to stop the GA from producing new generations, known as termination, is up to the designer. Some common cases for termination include GA run time or population diversity (how different each population member

is from one another) dropping below some threshold [3].

2.2 Genomes

Taking a page from nature, GAs have the concept of genomes (also called chromosomes in some texts). Genomes make up the GA's population at any given time and each represents a possible solution (or possible partial solution) to the problem the GA is attempting to solve or rather find a optimum solution to. Various data structures ranging from strings to classes can be used to construct a genome [3]. However, at the heart of every genome is its genes or vector of variables where each variable corresponds to a dimension in the multidimensional state space the GA is searching.

2.2.1 Genes

The genes of genome represent a state in some multivariate state space inherent to the problem the GA is attempting to solve. Collectively, the genes of a genome describe a genotype while the construct they produce is known as the phenotype. Based its genes, a genotype will be evaluated by its phenotype's performance [4]. Imagine the genes of a genome being the blueprint of a motorcycle. This blueprint would be the genotype while the motorcycle built by following the blueprint would be the phenotype. If the goal was to produce the fastest motorcycle on earth, each blueprint's score would correlate to its motorcycle's maximum speed. Depending on the encoding the designer has chosen, each gene may be a bit, string, real number, natural number, or some other datum. Furthermore, not all the genes have to be of the same type but collectively could be some mixed tuple of various data types [3].

2.2.1.1 Encoding

How the variables to a problem are encoded into genes is entirely problem dependent and up to the designer. Typical encoding schemes are binary where every gene is a bit or a string of bits, a permutation where the genes are some possible order of a sequence, a real value where the genes could be real numbers, integers, or character strings, and a tree where each gene is a node in a tree that when parsed, forms some kind of expression [6]. There is no hard and fast rule governing the encoding of genes. Some have claimed that a

binary encoding is best but this notion has been contested [3].

2.3 Evaluation

Evaluation of each genome's phenotype yields the genome's fitness which is entirely based on the fitness function. Besides the encoding scheme, the fitness function is most critical aspect to get *right* when developing a GA. Fitness functions that are binary, noisy, computationally expensive, erratic, or discontinuous can produce poor results on behalf of a GA [4]. If at all possible, the fitness function should have gradual gradients and a minimal amount of local optima [4].

2.3.1 Criteria

The criteria for evaluation usually lends itself when one envisions what goals a perfect solution would meet. For certain problem domains, the criteria for evaluation may include constraints where a genome is penalized for some constraint violated [4].

2.3.2 Fitness Function

Constructing the right fitness function for the problem at hand is paramount when developing a GA. Based on the problem and the way in which the fitness function has been constructed, the GA's task usually boils down to one of either minimizing or maximizing the fitness function [3]. For problems with a unique solution where a genome is either completely correct or completely wrong with no leeway in between, one may want to include subgoals or partial credit for a genome that only provides a partial solution [4]. An analogy would be a professor giving partial credit for an answer on an exam.

Generally, the fitness function should always provide some new knowledge of how well one genome performed in comparison to others where each fitness value outputted logically and intuitively corresponds to the problem [4]. If after evaluating an entire population, all genomes have the same fitness value even though they all provide varying proposed solutions, then the GA has gained no new knowledge from which it can capitalize on as it constructs a new generation.

2.4 Operators

The tools of the GA are its operators. Many variants exist but the general operator categories are selection, elitism, crossover, and mutation. Crossover and mutation play off of another another by adding a delicate mix of exploitation versus exploration [4]. Elitism works to never lose the best solutions found so far when generating a new population. Mimicking natural selection, the selection operator provides each population member with some opportunity (however small) at passing its genes on to the next generation [4].

2.4.1 Selection

The general concept behind selection is to weight the probability of being selected for reproduction towards fitter genomes. Roulette wheel, rank, and tournament selection are some of the most well known selection methods. All have been studied in depth with each method having its own set of parameters. Some researchers have conjectured that by tweaking the parameters of each method, all can be constructed to have similar performance and thus no method is superior over another [4].

2.4.2 Elitism

Elitism is one of the more simple operators. After the entire population has been evaluated, the top n performers are directly copied into the new population during the reproduction cycle. Note that the n elites are not necessarily immune from being selected for crossover and/or mutation. In addition to being directly copied into the next generation, the n elites could also be crossed and/or mutated. Whether or not the elites are also considered during selection is up to the designer. Deciding on the size of n —in relation to the population size—becomes a delicate balance between premature convergence and the GA losing information about the best solutions found so far thereby forcing it to possibly run longer relearning what it already learned previously [7].

2.4.3 Crossover

Crossover employs the notion of exploitation where knowledge gained about points on the fitness landscape is capitalized on to find better points. Exploitation differentiates GAs from random search [4]. Selecting two

genomes as parents, the crossover operator produces one or more offspring such that the offspring have some genes from one parent and some genes from parent two. Typical variants of crossover are 1-point crossover, m -point crossover, and uniform crossover where the offspring receives a random number of genes from parent one and a random number of genes from parent two [3]. Note that crossover does not always occur after selection but rather occurs based on some probability known as the crossover probability.

2.4.4 Mutation

Mutation is the exploration component of a GA where mutant offspring may discover previously uncharted points on the fitness landscape [4]. Coupled with a mutation step, the mutation operator disperses current genomes in the population to random points on the fitness landscape like dandelion seeds blowing in the wind; the larger the mutation step, the greater the dispersion. Like most aspects of a GA, there are many mutation operator variants depending what gene encoding scheme was employed. Central to all variants, however, is some random number generator.

Mutation can occur on a gene by gene or on a genome by genome basis. With some probability of occurring—known as the mutation probability—a genome’s genes will be mutated (randomly altered) thereby producing a mutant offspring. These mutant offspring allow a GA to escape local optimums and also help to keep the population diverse [3]. As the GA progresses, the diversity of the population will ultimately depend on the mutation probability but by employing some self-adaptation mechanism, the mutation probability can vary over time [3]. Note that like the mutation probability, the crossover probability could also be self-adapted over time as well.

2.5 Population

2.5.1 Initialization

Typically, at the start of a GA run, the population is randomly generated but it could be seeded with some high performing genomes found by an earlier run or by some other mechanism [5][3]. However, if one does seed the initial population, this may lead to premature convergence [3]. Determining the population size

and whether or not the population size remains constant throughout the run of the GA is a debated topic with no straight forward answers [3].

2.5.2 Reproduction

Once every genome in the population has been evaluated, a GA enters into its reproduction cycle producing a new population known as a generation. There are many approaches to the reproduction cycle but the most basic is performing selection, crossover, and then mutation in a loop until the new population size reaches the old population size [4]. This basic approach is known as the generational approach with a generational gap of one between each generation. The generational gap is how much of an old population is replaced by a new population. A generational gap of one indicates that the new generation entirely replaces the old generation or in other words, parents never coexist with offspring during any given generation [4]. Another approach is the steady-state or incremental approach where the generational gap is greater than one with only a few offspring being produced and a few population members being terminated during each reproduction cycle [3]. Human beings follow a steady-state approach with some dieing and some coexisting with their children after having reproduced.

2.6 Termination

GAs are inherently a stochastic process and will run forever unless constructed otherwise [3]. Deciding when to terminate is entirely up to the designer and their needs. The simplest method of termination is time where the GA is run for some amount of time and then stopped. A more sophisticated method involves terminating a GA after the population has dropped below some threshold of diversity [3].

Population diversity coincides with population convergence. Gene convergence occurs when 95% of the population shares the same gene value while population convergence occurs when every gene in the population has converged [4]. A converged population will be minimally diverse at the gene, genotype, and phenotype level [3].

Chapter 3

SimPL

3.1 Overview

SimPL is an asymmetric autonomous pong clone with one paddle and one ball. SimPL is comprised of web-based technologies: HTML5, JavaScript, CSS, AJAX, MySQL, and PHP. At the time of this writing, SimPL can be viewed at <http://www.lettier.com/simpl/>. The paddle in SimPL is controlled by a feed-forward neural network. The neural network's weights are tuned via a genetic algorithm.

The focus for SimPL was to learn about and to cultivate a genetic algorithm capable of tuning parameters with respect to a fitness landscape thereby producing an optimum solution to a given parameter space. The genetic algorithm developed for SimPL was used as the basis for the genetic algorithm needed to solve the harder problem of tuning a 3D physics engine.

3.2 Implementation

3.2.1 Arena

The arena for SimPL resides in a browser window. Four transparent walls reside at the top, right, bottom, and left of the screen. The arena contains a ball and paddle with the paddle affixed to the far left of screen and the ball originating from the far right of the screen. See Figure 3.1.

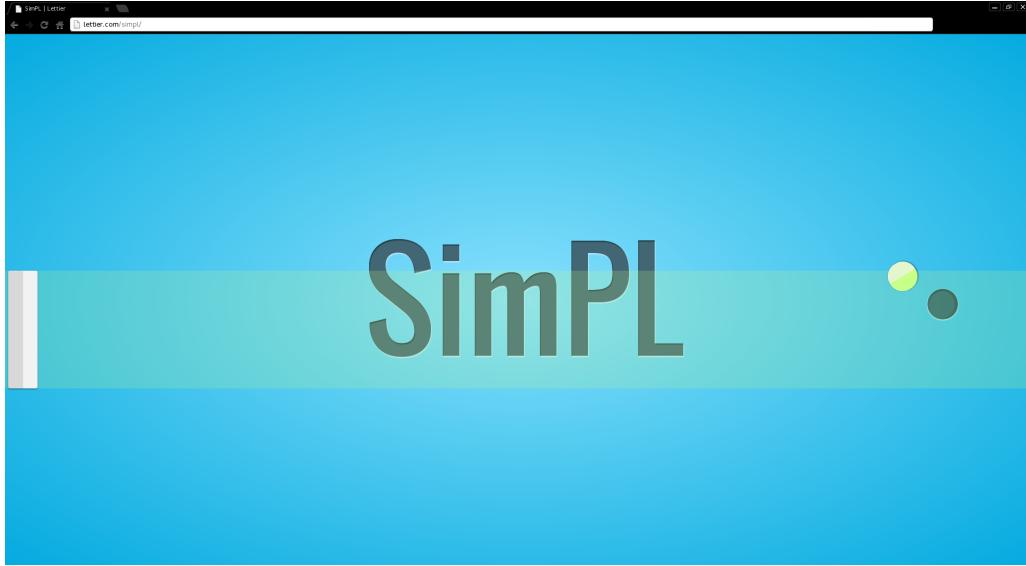


FIGURE 3.1: Here you see the SimPL arena containing the paddle and ball.

3.2.2 Ball

The ball is a physics-based dynamic object. Its starting position and starting velocity magnitude are the same at the start of every round¹. Just before the beginning of a round, a random angle in the range $[135^\circ, 225^\circ]$ is chosen as the ball's starting angle. See Figure 3.2. The ball's position is managed by the physics engine which responds to any collisions against the arena walls and/or the paddle.

If the ball collides with the left wall, the round is over. Otherwise, if the ball collides with the top, right, or bottom wall, the ball is bounced back into the arena via its angle-of-reflection based on its collision angle-of-incidence. Collisions with the paddle work in the same fashion where the ball is bounced back via its angle-of-reflection based on its collision angle-of-incidence. See Figure 3.3.

Each collision the ball makes reduces its velocity magnitude. Let m denote the ball's velocity magnitude. The formula used is $m = m - (m * 50\%)$. Once the ball's velocity magnitude drops below 100 the round is over.

¹A round is defined as the time from when the ball is launched from its starting position to either the time at which the ball leaves the left side of the arena or at the time in which the ball's velocity magnitude drops below 100.

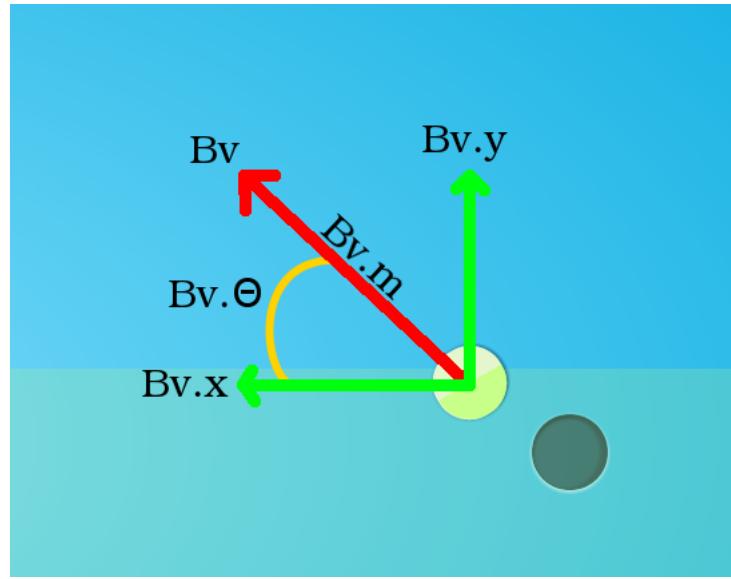


FIGURE 3.2: Here you see the ball's dynamic physics properties.

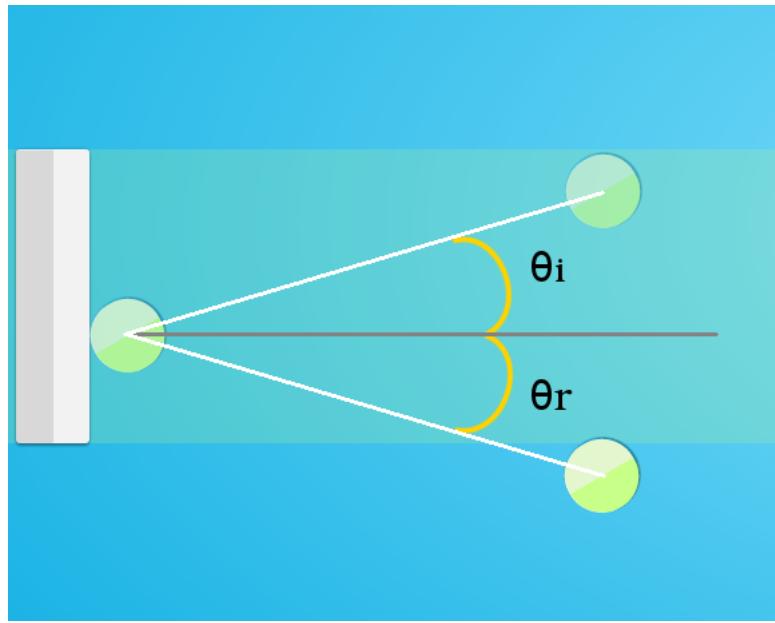


FIGURE 3.3: Here you see the ball's collision angle-of-incidence θ_i and its angle-of-reflection θ_r .

3.2.3 Paddle

The paddle is a physics-based dynamic object that has a fixed velocity angle of either 90° or 270° . See Figure 3.4. Its starting position as well as its starting velocity magnitude are the same at the start of every round.

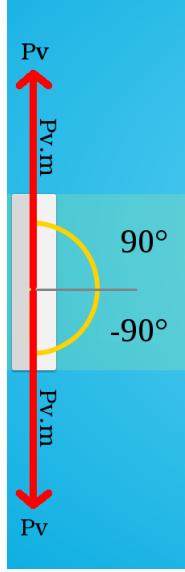


FIGURE 3.4: Here you see the paddle’s dynamic physics properties.

The paddle’s direction and speed are regulated by the output of the neural network. The output of the neural network is in the range $[-1, 1]$. A neural network output of 0 results in the paddle not moving from its current position. A neural network output in the range of $(0, 1]$ results in the paddle traveling up by some percentage of its starting velocity magnitude. A neural network output in the range of $[-1, 0)$ results in the paddle traveling down by some percentage of its starting velocity magnitude. For example, say the neural network output is 0.68 and let m denote the paddle’s velocity magnitude. The paddle’s velocity angle would be set to 90° and its velocity magnitude would be set to $m_{new} = 0.68 * m_{initial}$ where $m_{initial}$ is the paddle’s starting velocity magnitude (specifically $1000 \frac{\text{pixels}}{\text{second}}$). Here the paddle can always travel as fast or faster than the ball given that the starting velocity magnitude of the paddle and ball are the same at the start of every round. However, while ball’s velocity magnitude is reduced after every collision the ball

makes, the paddle has the possibility of always traveling as fast as it could at the start of the round (albeit depending on the neural-network output). Thus, it is never the case that the paddle does not have the possibility of never reaching the ball in time during the duration of any round. In other words, it is never the case that the neural-network outputs the correct movement—given the paddle’s and ball’s states—but the paddle misses out on fitness it could have gained had it been traveling fast enough to reach the ball. The paddle can always reach the ball given the neural network output is correct given the paddle’s and ball’s states. Thus, at any time t during any round R_i , the paddle’s velocity magnitude is $0 \leq m \leq 1000 \frac{\text{pixels}}{\text{second}}$ as $m = |[-1, 1]| * m_{initial}$ where $|[-1, 1]|$ is the absolute value of the neural-network’s output range.

Collisions can occur for the paddle between the top wall, the bottom wall, and the ball. Collision with either the top or bottom wall results in the paddle’s top or bottom being placed just before the wall. Collision with the ball results in no change of movement for the paddle—the paddle merely continues moving as it was before the collision occurred with the ball.

3.2.4 Physics Engine

All dynamic and static objects are registered with the physics engine before the first round. Once every draw loop of the SimPL game, the physics engine tests for collisions between dynamic objects and other dynamic objects and between dynamic objects and static objects. Those dynamic objects that are found to be colliding with either another dynamic object and/or static object are flagged as such and their collisions are handled accordingly. For those dynamic objects that are not colliding, their positions are updated based on their velocity.

3.2.5 Neural Network

The neural network is a feed-forward neural network that contains one input layer consisting of six input nodes, one hidden layer consisting of 5 hidden nodes, and one output layer consisting of one output node [8]. Each threshold input to the hidden nodes and the output node is included among the weights of the network and thus are optimized or tuned via the genetic algorithm. In total, there are 41 weights $((6 + 1) * 5 + (5 + 1) * 1 = 41)$ contained in the network. See Figure 3.5. Thus, there are 41 genes per

genome in the genetic algorithm's population. All output from the hidden nodes and the output node are run through a sigmoid, hyperbolic-tangent-activation function ($\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$) resulting in an output range of $[-1, 1]$.

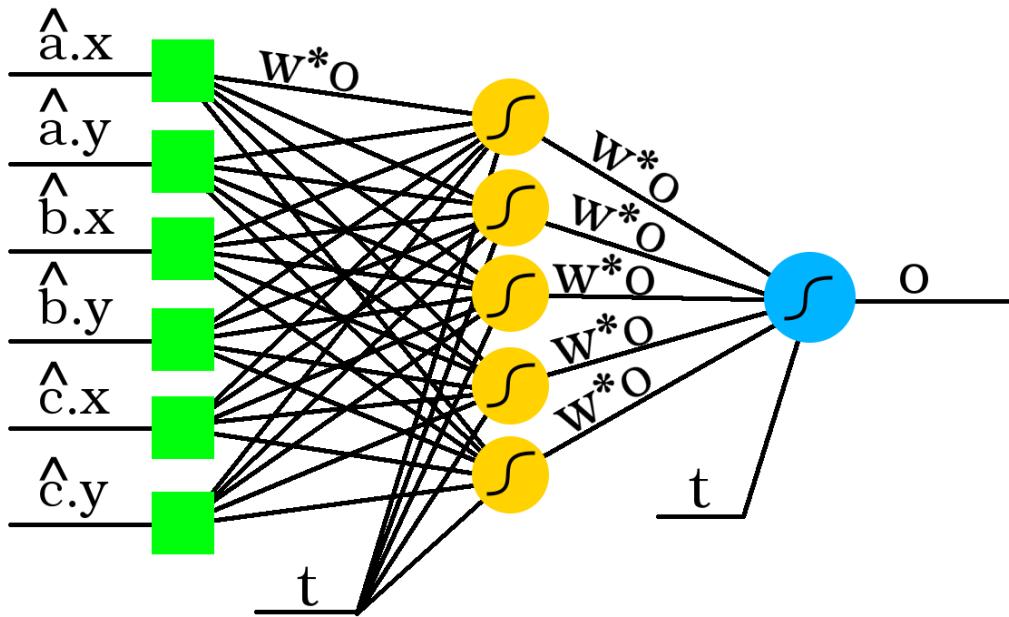


FIGURE 3.5: Here you see the neural network as constructed in SimPL.

Input to the neural network is normalized generating three unit vectors: from the ball's center to the paddle's center, the ball's velocity, and from the window's origin to the paddle's center. See Figure 3.6. These three unit vectors are broken down into their components resulting in six inputs to the neural network. See Figure 3.7.

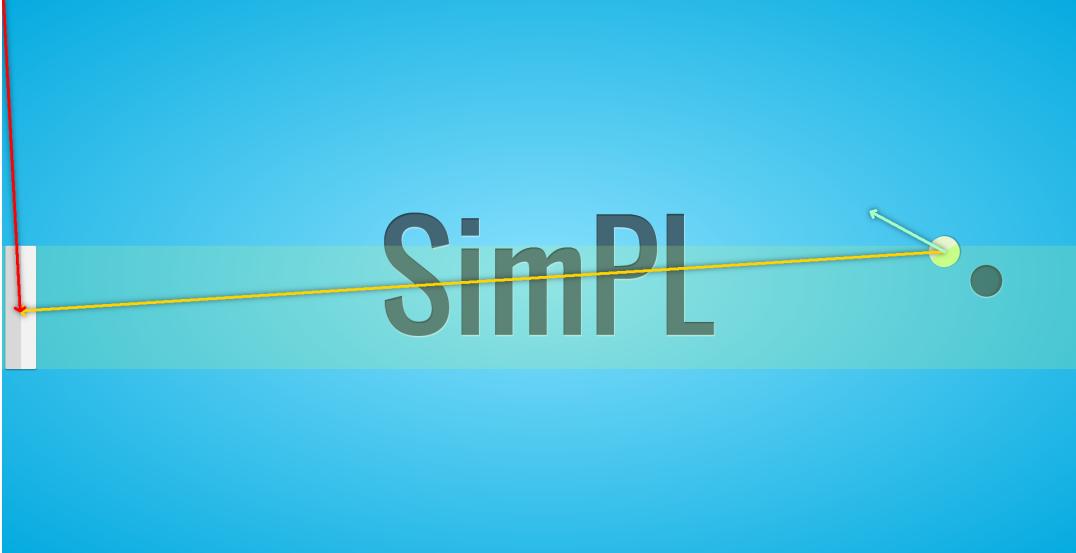


FIGURE 3.6: Here you see the three input vectors to the neural network. Note that these vectors are normalized thereby turning them into unit vectors.

3.2.6 Genetic Algorithm

Instead of using back-propagation to train the weights of the neural network, a genetic algorithm is used to optimize or tune the weights of the neural network [8]. The genetic algorithm consists of a population of genomes with each having a fitness property and an array of genes. For SimPL, the genes represent a solution of weights to be used in the neural network. Each genome is evaluated by a fitness function. As the genomes evolve over generations to produce fitter genomes, the neural network becomes increasingly accurate at outputting what the paddle should do (move up, stay still, or move down) based on the state of the ball and the paddle.

The genetic algorithm contains four operators that work to produce fitter generations during the creation of a new population. The operators include: the elitism operator, the selection operator, the crossover operator, and the mutation operator. Initially, the genetic algorithm creates a random population of genomes. These initial genomes have zero fitness and a fixed number of genes. Each gene in every genome is given a random value sampled from a uniform distribution coinciding with some valid range. The genes are the input parameters to the mechanism the genetic algorithm is working to optimize. In the case of SimPL,

```
Paddle:  
    id: paddle  
    magnitude: 134.7752765851305  
    PI: 3.141592653589793  
    angle: 1.5707963267948966  
  
Ball:  
    id: ball  
    magnitude: 205.891132094649  
    PI: 3.141592653589793  
    angle: 2.5307274153917776  
  
NN Input: 0.989 0.146 -0.819 0.574 0.082 0.997  
  
NN Output:  
    0: -0.13477527658513053  
  
Current Genome: 1  
  
NN:  
    number_of_inputs: 6  
    number_of_outputs: 1  
    number_of_hidden_layers: 1  
    neurons_per_hidden_layer: 5  
    bias: -1  
  
NN Weights:  
    0: -1  
    1: 0.9813534922958044  
    2: -1  
    3: 1  
    4: 0.19044709740921306  
    5: 0.45017305545771064  
    6: 0.8754844796792515  
    7: -0.21012540145260464  
    8: 0.6151249941406718  
    9: 1  
    10: -1  
    11: -0.6453344198759198  
    12: 0.17499051310257202  
    13: 0.31071486210448007  
    14: 0.9123286228335877
```

FIGURE 3.7: Here you see the normalized input to the neural network.

the mechanism is the neural network and the parameters—of each genome—represent the 41 weights of the neural network. Each parameter has a valid range of $[-1, 1]$.

Once every genome in the population has been evaluated by the fitness function, the genetic algorithm constructs a new population from the previous generation. First, the elitism operator selects the n fittest genomes from the old generation. These elite genomes are allowed to survive intact (however their fitnesses are reset to zero) and are placed into the new generation. Second, the genetic algorithm enters into a loop creating new genomes via the crossover operator and the mutation operator until an entirely new generation has been created. This new generation goes on to be evaluated as their predecessors were and the cycle repeats until some termination criteria is met. See Figure 3.8.

```

BEGIN
    Generate a random population  $P$ 
    While not terminate do
        Evaluate population  $P$ 
        Create empty population  $P'$ 
        Sort  $P$  in order of fitness
        Select  $n$  fittest from  $P$  and add them to  $P'$ 
        While size of  $P' \leq$  population size do
            If perform crossover and mutation in sequence then
                Select two genomes from  $P$ 
                Generate two offspring via crossover with probability  $C$ 
                Mutate the two offspring generated from crossover with probability  $M$ 
                Add the generated offspring to  $P'$ 
            Else
                Select two genomes from  $P$ 
                Generate two offspring via crossover with probability  $C$ 
                Add the offspring to  $P'$ 
                Select one genome from  $P$ 
                Mutate the selected genome with probability  $M$ 
                Add the mutated offspring to  $P'$ 
            End if
        End while
         $P = P'$ 
    End while
END

```

FIGURE 3.8: Here you see the basic genetic algorithm.

The selection operator uses roulette selection where the probability of some genome being selected is proportional to their fitness. Roulette selection and roulette selection with rank fitness was experimented

with as outlined later [9]. The crossover operator takes two selected genomes to produce offspring where the offspring have some combination of their parents' genes. SimPL uses one-point crossover throughout each experiment. To select the crossover point per crossover operation, the crossover operator samples a random integer from a uniform distribution ($crossoverPoint = X \sim U(0, n - 1)$ where n is the number of genes per genome). The mutation operator takes a selected genome and mutates its genes using various means. If the mutation scope is at the gene level, the mutation operator traverses through the genome's gene array where for each gene, the mutation operator samples a random integer from a uniform distribution ($X \sim U(0, 1)$) and if this random integer X is less than or equal to the mutation probability, the mutation operator mutates the gene value either via uniform mutation ($geneValue_i = X \sim U(0, 1) * \varepsilon$) or Gaussian mutation ($geneValue_i = X \sim N(\mu, \sigma^2)$). If the mutation scope is at the genome level, the mutation operator samples a random integer from a uniform distribution ($X \sim U(0, 1)$) and if this random integer X is less than or equal to the mutation probability, the mutation operator mutates all of the genome's gene values one after the other via either uniform mutation ($geneValue_i = X \sim U(0, 1) * \varepsilon$) or Gaussian mutation ($geneValue_i = X \sim N(\mu, \sigma^2)$). These means of mutation were experimented with as outlined later. Crossover and mutation can be carried out in sequence or can be done separately with one operator not interfering the other operator's offspring. This was experimented with as outlined later.

Crossover and mutation are not guaranteed to always occur as the genetic algorithm produces new genomes from the old genomes. Rather, crossover and mutation occur based on some probability. If the crossover and mutation probabilities are both set to 1 then they always occur. Before the genetic algorithm is initialized, the crossover and mutation probabilities are set [10]. These initial crossover and mutation probability values can be arbitrary or can be based on some a-priori knowledge of the fitness landscape. The probabilities can change over time or can remain static throughout the run of the genetic algorithm. Different static settings and self-adaptation of the probabilities were experimented with as outlined later.

The fitness function evaluates each genome in the current population based on some fitness criteria. The fitness criteria coincides with finding the optimum solution to the parameter space of the mechanism the genetic algorithm is producing fitter and fitter solutions to. For SimPL, the parameter space is the weights of the neural network. Each genome represents a solution or point in the weights space. An optimal solution

in the weights space will make the neural network always give a correct output as to what movement the paddle must make based on the ball's and the paddle's current states. This will result in the paddle always following the ball or in other words, the paddle will never let the ball leave the left side of the screen. Thus the fitness criteria for SimPL could include how much or how well the paddle follows the ball and/or how many times the paddle hits the ball. Different fitness criteria were experimented with as outlined later.

3.2.7 Database Manager

A database manager interfaces with a remote MySQL server either asynchronously or synchronously. Experiment data is recorded in the MySQL database as the experiment is carried out. Additionally, each generation produced by the genetic algorithm is logged to the database. Upon visiting the SimPL site, the last generation produced can be loaded into the genetic algorithm thereby allowing the simulation to pick up where it left off last.

3.3 Platform

The browser window size has a significant influence over the fitnesses of the genomes being evaluated. While the arena fits to whatever size the browser window is, the size of the paddle and the ball do not directly change in proportion to the window size. Thus, a small browser window gives the paddle less screen real estate to cover in comparison to a large browser window. For example, imagine a browser window size with a height as large as the paddle's height. Here the paddle can never move—it always follows the ball and always hits the ball. Thus, the resulting fitnesses observed would be erroneously high. Therefore, all experiments were run using the same browser window size.

Experiments were run on a Lenovo Z580 Ideapad laptop running 64-bit Fedora 18 and was equipped with an Intel Core i5 CPU running at 2.5 GHz and 7.7 GiBs of memory. Google Chrome version number 30.0.1599.114 was used on the laptop. Screen resolution was 1366×768 . Browser window size (both reported and actual) was 1366 pixels wide by 681 pixels tall. Paddle size reported (by the browser) was 50 pixels wide by 200 pixels tall and the actual size was the same. Ball size reported (by the browser) was 50 pixels wide by 50 pixels tall and the actual size was the same. The paddle had $681 - 200 = 481$ pixels of available

screen-space to traverse.

3.4 Experimental Designs

Each SimPL experimental design (with the exception of experiments six and seven) was constructed in such a way as to focus on a set of one or more facets to the genetic algorithm. See Table 3.1. With each progressive experiment, the facets not focused on were kept the same from the previous experiment. The goal was to create a genetic algorithm that would eventually—with efficiency—produce the optimal set of neural-network weights needed in order to generate a paddle that performs as well as the performance-standard paddle constructed in experiment six. Ultimately, the performance of any paddle in SimPL is how long it keeps the ball-in-play before the termination of any round.

Experiments one and two revolved around the fitness function, the selection operator, and the static crossover and static mutation properties of the genetic algorithm. Experiments three, four, and five revolved around the self-adaptation of the crossover and mutation probabilities and whether allowing the mutation operator to disrupt the offspring, produced by the crossover operator, degraded the performance of the genetic algorithm. Experiment six revolved around constructing a paddle as an optimal-performance standard by which any other paddle could be measured by. Finally, experiment seven revolved around constructing a randomly behaving paddle thereby producing a performance lower bound.

For each experiment, the genetic algorithm was run for 100 generations where for each generation, the population’s average fitness was recorded. Once every 10th generation, the current population’s top performing genome was saved. After the run of the genetic algorithm was over, the saved top-performers were run in a tournament. This tournament consisted of running each top-performer for five rounds where for each top performer, the time in seconds they kept the ball-in-play per round was recorded. Once an every-10th-generation-top-performer was done with their five rounds, the average of their ball-in-play time per round was calculated and recorded.

GA Parameters × Experiment	One	Two	Three	Four	Five	Six & Seven
<i>Population Size</i>	10	10	10	10	10	10
<i>Fitness Function Type</i>	Partial/Full	Full	Full	Full	Full	Full
<i>Number of Elite Offspring</i>	2	2	2	2	2	10
<i>Roulette Selection - Actual Fitness</i>	True	False	False	False	False	N/A
<i>Roulette Selection - Rank Fitness</i>	False	True	True	True	True	N/A
<i>Sequential Crossover & Mutation</i>	True	True	False	False	True	N/A
<i>Self-adaptation</i>	False	False	True	False	False	N/A
<i>Crossover Type</i>	One-point	One-point	One-point	One-point	One-point	N/A
<i>Crossover Probability</i>	0.7	0.8	0.5	0.7816	0.7816	N/A
<i>Mutation Type</i>	Uniform	Gaussian	Gaussian	Gaussian	Gaussian	N/A
<i>Mutation Scope</i>	Gene	Gene	Genome	Genome	Gene	N/A
<i>Mutation Probability</i>	0.1	$(\frac{1}{n}) = (\frac{1}{41}) \approx 0.0244$	0.5	0.2184	0.2184	N/A
<i>Mutation Step</i>	$X \sim U(0, 1) * 0.3$	$X \sim N(\mu, \sigma^2)$	$X \sim N(\mu, \sigma^2)$	$X \sim N(\mu, \sigma^2)$	$X \sim (\mu, \sigma^2)$	N/A
<i>Mutation Step μ</i>	N/A	Gene Value	Gene Value	Gene Value	Gene Value	N/A
<i>Mutation Step σ</i>	N/A	0.5	$M_{Prob.}$	$M_{Prob.}$	$M_{Prob.}$	N/A

TABLE 3.1: Here you see the genetic algorithm parameters used per experiment. The highlighted cells indicate the experimental variables per experiment.

Experiment six and seven are included in the table for completeness but no evolution ever took place.

3.4.1 Experiment one: use of a full and partial credit granting fitness function.

Experiment one centered around the use of a fitness function that would give full and partial credit based on the behavior of the neural network and thus the paddle. The genetic algorithm parameters used are listed in Table 3.2.

During each generation of the genetic algorithm, each genome from the population was allowed to run one round until the round terminated either due to the ball leaving the left side of the screen or the ball's velocity magnitude dropping below 100. Note that during the round, the paddle's movement (in relation to the ball's movement) was tracked via an array as well as how many times the paddle hit the ball during the round. Once the round terminated, the genome was evaluated by the fitness function. During evaluation, if the genome's phenotype (the paddle's observable characteristics) followed the ball (from one draw loop to next) it would be given a positive partial fitness credit of 0.1 every time it followed the ball. If the phenotype managed to collide with the ball, the genome was given a full fitness credit of 1 every time it hit the ball. However, if the phenotype moved away from the ball (from one draw loop to the next), it was given a negative partial fitness credit of -0.1 every time it moved away from the ball. Lastly, if the phenotype didn't move at all (while the ball moved from one draw loop to the next) it was given 0 fitness every time it did not move. At the end of the fitness function, all of these partial and full fitness credits were summed giving the currently-being-evaluated genome its total fitness. If the total summed fitness was less than zero, the total summed fitness was set to zero. That is, no genome's fitness—after having been evaluated by the fitness function at the end of its round—was ever negative.

To facilitate tracking the paddle's movements in relation to the ball's movements (during the round), the absolute difference in heights between the paddle's center and the ball's center were recorded every draw loop into an array. Once the genome was ready to be evaluated, this array of differences was analyzed linearly in pairs, that is, $A[i]$ was compared with $A[i + 1]$. If $A[i] > A[i + 1]$ this indicated that the paddle's center was moving closer to the ball's center and thus the genome was awarded a positive partial credit fitness. If $A[i] < A[i + 1]$ this indicated that the paddle's center was moving away from the ball's center and thus the genome was awarded a negative partial credit fitness. If $A[i] = A[i + 1]$ this indicated that the paddle's center was neither moving toward nor away from the ball's center and thus the genome was awarded 0 fitness. A

special case for $A[i] = A[i + 1]$ was that if $A[i] = 0$ and thus $A[i + 1] = 0$ as well, the paddle was awarded a positive partial fitness as the paddle's center was dead center with the ball's center and therefore the paddle was directly in the path of the ball which is ultimately the goal—that is, the paddle should always match its center with the ball's center thereby always preventing the ball from leaving the arena.

The hypothesis for using a full and partial fitness credit schema was that every new generation produced would have genomes that at least performed somewhat better than their predecessors at the optimal behaviors. Those optimal behaviors are following the ball and hitting the ball. Originally, only hitting the ball was going to be the fitness criteria. However, early generations may never hit the ball and thus would have zero fitness. Genomes that at least followed the ball could have been erroneously discarded due to having zero fitness. By giving partial credit for at least following the ball, it was hypothesized that it would seed early generations with promising genomes or rather promising solutions to the parameter space. Picture a classroom of students taking a test. Upon evaluation, it is either all or nothing credit per question and no student finds the solution to any problem. In other words, every student received a zero. This would give the teacher no information as to the quality of the students at least in terms of comparing them to one another. However, if partial credit is given for getting part of the solution correct, then this would give at least some information as to who performed well among the students. In other words, it was hypothesized that by using a finer grained fitness function, there would be some information gained as to the fitness of one genome compared to another (aiding elitism and the selection process) versus no information gained using only a coarsely grained fitness function resulting in every genome having a fitness of zero after being evaluated.

3.4.2 Experiment two: use of a simplified fitness function, use of rank fitness in selection, higher crossover probability, mutation probability based on the number of genes per genome, and a Gaussian distribution sample mutation step.

Experiment two had a simplified fitness function, used a genome's rank fitness during selection, increased the crossover probability, based the mutation probability on the number of genes per genome, and used Gaussian

distribution sampling as the mutation step. The genetic algorithm parameters used are listed in Table 3.3.

Going from awarding full and partial credit fitness based on two behaviors to only awarding a fitness credit of 1 if the paddle was in the path of the ball per every draw loop, the fitness function was simplified. Here the paddle doesn't necessarily have to be dead center to the ball but rather the paddle's top must be at or above the ball's top while at the same time the paddle's bottom has to be at or below the ball's bottom. See Figure 3.9. The reasoning behind this was that if the paddle were to always be in the path of the ball then it would always hit the ball and thus the paddle would never let the ball leave the arena or in other words the paddle would exhibit the optimum desired behavior. Therefore, this fitness function correctly evaluates the paddle's (or rather the genome phenotype's) behavior in relation to ball's movement throughout any round.

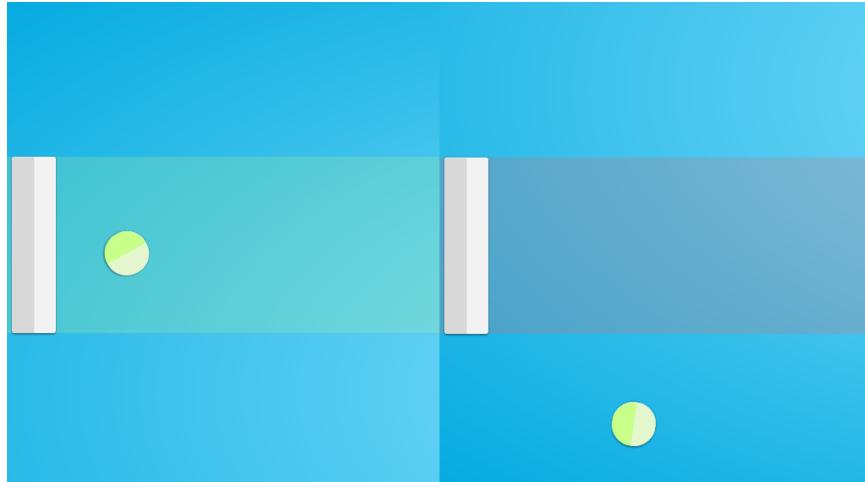


FIGURE 3.9: Here you see the simplified fitness function graphically where the paddle on the left is gaining fitness (as indicated by the green-hued bar) while the paddle on the right is gaining no fitness (as indicated by the purple-hued bar).

Instead of using the actual fitness of any particular genome in the population to determine its probability of being selected during the roulette wheel selection process, a genome's rank fitness was used to determine its probability of being selected [9]. It was observed during early runs of the genetic algorithm that after evaluation the population had wide gaps of fitness. The genomes with zero or relatively low

fitness had absolutely little to no chance of being selected for crossover and mutation while the genomes with a relatively high fitness had a high chance of being selected for crossover and mutation. These early top performers were continuously selected, crossed, and mutated generation after generation. Population diversity dwindled thereby causing the population to converge too early (due to an ever narrowing search of the fitness landscape) resulting in poor performance on behalf of the genetic algorithm. Thus it was hypothesized that by using rank fitness instead of actual fitness to determine a genome's probability of being selected, population diversity would remain sufficient generation after generation thereby avoiding early convergence.

Rank fitness is a genome's fitness according to their index in the population after the population is sorted in increasing order of fitness [11]. After sorting the population, genome one is given a fitness of one, genome two is given a fitness of two and so on and so forth until genome n is given a fitness of n or rather the population size. See Figure 3.10. Now all genomes have a better chance of being selected to undergo crossover and/or mutation thereby keeping the population diversity high and thus keeping the search scope of the fitness landscape large resulting in better performance of the genetic algorithm.

Crossover probability was increased from 0.7 to 0.8. This of course would result in more observed crossovers being generated as new populations were created. Since crossover produces an offspring solution somewhere between its parents in the fitness landscape, it was hypothesized that by increasing the crossover probability the local search capability of the genetic algorithm would also increase.

Based on empirical studies performed by others, setting the mutation probability to $\frac{1}{n}$ —where n is the number of genes per genome—is a sufficient enough amount of random search to allow the genetic algorithm to escape local maxima in the fitness landscape [12]. The reasoning behind $\frac{1}{n}$ is that per mutation on a gene-by-gene basis, only one gene is mutated on average. In addition to changing the mutation probability, the mutation step was changed from adding and/or subtracting a percentage of a maximum perturbation parameter value to and/or from the gene value to sampling a value from a Gaussian distribution where the distribution $N(\mu, \sigma^2)$ is defined by the mean μ being the gene's current value before mutation and the standard deviation σ being one fourth the valid range of the gene/parameter $\left(\frac{1-(-1)}{4} = 0.5\right)$. Here the standard deviation σ is one fourth the range and thus most of the sampled values will be within two stan-

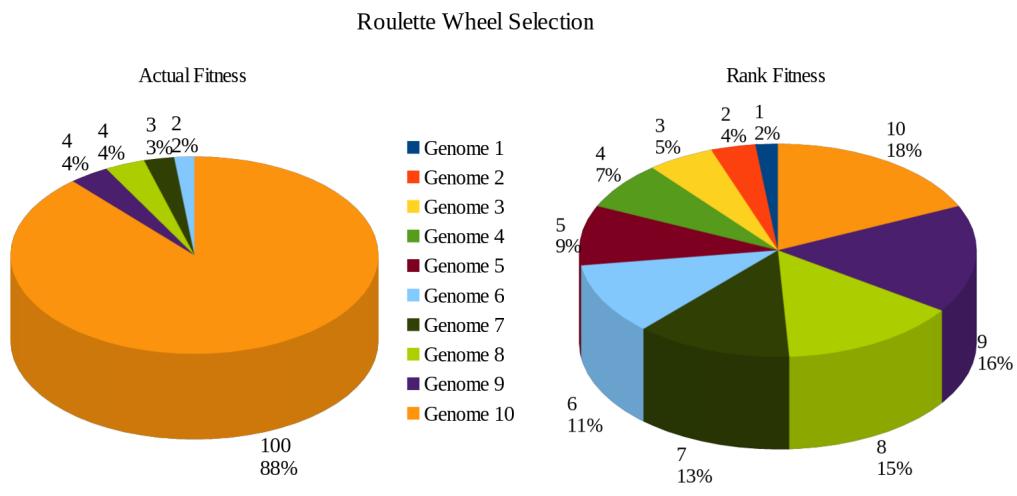


FIGURE 3.10: Here you see roulette selection using actual fitness versus rank fitness for a population of 10 genomes.

On the left, the integers are the genomes' actual fitness and the percentages are their probabilities of being selected.

On the right, the integers are the genomes' rank fitness and the percentages are the genomes' probabilities of being selected. Observe that Genome 10 no longer dominates the wheel when using its rank fitness (10) versus using its

actual fitness (100).

dard deviations of the mean μ or rather the gene value. Any sampled gene value that was outside the valid range of $[-1, 1]$ was clipped to the valid range. Note that by going with this new mutation step schema, the maximum perturbation parameter to the genetic algorithm could be removed thereby lessening the amount of parameters—to the genetic algorithm—that need to be evaluated.

3.4.3 Experiment three: self-adaptation of crossover and mutation probabilities with the crossover and mutation operators working in parallel.

Experiment three involved self-adaptation of the crossover and mutation probabilities. The genetic algorithm parameters used are listed in Table 3.4.

As outlined in [10], the crossover and mutation probabilities self-adapt based on the crossover and mutation operators' ability to produce fitter genomes from one generation to the next. To facilitate the self-adaptation, the crossover and mutation operators' viability to produce fitter and fitter offspring needed to be tracked from one generation to another. As the operators were being evaluated on their own accord, they were not allowed to interfere with each others' offspring and thus they were not run in sequence but rather were run in parallel. With each new generation produced, elite offspring were marked accordingly as well as offspring produced by crossover and offspring produced by mutation. For those offspring that were created by crossover, their parents' weighted-mean fitness was annotated along with their offspring. Annotated meaning it was recorded in the offspring's data structure later being used to calculate the crossover operator's progress. This weighted-mean fitness was calculated based on the crossover point which determines what percentage of genes came from parent one and what percentage of genes came from parent two. For example, let there be 41 genes per genome and let the crossover point be gene 10. Thus, offspring one received 10 genes from parent one and received 31 genes from parent two while offspring two received 10 genes from parent two and received 31 genes from parent one. Therefore, offspring one's weighted-mean parent fitness would be calculated as $\overline{PF} = (PF_1 * \frac{10}{41}) + (PF_2 * \frac{(41-10)}{41})$ where \overline{PF} is the weighted-mean parent fitness while offspring two's weighted-mean parent fitness would be calculated as $\overline{PF} = (PF_2 * \frac{10}{41}) + (PF_1 * \frac{(41-10)}{41})$. For those offspring created via mutation, their parent fitness was whatever the fitness was of the pre-mutated genome.

With the genomes marked as to how they were created and their parent fitness annotated, now when it came time to produce a new population, each operators' ability to produce fitter offspring could be tracked and calculated. Once a population was evaluated, all genomes created by crossover were used to calculate the average crossover progress and all genomes created by mutation were used to calculate the average mutation progress. If the crossover progress average was greater than the mutation progress average then the crossover probability would be adjusted up and the mutation probability would be adjusted down. Alternatively, if the crossover progress average was less than the mutation progress average then the crossover probability would be adjusted down and the mutation probability would be adjusted up. If the crossover progress average equaled the mutation progress average then neither were adjusted. In other words, for example, if the crossover operator produced fitter genomes greater than the mutation operator did for the last generation, then the probability of creating offspring via crossover would be increased and the probability of creating offspring via mutation would be decreased for the creation of the next generation. Adjustment of the crossover and mutation probabilities were handled by the adjustment parameter. Let the adjustment parameter be denoted as θ . Here θ was self-adjusted as well, as outlined in [10]. Once the probabilities were adjusted they were clamped to the range [0.001, 1.0]. With a minimum probability of 0.001, there would always be some crossover and/or mutation albeit not much. See Figure 3.11. Note that to insure a level playing field, both the crossover and mutation probabilities were set to an initial value of 0.5 before the start of the genetic algorithm.

Unlike previous experiments, the mutation probability was not set on a gene-by-gene basis but rather on a whole genome-by-genome basis. Every time through the population creation loop, a random float value was sampled from a uniform distribution between 0.0 and 1.0. If this random float value was less than or equal to the mutation probability, every gene in the selected genome was mutated using the Gaussian distribution mutation step method outlined earlier. However, the standard deviation was set to the mutation probability instead of it being statically set to 0.5 as before. The reason being that a high mutation probability would give way to a larger mutation step (since the standard deviation would be relatively large) allowing for larger random leaps around the fitness landscape while a low mutation probability would give way to a smaller mutation step (since the standard deviation would be relatively small) allowing for a more

```

BEGIN
    Population  $P$  has been evaluated
     $cCount = mCount = 0$ 
     $CP_{sum} = MP_{sum} = 0$ 
     $\overline{CP} = \overline{MP} = 0$ 
    For  $j = 1$  to population size do
        If  $P[j]$  created by crossover then
             $CP_{sum} = CP_{sum} + (P[j].fitness - P[j].parentFitness)$ 
             $cCount = cCount + 1$ 
        Else if  $P[j]$  created by mutation then
             $MP_{sum} = MP_{sum} + (P[j].fitness - P[j].parentFitness)$ 
             $mCount = mCount + 1$ 
        End if
    End for
     $\overline{CP} = \frac{CP_{sum}}{cCount}$ 
     $\overline{MP} = \frac{MP_{sum}}{mCount}$ 
    If  $P.bestFitness > P.worstFitness$  then
        Adjustment  $\theta = 0.01 * \frac{P.bestFitness - P.meanFitness}{P.bestFitness - P.worstFitness}$ 
    Else if  $P.bestFitness = P.meanFitness$  then
        Adjustment  $\theta = 0.01$ 
    End if
    If  $\overline{CP} > \overline{MP}$  then
        Crossover Probability  $C = C + \theta$ 
        Mutation Probability  $M = M - \theta$ 
    Else if  $\overline{CP} < \overline{MP}$  then
        Crossover Probability  $C = C - \theta$ 
        Mutation Probability  $M = M + \theta$ 
    End if
    Clamp C to range [0.001, 1.0]
    Clamp M to range [0.001, 1.0]
END

```

FIGURE 3.11: Here you see the self-adaptation algorithm.

finely tuned search as the population converges to the optimum in the fitness landscape.

3.4.4 Experiment four: static crossover and mutation probabilities with the crossover and mutation operators working in parallel.

Experiment four had an identical setup to experiment three with the exception that the crossover and mutation probabilities were statically set, throughout the experiment, to the crossover and mutation probabilities arrived at after the 100th generation of the genetic algorithm in experiment three. The genetic algorithm parameters used are listed in Table 3.5.

Experiment four as well as experiment five were devised as comparisons to experiment three. The hypothesis was that self-adaptation, along with non-interfering operators, would have the fastest average fitness growth rate among the three.

3.4.5 Experiment five: static crossover and mutation probabilities with the crossover and mutation operators working in sequence.

Experiment five had an identical setup as experiment four with the exception that the crossover and mutation operators were run in sequence instead of in parallel. By running the operators in sequence, the mutation operator could disrupt the offspring created by the crossover operator. The genetic algorithm parameters used are listed in Table 3.6.

3.4.6 Experiment Six: fitness and ball-in-play upper bound.

The setup for experiment six was identical to that of experiment two, three, four, and five with the exception that no evolution ever took place (crossover and mutation probabilities were set to 0.0) during the run of the genetic algorithm over 100 generations. Only the simplified fitness function (described in experiment two) was used during the run of the genetic algorithm. With no evolution taking place, the number-of-elite-offspring parameter was set to 10—equal to the population size—in order to use the same genetic algorithm code base that was implemented and employed for the previous experiments. The genetic algorithm parameters used are listed in Table 3.7.

The goal of experiment six was to obtain the average fitness over 100 generations, the fitness of every 10th generation top performer, and the average ball-in-play time of five rounds per every 10th generation top performer **where the paddle always performed the correct movement** no matter the state of the

paddle and the ball at any time during any round. In other words, the goal of experiment six was to obtain an average fitness upper bound and an average ball-in-play time of five rounds upper bound utilizing the same fitness function as was used in experiment two, three, four, and five.

To accomplish the goal of experiment six, the neural-network output was ignored and instead, the paddle's center height (y-coordinate) was always set to the ball's center y-coordinate once every draw loop. With this modification, the paddle was always dead center with the ball, was always in the path of the ball, and thus always kept the ball from leaving the left side of the arena. That is, it always hit the ball back into the arena. At no point was the paddle ever not in the path of the ball and thus the paddle always obtained the maximum fitness possible as well as the maximum ball-in-play time possible for any particular round. The only way any round ever terminated was by the ball's velocity magnitude falling below the 100 threshold.

3.4.7 Experiment seven: random paddles.

The setup for experiment seven was identical to that of experiment six where no evolution ever took place (crossover and mutation probabilities were set to 0.0) during the run of the genetic algorithm over 100 generations. Only the simplified fitness function (described in experiment two) was used during the run of the genetic algorithm. With no evolution taking place, the number-of-elite-offspring parameter was set to 10—equal to the population size—in order to use the same genetic algorithm code base that was implemented and employed for the previous experiments. The genetic algorithm parameters used are listed in Table 3.8.

The goal of experiment seven was to obtain the average fitness over 100 generations, the fitness of every 10th generation top performer, and the average ball-in-play time of five rounds per every 10th generation top performer **where the paddle always performed a random movement**. By obtaining these metrics for randomly moving paddles, it could be demonstrated that the genetic algorithm either did or did not ultimately produce paddles that performed better than the randomly moving paddles.

To accomplish the goal of experiment seven, the neural-network output was ignored and instead, the paddle's movement was always randomly generated once per draw loop. To generate the random movement, a random float was sampled from a uniform distribution $X \sim U(-1, 1)$. If X was in the range $[-1, 0]$, the paddle's velocity angle was set to 270° and the paddle's velocity magnitude was set to $m_{new} = -1*X*m_{initial}$

where $m_{initial}$ was the paddle's starting velocity magnitude (specifically 1000 $\frac{\text{pixels}}{\text{second}}$). If X was in the range $(0, 1]$, the paddle's velocity angle was set to 90° and the paddle's velocity magnitude was set to $m_{new} = X * m_{initial}$. Otherwise, if X was zero, the paddle did not move from its current position.

3.5 Experimental Results

The results of each experiment are presented below (Sections 5.1 through 5.7), followed by a comparison of all results (Section 5.8).

For each experiment (one through seven), there are three plots shown. The first plot shows the average fitness over 100 generations. The second plot shows the fitness of the top performer for every 10th generation (over 100 generations). The third plot shows the ball-in-play time, averaged over 5 games, for the top performers whose fitness is illustrated in the second plots.

3.5.1 Experiment one: use of a full and partial credit granting fitness function.

Experiment one (Figures 3.12-3.14) show that the player learns to keep the ball in play longer after generation 49, improving from 3.00584 seconds per round on average to 28.30964 seconds. This is despite the fact that the average fitness doesn't begin to show significant improvement until generation 80. Indeed, it is strange that the improvement in average ball-in-play time drops in generation 79, but then improves and exceeds the previous maximum (generation 69) after generation 89.

3.5.2 Experiment two: use of a simplified fitness function, use of rank fitness in selection, higher crossover probability, mutation probability based on the number of genes per genome, and a Gaussian distribution sample mutation step.

Experiment two (Figures 3.15-3.17) shows marked improvement in average fitness to generation 20, and then a more gradual improvement for the remaining 80 generations. The average ball-in-play times improve dramatically from 10.1 seconds to 37.8 seconds within the first 20 generations, but levels off and does not show marked improvement for the remainder of the experiment.

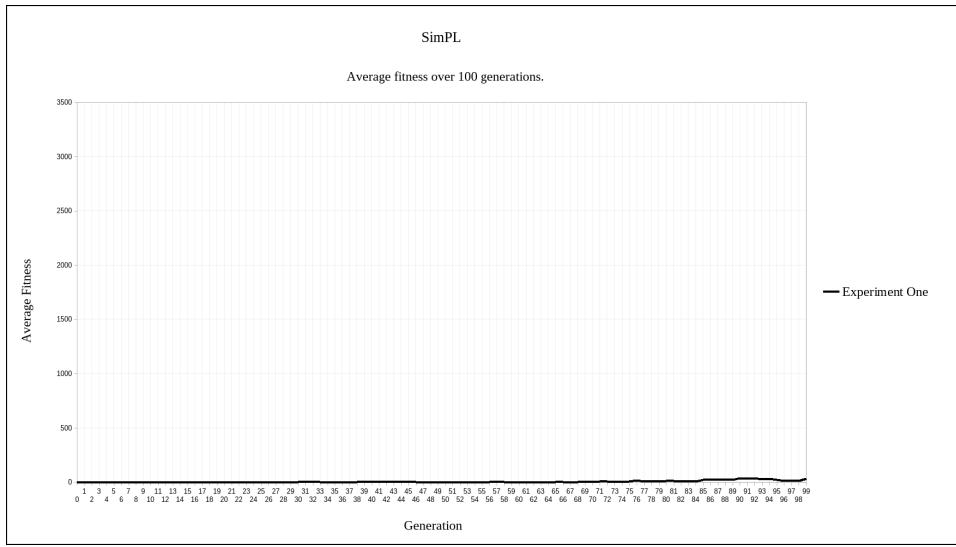


FIGURE 3.12: Here you see the average fitness over 100 generations for experiment one.

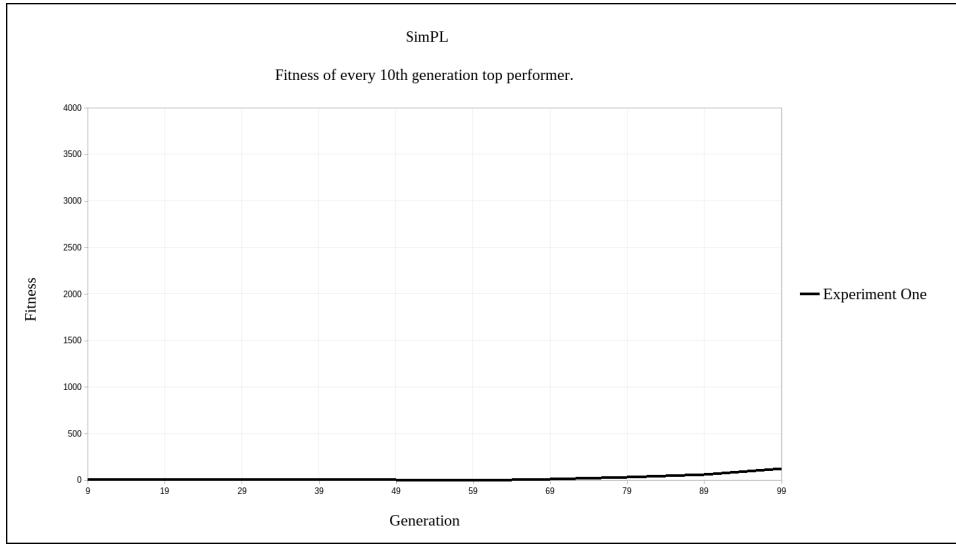


FIGURE 3.13: Here you see the fitness of every 10th generation top performer for experiment one.

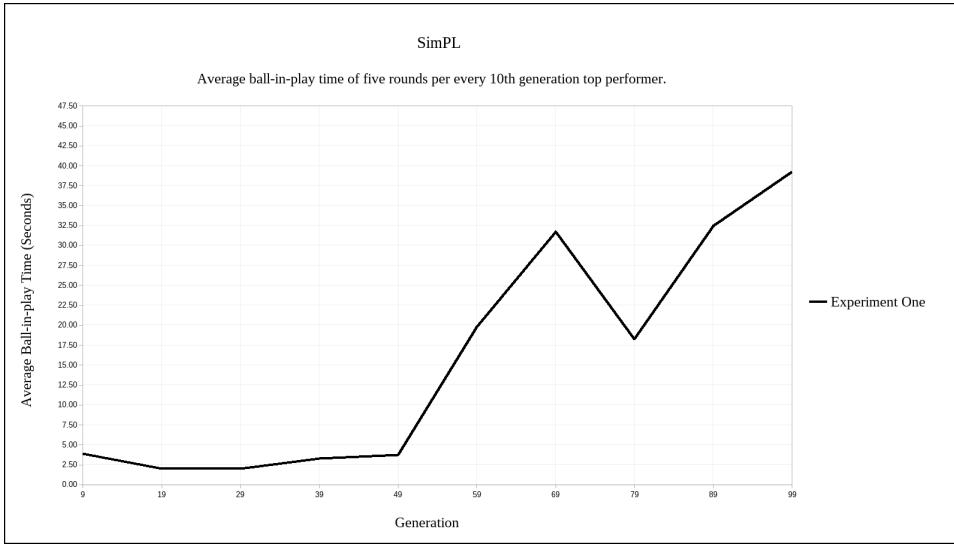


FIGURE 3.14: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment one.

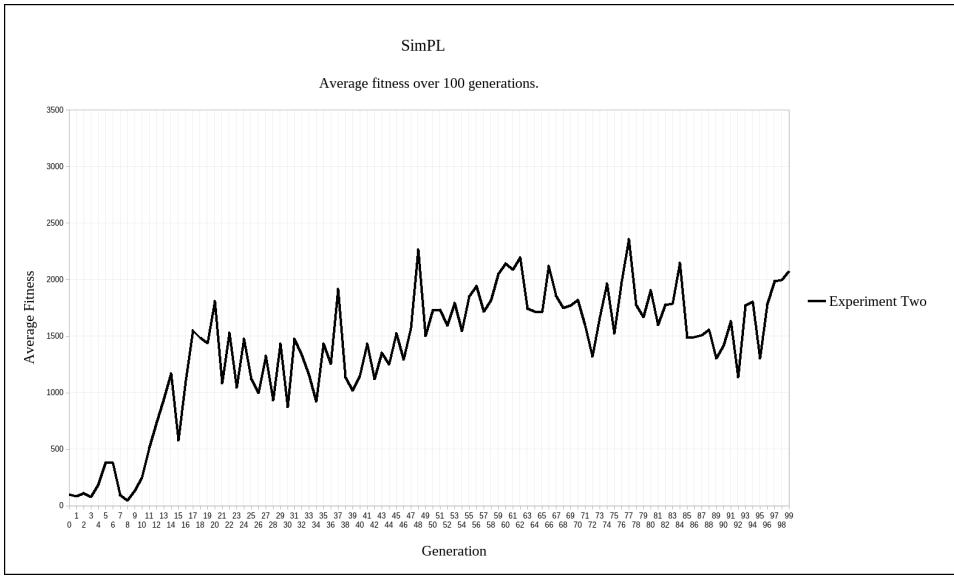


FIGURE 3.15: Here you see the average fitness over 100 generations for experiment two.

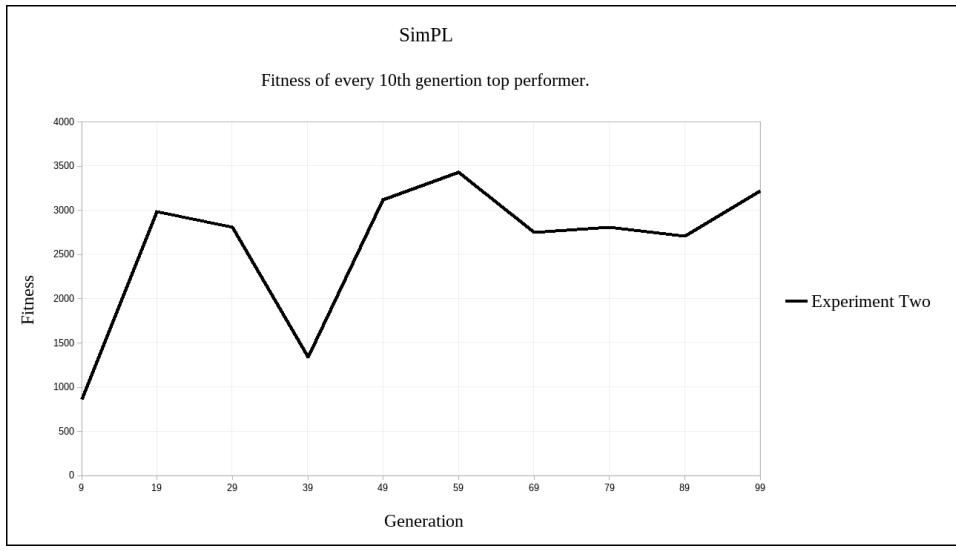


FIGURE 3.16: Here you see the fitness of every 10th generation top performer for experiment two.

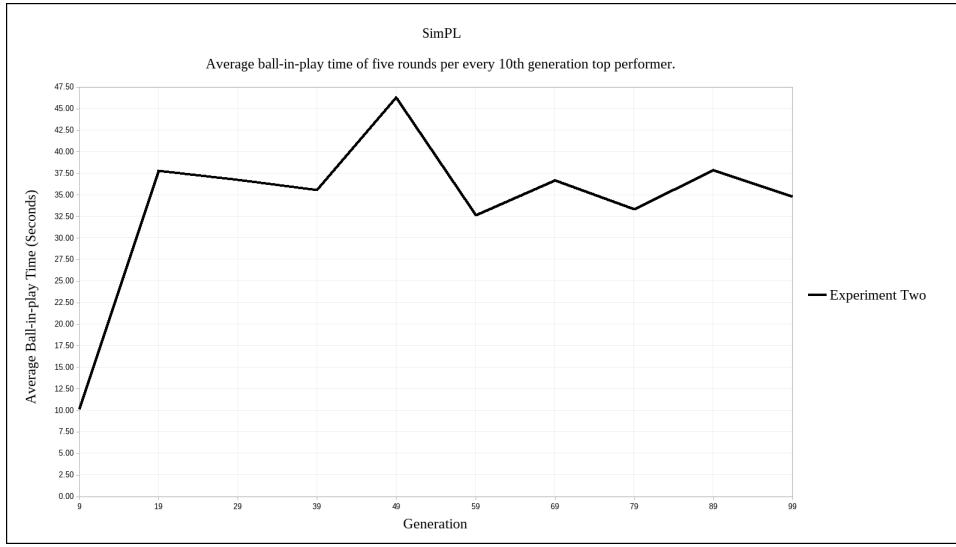


FIGURE 3.17: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment two.

3.5.3 Experiment three: self-adaptation of crossover and mutation probabilities with the crossover and mutation operators working in parallel.

Experiment three (Figures 3.18-3.21) shows initial improvement in average fitness for the first 12 generations, but then oscillates and does not show measurable improvement for the rest of the experiment. This result is reflected in the average ball-in-play time metric, which starts at 35.7314 seconds and oscillates but does not trend significantly. Figure 3.19 illustrates the self-adaptation of the crossover and mutation probabilities, as chosen by the algorithm at run-time. Although the proportions change, they do not appear to correlate with the fitness or average ball-in-play time results.

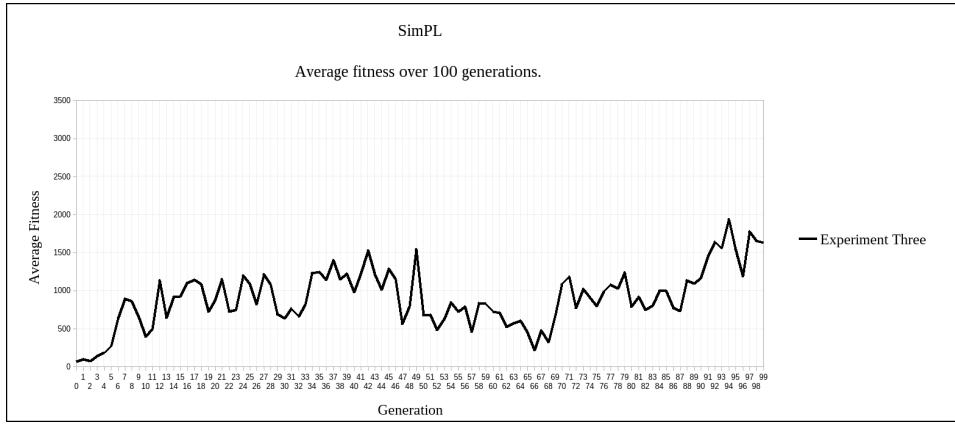


FIGURE 3.18: Here you see the average fitness over 100 generations for experiment three.

The crossover and mutation probabilities were both initially set to 0.5 before the start of the experiment. After the experiment was over, the genetic algorithm self-adapted the crossover probability to 0.7816 and self-adapted the mutation probability to 0.2184. You'll notice in Figure 3.19 that the mutation probability overtook the crossover probability at first but the two probabilities eventually diverged with mutation becoming less probable and crossover becoming more probable as the genetic algorithm produced fitter generations. This outcome is almost the exact opposite of the outcome shown in [10].

3.5.4 Experiment four: static crossover and mutation probabilities with the crossover and mutation operators working in parallel.

Experiment four (Figures 3.22-3.24) produces similar results, with initial improvement in average fitness for the first 10 generations, but oscillation thereafter. However, the mean of the average ball-in-play times was

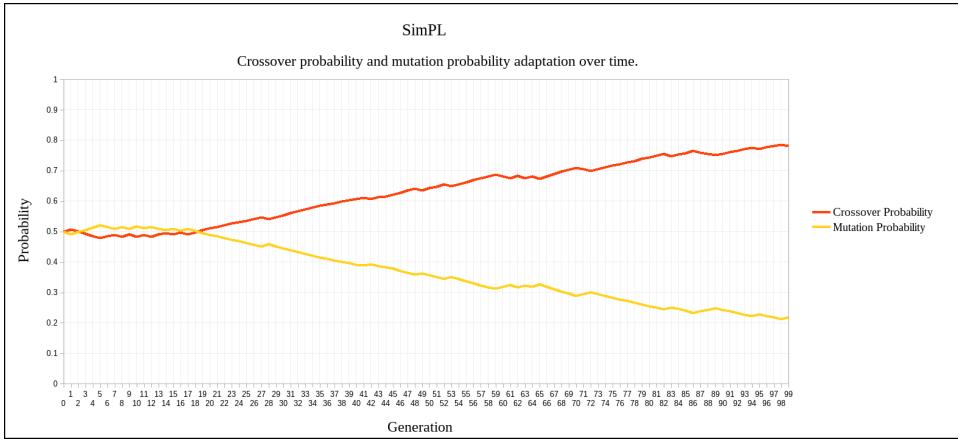


FIGURE 3.19: Here you see the self-adaptation of the crossover and mutation probabilities for experiment three.

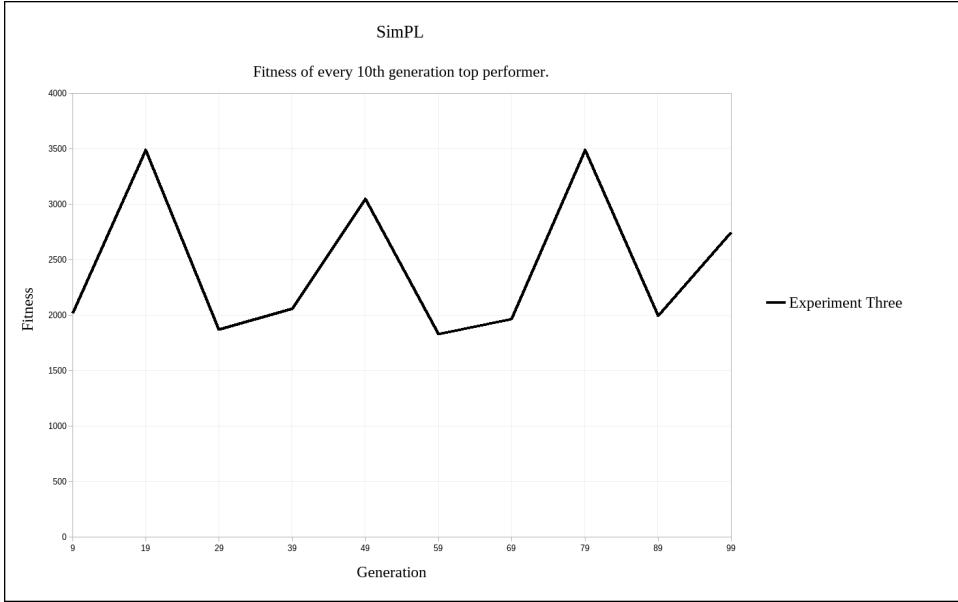


FIGURE 3.20: Here you see the fitness of every 10th generation top performer for experiment three.

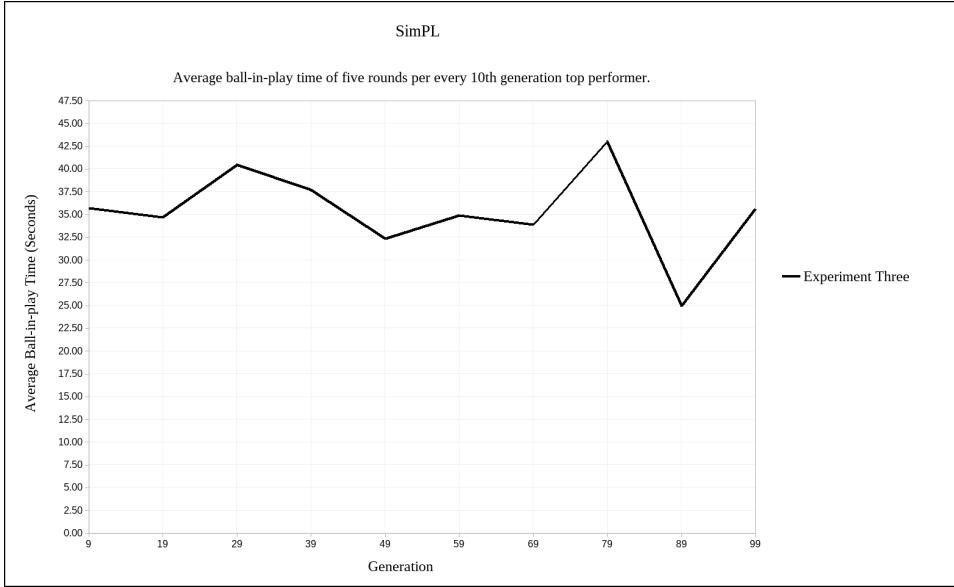


FIGURE 3.21: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment three.

37.28646 seconds, over the mean 35.33994 seconds shown for the player learned in Experiment three and the mean 34.18604 seconds in Experiment two.

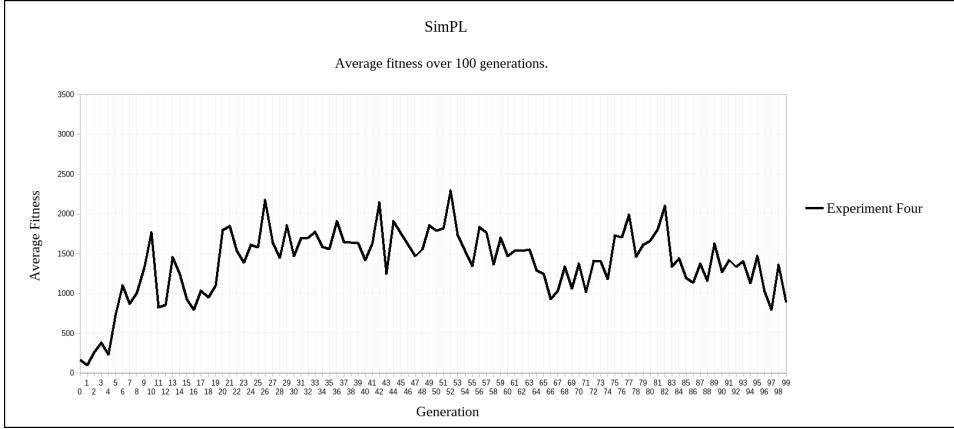


FIGURE 3.22: Here you see the average fitness over 100 generations for experiment four.

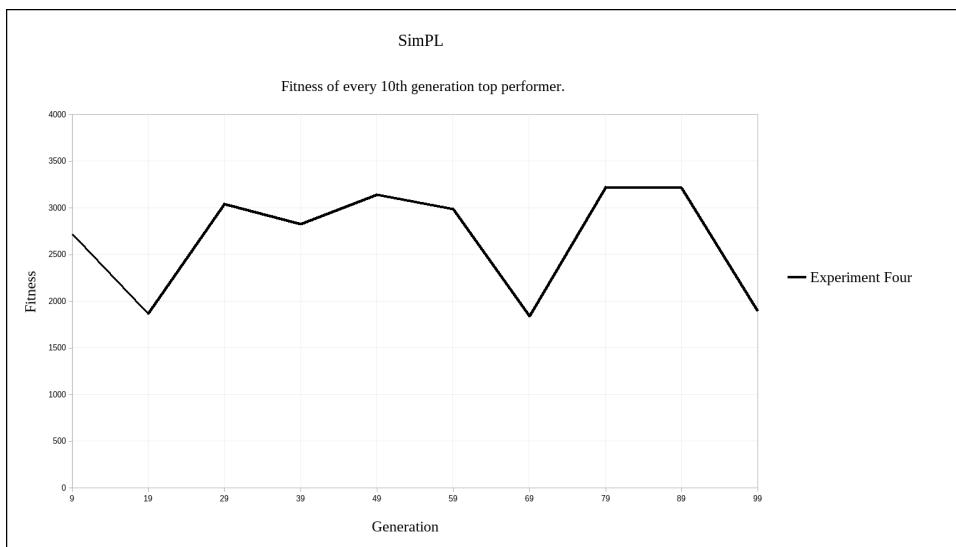


FIGURE 3.23: Here you see the fitness of every 10th generation top performer for experiment four.

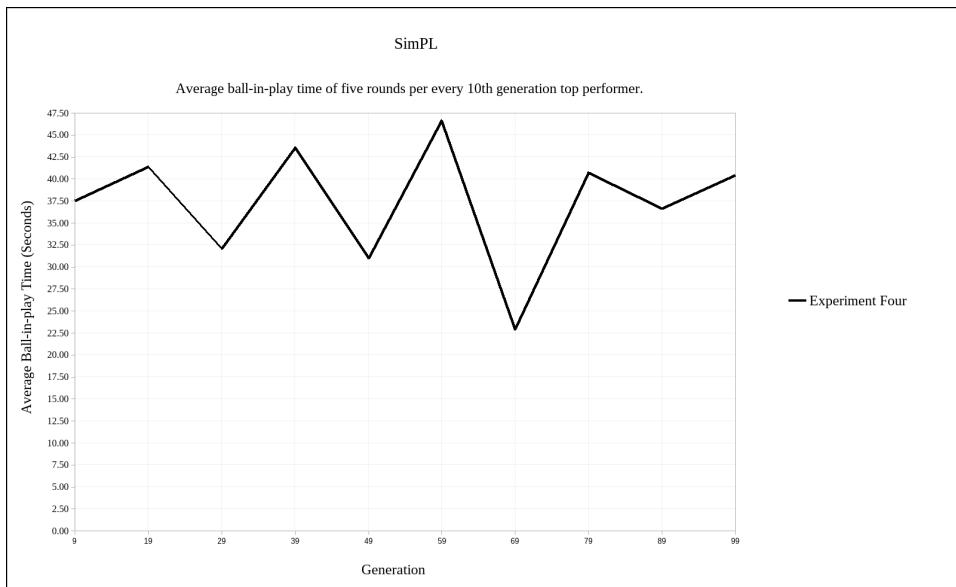


FIGURE 3.24: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment four.

3.5.5 Experiment five: static crossover and mutation probabilities with the crossover and mutation operators working in sequence.

Experiment five (Figures 3.25-3.27) shows more gradual improvement in average fitness over the first 31 generations. Overall, the mean of the average ball-in-play times was 37.91126 seconds—higher than that of Experiment four.

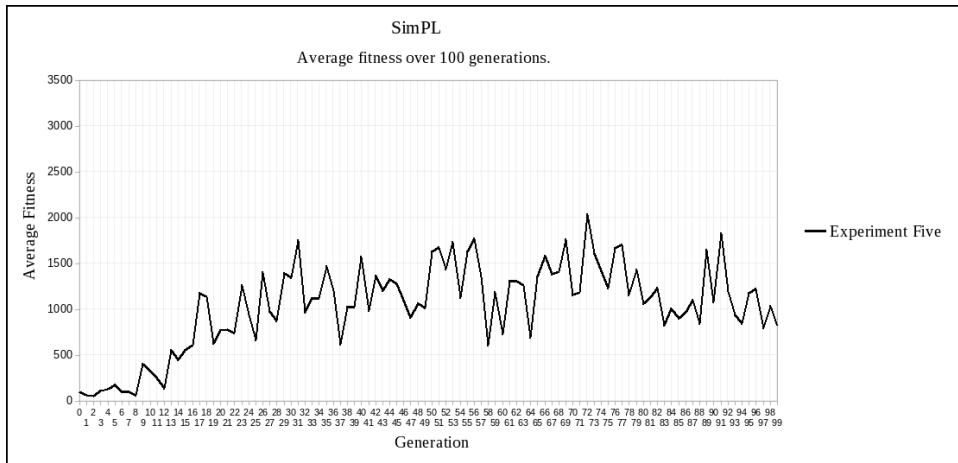


FIGURE 3.25: Here you see the average fitness over 100 generations for experiment five.

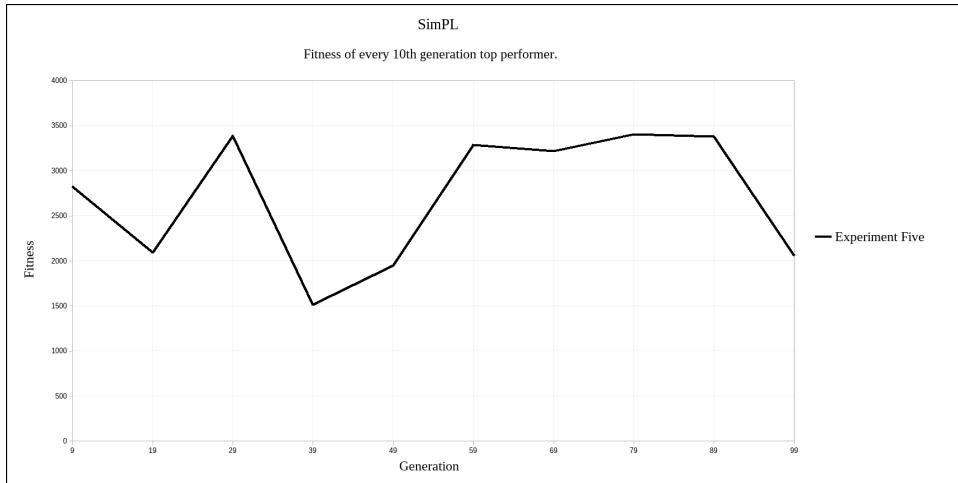


FIGURE 3.26: Here you see the fitness of every 10th generation top performer for experiment five.

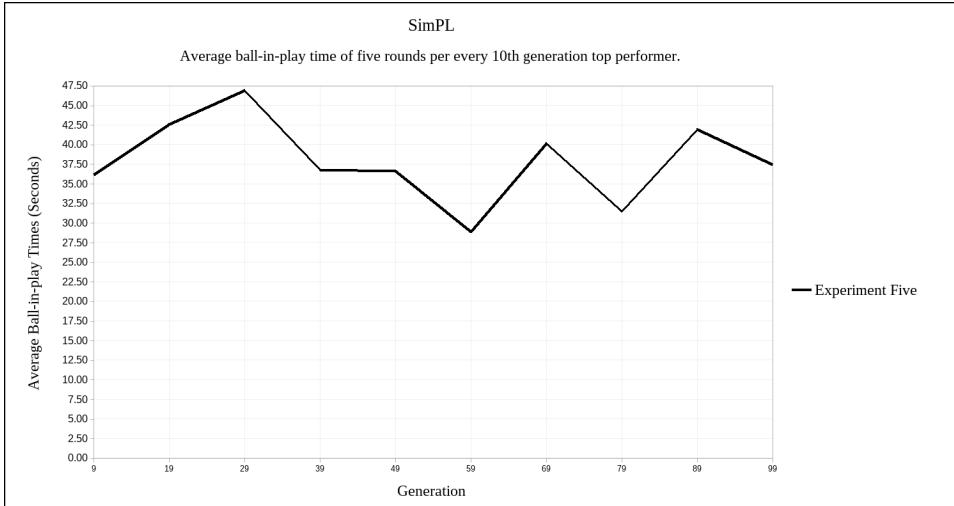


FIGURE 3.27: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment five.

3.5.6 Experiment six: fitness and ball-in-play upper bound.

Experiment six (Figures 3.28-3.30) illustrates the *perfect* results possible of a player who always behaves correctly. These metrics demonstrate optimal values, which can be considered targets for the evolution experiments. The mean of the average ball-in-play times was 40.01528 seconds, which means that Experiment five (above) comes closest to optimal performance.

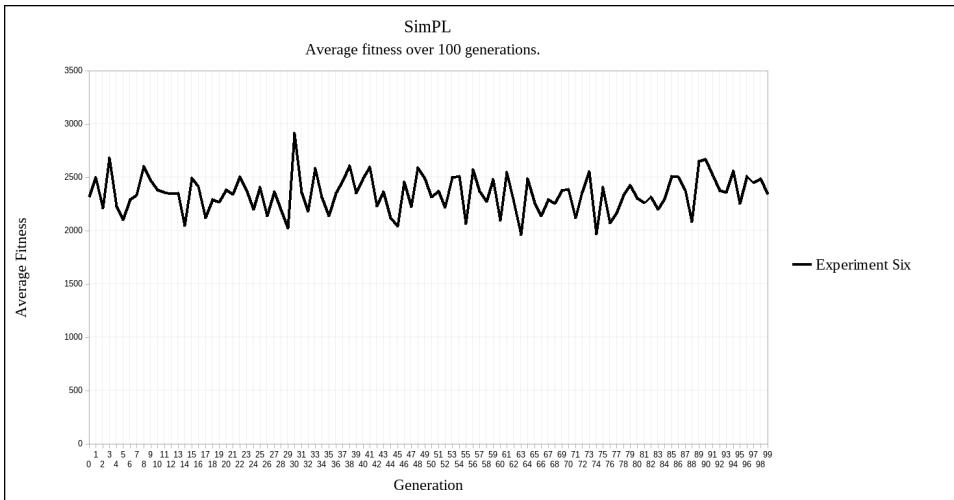


FIGURE 3.28: Here you see the average fitness over 100 generations for experiment six.

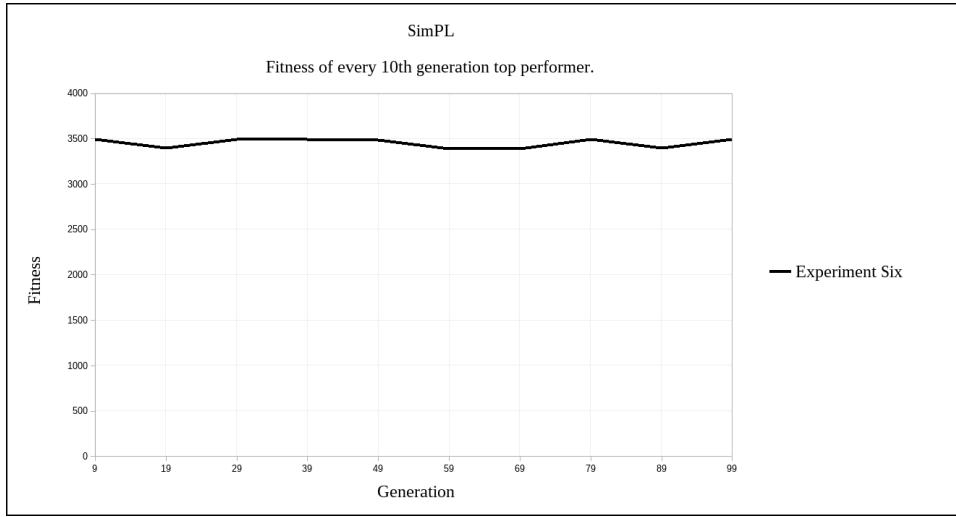


FIGURE 3.29: Here you see the fitness of every 10th generation top performer for experiment six.

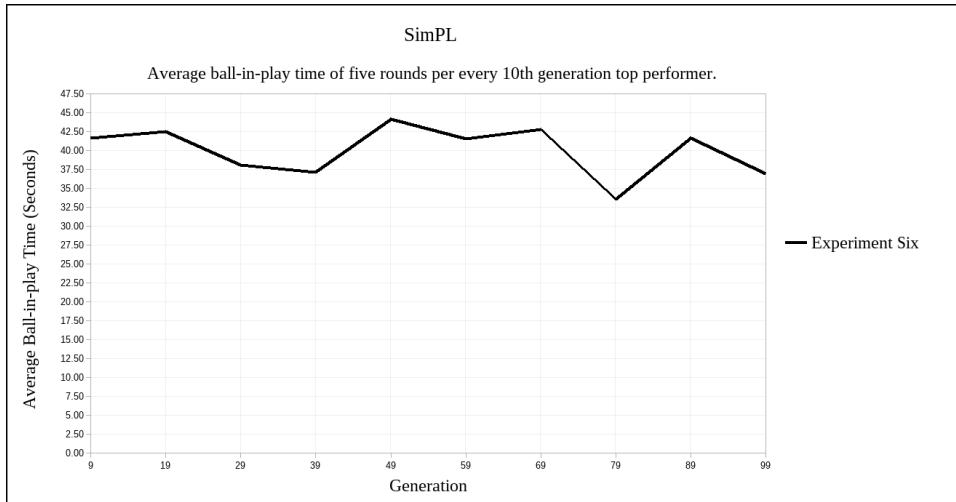


FIGURE 3.30: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment six.

3.5.7 Experiment seven: random paddles.

Experiment seven (Figures 3.31-3.33) illustrates the results of a player who behaves randomly. This provides an example of worst case metrics. The mean of the average average ball-in-play times was 5.2656 seconds, which is comparable to the first-generation games of the evolved players.

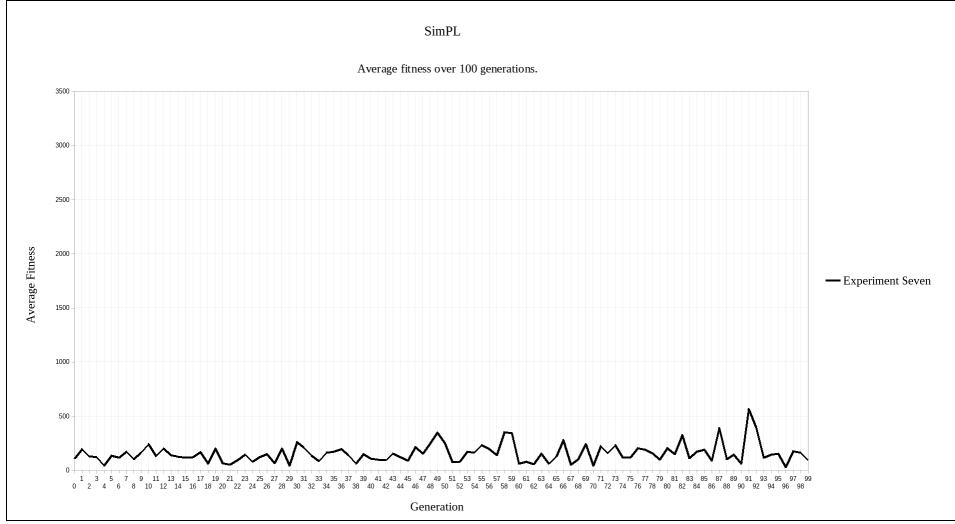


FIGURE 3.31: Here you see the average fitness over 100 generations for experiment seven.

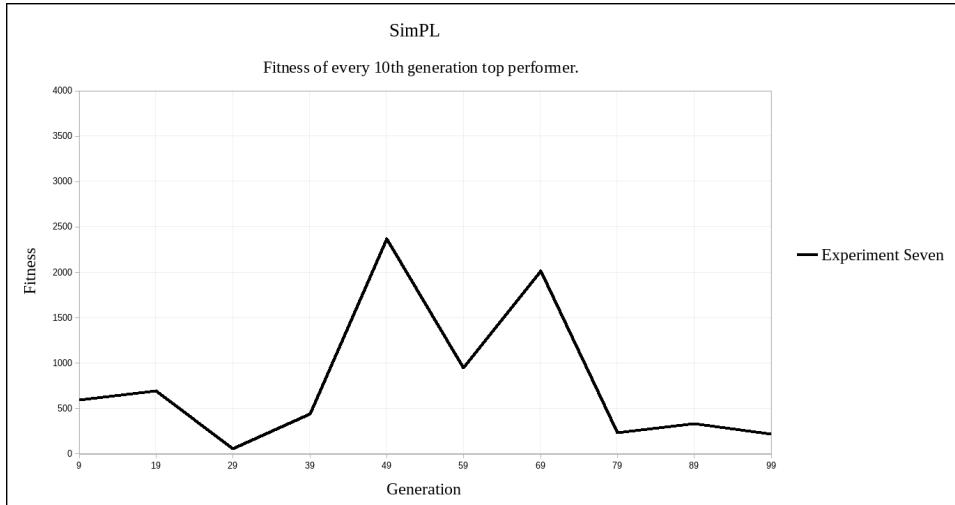


FIGURE 3.32: Here you see the fitness of every 10th generation top performer for experiment seven.

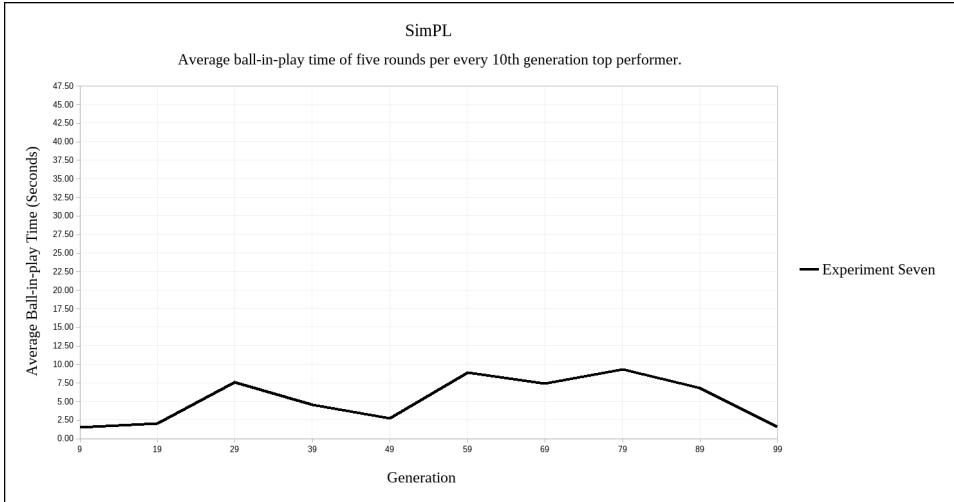


FIGURE 3.33: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment seven.

3.5.8 Comparative Results

This section illustrates comparative results by plotting all curves on the same axes (Figures 3.34 through 3.36).

3.5.8.1 Average fitness over 100 generations.

See Figure 3.34.

3.5.8.2 Fitness of every 10th generation top performer.

See Figure 3.35.

3.5.8.3 Average ball-in-play time (in seconds) of five rounds per every 10th generation top performer.

See Figure 3.36.

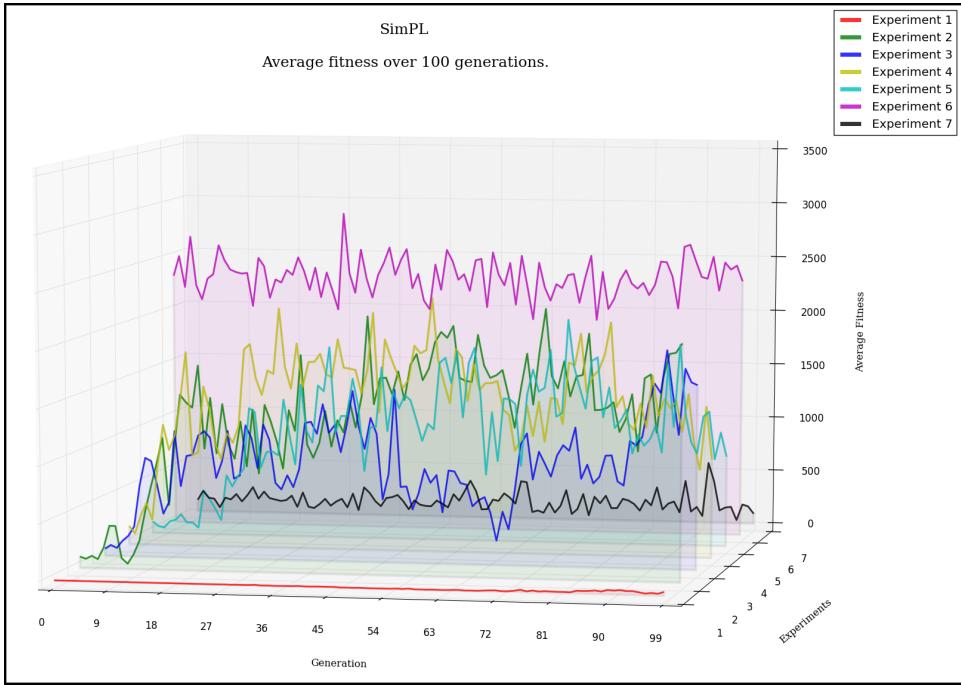


FIGURE 3.34: Here you see the average fitness over 100 generations for experiments one through seven.

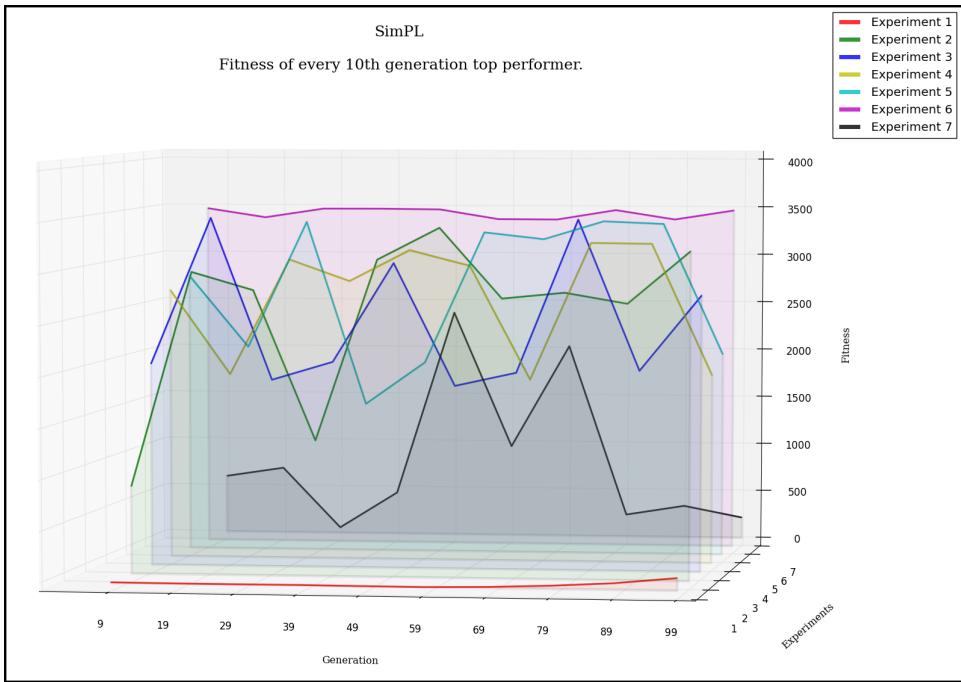


FIGURE 3.35: Here you see the fitness of every 10th generation top performer for experiments one through seven.

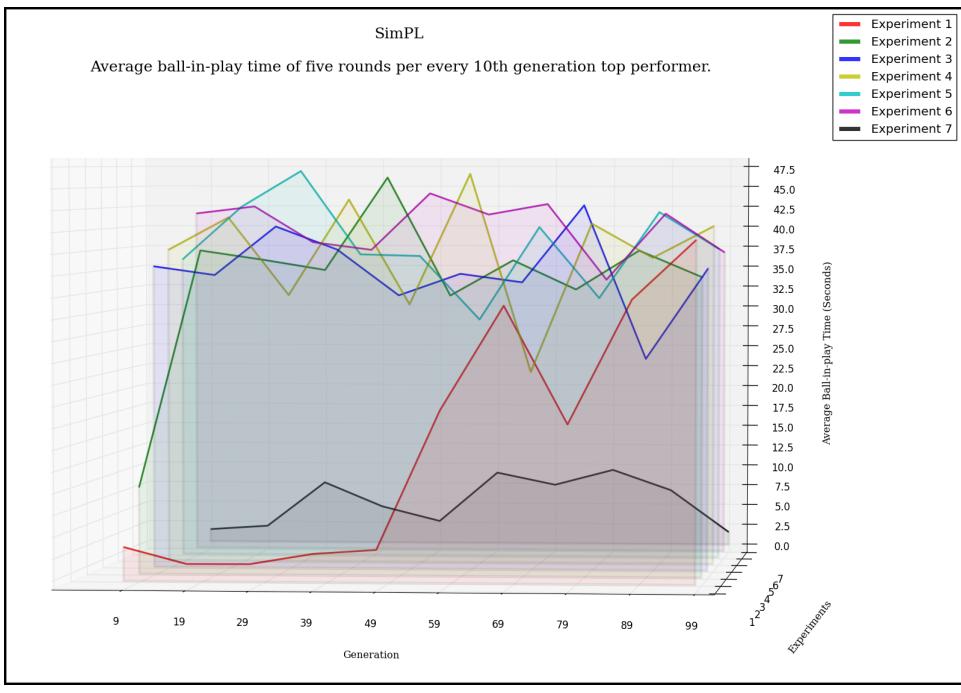


FIGURE 3.36: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiments one through seven.

Population Size	10
Number of Elite Offspring	2
Roulette Selection using Actual Fitness	True
Roulette Selection using Rank Fitness	False
Crossover Probability	0.7
Crossover Type	One Point Crossover
Mutation Probability	0.1
Mutation Scope	Gene Level
Mutation Step	$X \sim U[-1, 1] * \text{Max-Perturbation}$
Max-Perturbation	0.3
Sequential Crossover and Mutation	True
Fitness Function	Partial/Full
Fitness Credit for Hitting the Ball	1.0
Fitness Credit for Following the Ball	0.1
Fitness Credit for Matching the Ball's Center Y-coordinate	0.1
Fitness Credit for not Following the Ball	-0.1

TABLE 3.2: Here you see the genetic algorithm parameters for experiment one.

Population Size	10
Number of Elite Offspring	2
Roulette Selection using Actual Fitness	False
Roulette Selection using Rank Fitness	True
Crossover Probability	0.8
Crossover Type	One Point Crossover
Mutation Probability	$\frac{1}{n \text{ genes}} = \frac{1}{41} = 0.024390244$
Mutation Scope	Gene Level
Mutation Step	$X \sim N(\mu, \sigma^2)$
Mutation Step μ	Gene Value
Mutation Step σ	0.5
Max Perturbation	N/A
Sequential Crossover and Mutation	True
Fitness Function	Full
Fitness Credit for Staying in the Path of the Ball	1.0
Fitness Credit for not Staying in the Path of the Ball	0.0

TABLE 3.3: Here you see the genetic algorithm parameters for experiment two.

Population Size	10
Number of Elite Offspring	2
Roulette Selection using Actual Fitness	False
Roulette Selection using Rank Fitness	True
Self-adaptation	True
Initial Crossover Probability	0.5
Minimum Crossover Probability	0.001
Crossover Type	One Point Crossover
Initial Mutation Probability	0.5
Minimum Mutation Probability	0.001
Mutation Scope	Genome Level
Mutation Step	$X \sim N(\mu, \sigma^2)$
Mutation Step μ	Gene Value
Mutation Step σ	Mutation Probability
Sequential Crossover and Mutation	False
Fitness Function	Full
Fitness Credit for Staying in the Path of the Ball	1.0
Fitness Credit for not Staying in the Path of the Ball	0.0

TABLE 3.4: Here you see the genetic algorithm parameters for experiment three.

Population Size	10
Number of Elite Offspring	2
Roulette Selection using Actual Fitness	False
Roulette Selection using Rank Fitness	True
Self-adaptation	False
Crossover Probability	0.7816
Crossover Type	One Point Crossover
Mutation Probability	0.2184
Mutation Scope	Genome Level
Mutation Step	$X \sim N(\mu, \sigma^2)$
Mutation Step μ	Gene Value
Mutation Step σ	Mutation Probability
Sequential Crossover and Mutation	False
Fitness Function	Full
Fitness Credit for Staying in the Path of the Ball	1.0
Fitness Credit for not Staying in the Path of the Ball	0.0

TABLE 3.5: Here you see the genetic algorithm parameters for experiment four.

Population Size	10
Number of Elite Offspring	2
Roulette Selection using Actual Fitness	False
Roulette Selection using Rank Fitness	True
Self-adaptation	False
Crossover Probability	0.7816
Crossover Type	One Point Crossover
Mutation Probability	0.2184
Mutation Scope	Gene Level
Mutation Step	$X \sim N(\mu, \sigma^2)$
Mutation Step μ	Gene Value
Mutation Step σ	Mutation Probability
Sequential Crossover and Mutation	True
Fitness Function	Full
Fitness Credit for Staying in the Path of the Ball	1.0
Fitness Credit for not Staying in the Path of the Ball	0.0

TABLE 3.6: Here you see the genetic algorithm parameters for experiment five.

Population Size	10
Number of Elite Offspring	10
Roulette Selection using Actual Fitness	N/A
Roulette Selection using Rank Fitness	N/A
Self-adaptation	N/A
Crossover Probability	N/A
Crossover Type	N/A
Mutation Probability	N/A
Mutation Scope	N/A
Mutation Step	N/A
Sequential Crossover and Mutation	N/A
Fitness Function	Full
Fitness Credit for Staying in the Path of the Ball	1.0
Fitness Credit for not Staying in the Path of the Ball	0.0

TABLE 3.7: Here you see the genetic algorithm parameters for experiment six.

Population Size	10
Number of Elite Offspring	10
Roulette Selection using Actual Fitness	N/A
Roulette Selection using Rank Fitness	N/A
Self-adaptation	N/A
Crossover Probability	N/A
Crossover Type	N/A
Mutation Probability	N/A
Mutation Scope	N/A
Mutation Step	N/A
Sequential Crossover and Mutation	N/A
Fitness Function	Full
Fitness Credit for Staying in the Path of the Ball	1.0
Fitness Credit for not Staying in the Path of the Ball	0.0

TABLE 3.8: Here you see the genetic algorithm parameters for experiment seven.

Chapter 4

BBAutoTune

4.1 Overview

The purpose of BBAutoTune is to find the correct combination of physics parameters such that the motion of the simulated robot is indistinguishable from the real world counterpart. Components of BBAutoTune include a genetic algorithm, database manager, GUI panel, and an external progress monitor which tracks various metrics of the running genetic algorithm.

4.2 Implementation

Most of the components to BBAutoTune run inside of Blender itself with the exception of the external GA monitor. Blender's API uses the Python programming language and thus BBAutoTune was written entirely in Python.

4.2.1 Surveyor SRV-1 Blackfin 3D Model

The robot used during experimentation was a 3D model of the Surveyor SRV-1 Blackfin. This model is dimensionally and aesthetically based on the real robot. The extents of the model are $17.64cm \times 14.54cm \times 14.33cm$ including the braille hat that rests above the body of the robot. The base and the wheels are the only physics based objects on the model with the wheels being connected to the base via a rigid body hinge joint. See Figure 5.5.

4.2.2 GUI

Blender's graphical user interface can be extended via its Python API. BBAutoTune adds a custom panel to Blender that allows the user to specify certain parameters pertaining to the GA and the overall tuning loop. Once the user specifies their preferred parameters, they press start to begin the tuning process. After pressing start, BBAutoTune will continue to run until the GA has reached the max generations specified.

See Figure 4.1.

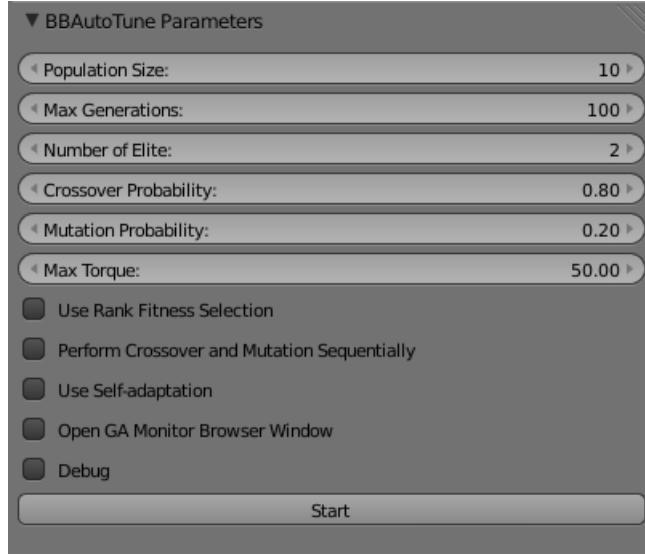


FIGURE 4.1: Here you see the BBAutoTune GUI panel added to Blender.

4.2.3 Physics Engine API

Blender exposes the Bullet¹ physics engine API via its own Python API. Most if not all of the parameters to Bullet can be modified via the physics engine API during or before running the Blender game engine. There are 41 different physics engine parameters that can be set via the API.

4.2.3.1 Parameter Set and Ranges

See Table 4.1.

¹Bullet is the real time physics engine used by Blender. Released under the zlib license, Bullet provides real time continuous collision detection and rigid body dynamics [web].

Parameter	Range	Default Value
Gravity	[0.0 $\frac{m}{s^2}$,10000.0 $\frac{m}{s^2}$]	9.8 $\frac{m}{s^2}$
Mass	[0.0,10000.0]	1.0
Force	[-inf,inf]	0.0
Torque	[-inf,inf]	0.0
Linear Velocity	[-inf,inf]	0.0
Angular Velocity	[-inf,inf]	0.0
Apply Force Locally	[False,True]	True
Apply Torque Locally	[False,True]	True
Apply Linear Velocity Locally	[False,True]	True
Apply Angular Velocity Locally	[False,True]	True
Use Material Physics	[False,True]	True
Material Friction	[0.0,100.0]	0.5
Material Elasticity	[0.0,1.0]	0.0
Material Force	[0.0,1.0]	0.0
Material Damping	[0.0,1.0]	0.0
Material Distance	[0.0,20.0]	0.0
Material Align to Normal	[False,True]	False
Actor	[False,True]	True
Ghost	[False,True]	False
Physics Type	[NO_COLLISION, STATIC, DYNAMIC, RIGID_BODY, SOFT_BODY, OCCLUDE, SENSOR, NAVMESH, CHARACTER]	STATIC
Use Material Force Field	[False,True]	False
Rotate From Normal	[False,True]	False
No Sleeping	[False,True]	False
From Factor	[0.0,1.0]	0.4
Use Anisotropic Friction	[False,True]	False
Anisotropic Friction XYZ	[0.0,1.0]	1.0
Velocity Min/Max	[0.0,1000.0]	0.0
Lock Translation XYZ	[False,True]	False
Lock Rotation XYZ	[False,True]	False
Damping Translation	[0.0,1.0]	0.025
Damping Rotation	[0.0,1.0]	0.159
Use Collision Bounds	[False,True]	False
Collision Radius	[0.01m,inf]	1m
Collision Bound Type	[BOX, SPHERE, CYLINDER, CONE, CONVEX_HULL, TRIANGLE_MESH, CAPSULE]	BOX
Collision Margin	[0.0m,1.0m]	6cm
Max Physics Steps	[1,5]	5
Physics Sub-steps	[1,50]	1
FPS	[1,10000]	60
Linear Deactivation Threshold	[0.001,10000.0]	0.8
Angular Deactivation Threshold	[0.001,10000.0]	1.0
Deactivation Time	[0.0s,60.0s]	2.0s

TABLE 4.1: Here you see the various physics parameters, their ranges, and their default values. Note that inf=340,282,346,638,528,859,811,704,183,484,516,925,440.0 in Blender.

4.2.4 Database Manager

Running within Blender, the database manager opens a connection to a local MySQL database. For each evaluated genetic algorithm generation, the database manager stores the generation number, the highest fitness, the average fitness, the lowest fitness, the crossover probability, and the mutation probability in the database.

4.2.5 Robot Monitor

Running within the Blender game engine, the robot monitor records the position and rotation state of the simulated robot's base throughout the duration of running the game engine. At the very start of the game engine, the robot monitor records the position and rotation state S of the robot's base with a time stamp T . After one second has passed, the robot monitor records the position and rotation state S' of the robot's base with a time stamp T' . At this point, if the robot's base has come to a rest, the robot monitor exits the game engine. Otherwise, if the robot's base is still moving, the robot monitor will update S' and T' every half second for the rest of the evaluation period. After 16 seconds have elapsed, the robot monitor exits the game engine regardless of whether or not the robot's base is still moving. Every time the robot monitor records S' and T' , it writes S , S' , T , and T' to a file that will be later read by the fitness function.

4.2.6 Progress Monitor

The progress monitor is an external and self-contained Python HTTP-CGI server that listens on port 8000. By visiting `http://localhost:8000/index.py`, a user can track the GA's progress concerning the highest fitness, average fitness, lowest fitness, crossover probability, and the mutation probability. See Figure 4.2. Once the user presses start on the GUI panel, BBAutoTune starts the server as an external process with the option of opening a browser to the progress page. Once every minute, the progress monitor retrieves the most current GA run information from the local MySQL database which was populated by the database manager.

4.2.7 Genetic Algorithm

The genetic algorithm for BBAutoTune was borrowed from the SimPL project and ported to Python with some aspects of the GA being altered to suit the needs of BBAutoTune. Since lower values of fitness are

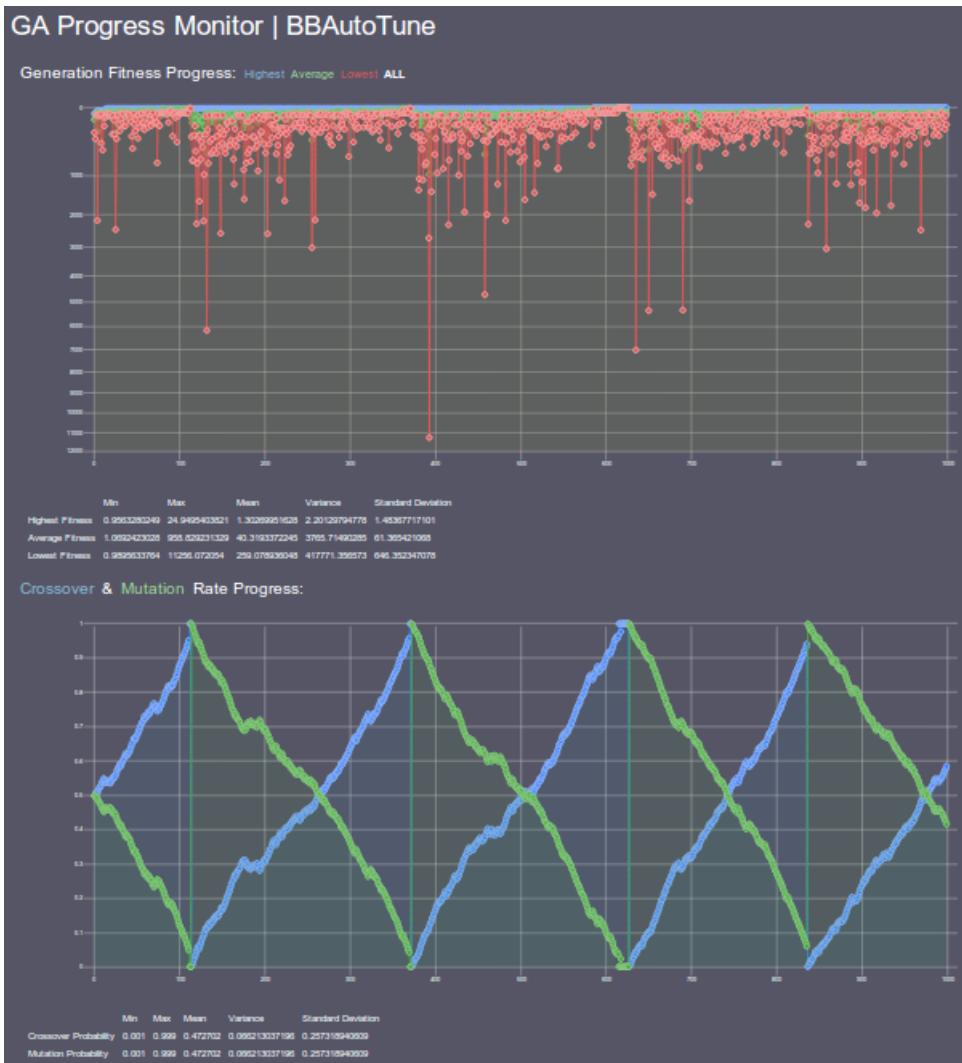


FIGURE 4.2: Here you see external GA progress monitor.

considered higher than those of higher values of fitness, the population sorting function needed to be altered. Other portions altered were the selection operator, the population metrics calculator, and the self-adaptation algorithm.

4.2.7.1 Encoding Scheme

The physics parameters selected for tuning were a mixed set of floats, integers, string arrays, and boolean values. To ease the process of crossover and mutation, all genome genes were homogenized to a normalized range of [0.0, 1.0]. Let any given genome's gene value be g_i and any given physics parameter be p_i . For a float type with a maximum range value r_{max} and minimum range value r_{min} , the mapping function was $p_i = (g_i * (r_{max} - r_{min})) + r_{min}$. For an integer type with a maximum range value r_{max} and a minimum range value r_{min} , the mapping function was $p_i = \lfloor (g_i * (r_{max} - r_{min})) + r_{min} \rfloor$. For an array A of strings type with size n , the mapping function was $p_i = A[\lfloor g_i * (n - 1) \rfloor]$. Finally for a boolean type, the mapping function was

$$p_i = \begin{cases} \text{True} & \text{if } g_i \geq 0.5, \\ \text{False} & \text{if } g_i < 0.5. \end{cases}$$

4.2.7.2 Operators

The operators used include selection, elitism, crossover, and mutation. All of these operators work together to generate a new population once the current population has been fully evaluated by the fitness function.

The selection operator includes two variants: tournament selection and rank fitness selection. The user can indicate on the GUI panel if rank fitness selection is to be used—otherwise tournament selection will be used. Tournament selection works by gathering a sub-portion of the total population where the fittest genome among the sub-portion is selected thereby winning the tournament [13]. While gathering the sub-portion, all genomes in the population have a uniform probability of being included in the tournament regardless of their respective fitness values. There is the possibility that the same genome may be included in the tournament more than once. For crossover, two tournaments of size three are run thereby giving two genomes to be crossed. For mutation, one tournament of size two is run thereby giving one genome to be mutated. Rank fitness selection works by first sorting the population in non-increasing order according to

fitness and then selects a genome at random where the probability of a genome being selected is proportional to its rank fitness. With the population in sorted order, the first genome is given a rank fitness of 1, the second genome is given a rank fitness of 2, ..., and the last genome is given a rank fitness of n which is the population size. The rank fitness prefix-sum for each genome is calculated in an array such that the first index value in the prefix-sum array is 1 while the last index value in the prefix-sum array is $\frac{n(n-1)}{2}$. A uniform random number is selected in the range $[0, \frac{n(n-1)}{2}]$. The genome selected G is the one in which the random number is greater than the previous prefix-sum for genome G_{i-1} and less than or equal to the prefix-sum for G_i . Genomes with a higher fitness will have a higher rank fitness and thus will have a higher probability of being selected for either crossover or mutation. See Figure 4.3.

```

BEGIN
    Population  $P$  with size  $n$  has been evaluated
    Sort  $P$  in non-increasing order
    For i=1 to  $n$  do
         $P[i-1].rankFitness = i$ 
    End for
    For i=0 to  $n - 1$  do
         $P[i].prefixSum = \sum_{k=0}^i (P[k].rankFitness)$ 
    End for
    Select a random number  $r = unif\left(0, \frac{n(n-1)}{2}\right)$ 
    Genome selected  $G$ 
    For i=0 to  $n - 1$  do
        If  $P[i].prefixSum \geq r$  then
             $G = P[i]$ 
            Break
        End if
    End for
    Return  $G$ 
END

```

FIGURE 4.3: Here you see the rank fitness selection algorithm.

4.2.7.3 Fitness Function

To construct the fitness function, real robot motion data was collected which included 1040 sample points. X-translation, y-translation, and z-rotation were recorded for the real robot. With a camera overhead, the real robot was placed in the arena and was repeatedly commanded to go forward (relative from its current position and orientation) 25 centimeters. Care was taken to avoid having the robot collide with the arena

walls. Once the robot consecutively moved forward three times, the robot rotated in place by 135 degrees and was not recorded during this period of rotation.

Using the camera, the robot's position and orientation before and after performing the forward command was recorded for each forward command issued. The robot's position and orientation was not reset each time the robot performed a forward command. Instead, the robot was allowed to continue forward from its current position and orientation as it traveled around the arena. Thus each recorded pair of initial and final states were translated and rotated to be in the same reference frame such that the robot was always at the arena origin facing down the positive x-axis before performing the forward command. See Figure 4.4 and 4.5. Note that from the robot's perspective, it is always facing down its local positive x-axis no matter its position or orientation as seen from some other reference point.

Analyzing the distribution of x-translation, y-translation, and z-rotation separately, all are unimodal and nearly symmetric with only a slight skew from their respective means. See Table 4.2 and Figure 4.6. When viewing the change in dimensions together, a large mass is centered around the point: 23.8644631679cm, 0.338269853117cm, and -0.00417473025048rad. See Figure 4.7. Based on the variance-covariance matrix, all three dimensions vary together and are not independent from one another.

	X-translation	Y-translation	Z-rotation
Max	46.5183415482cm	19.7486813209cm	6.23776rad
Min	0.0cm	-19.4028539247cm	-1.148163rad
Mean/Centroid	23.8644631679cm	0.338269853117cm	-0.00417473025048rad
Mode	25.0cm	-1.0cm	0.0rad
Variance	10.3960320996	9.46441502772	0.152467827567
Standard Deviation	3.22428784378	3.07642894079	0.390471289043
Covariance with X-translation	10.4060572	2.3348963	0.0685591
Covariance with Y-translation	2.3348963	9.47354175	0.08654865
Covariance with Z-rotation	0.0685591	0.08654865	0.15261486

TABLE 4.2: Here you see the distribution metrics for surge, sway, and yaw.

With the real robot motion data collected and analyzed, a metric was needed for the fitness function. This metric would need to indicate how different or dissimilar the movement of the simulated robot was from the real robot while running the GA. The intuition was that as the simulated robot motion observed moved closer to the centroid of the real robot motion, the simulated motion would become increasingly

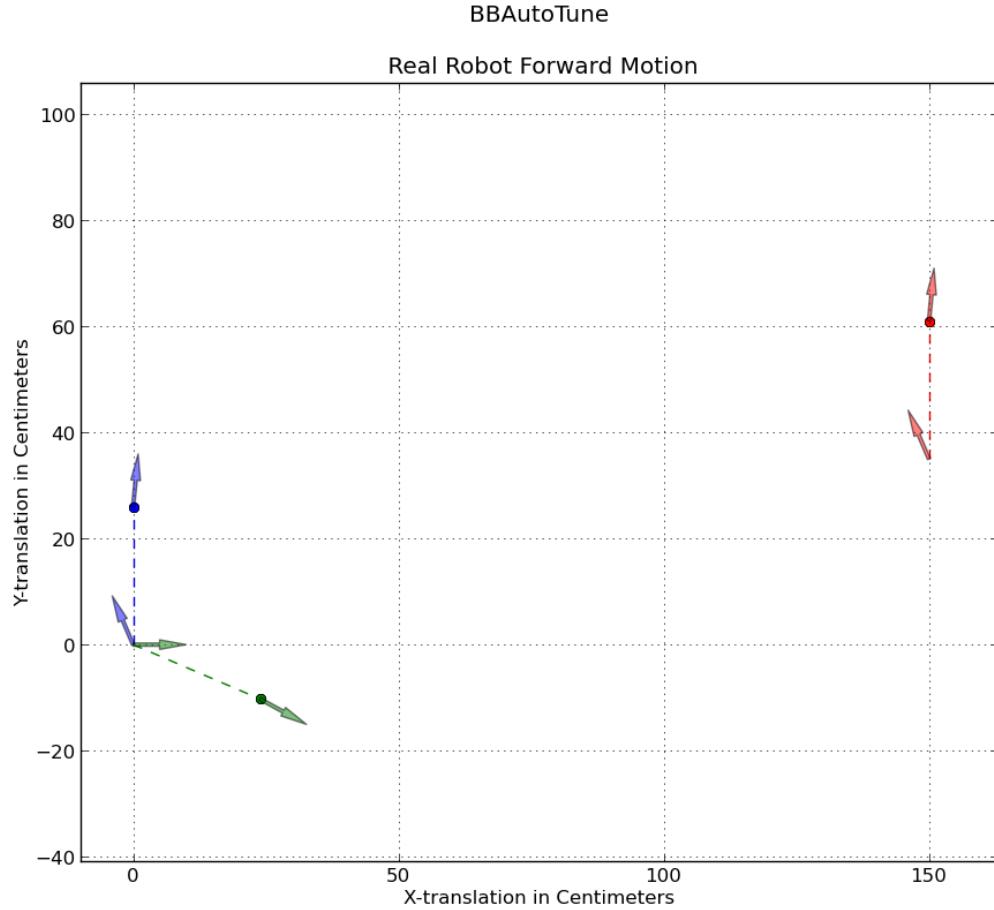


FIGURE 4.4: Here you see a specific instance of how the raw real robot forward motion data was translated and rotated such that each pair of initial and final states have the same reference frame. The red arrows, dot, and line represent the raw initial and final state of the robot recorded from the overhead camera before and after it performed the forward command. The blue arrows, dot, and line represent the initial and final state translated to the origin. The green arrows, dot, and line represent the initial and final state rotated by the amount needed to align the initial orientation with the positive x-axis. For each colored group, the dot and arrow pair represent the position and orientation of the robot after having performed the forward command while the arrow without a dot represents the position and orientation state of the robot before it performed the forward command.

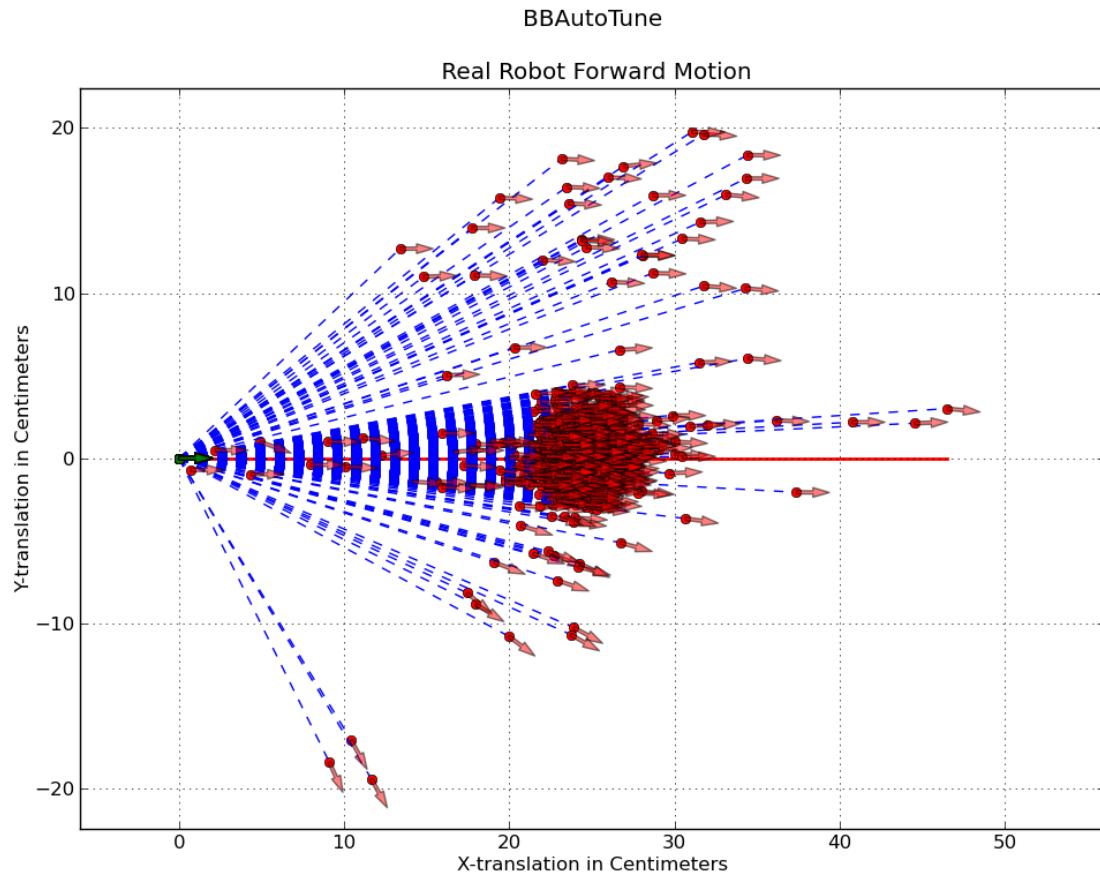


FIGURE 4.5: Here you see the collected real robot forward motion plotted from the same reference point. The green dots and arrows indicate the starting position and orientation of the real robot before the forward command was performed. The red dots and arrows indicate the ending position and orientation of the real robot after the forward command was performed. The dots represent position and the arrows represent orientation. The broken blue lines connect the initial states to their corresponding final states.

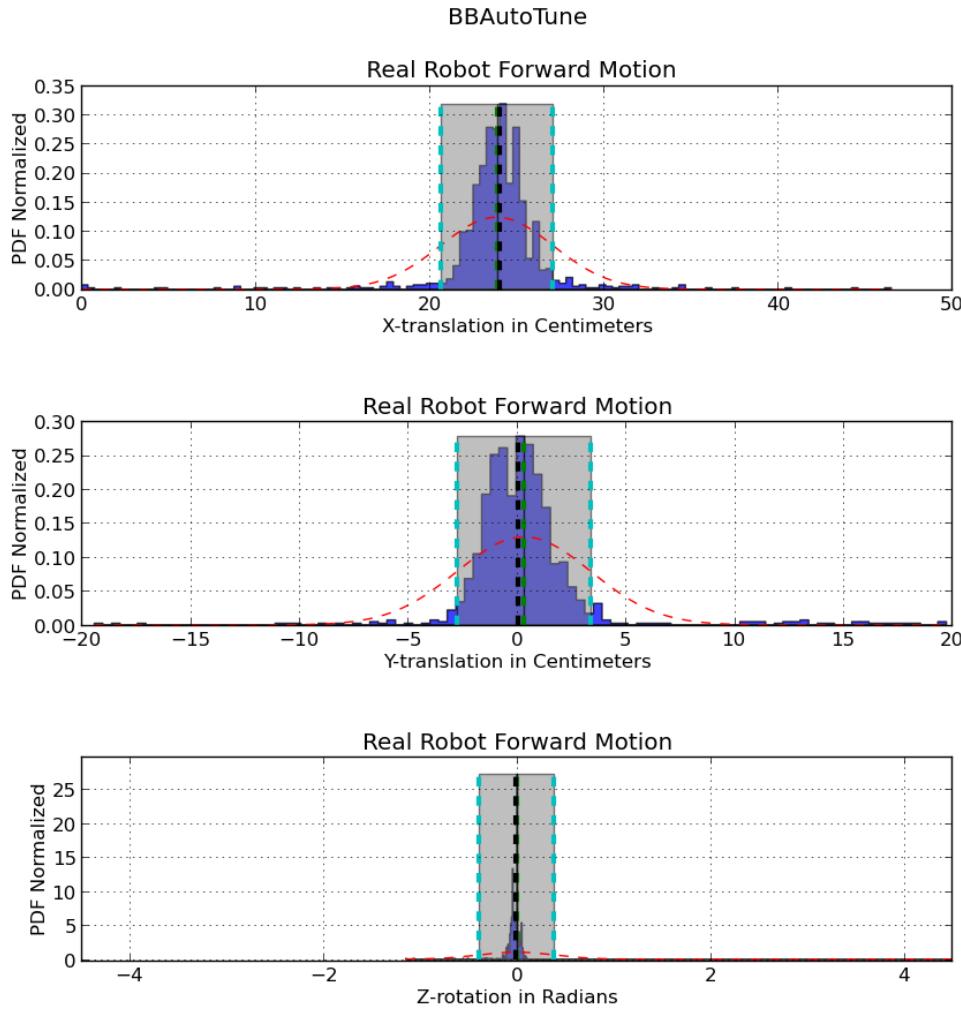


FIGURE 4.6: Here you see the distribution of x-translation, y-translation, and z-rotation dimensions recorded for the real robot forward motion. The broken black bar represents the mode, the green broken bar represents the mean, the cyan broken bars represent one standard deviation from the mean, and the red broken curve represents the best fit normal curve.

BBAutoTune
Real Robot Forward Motion

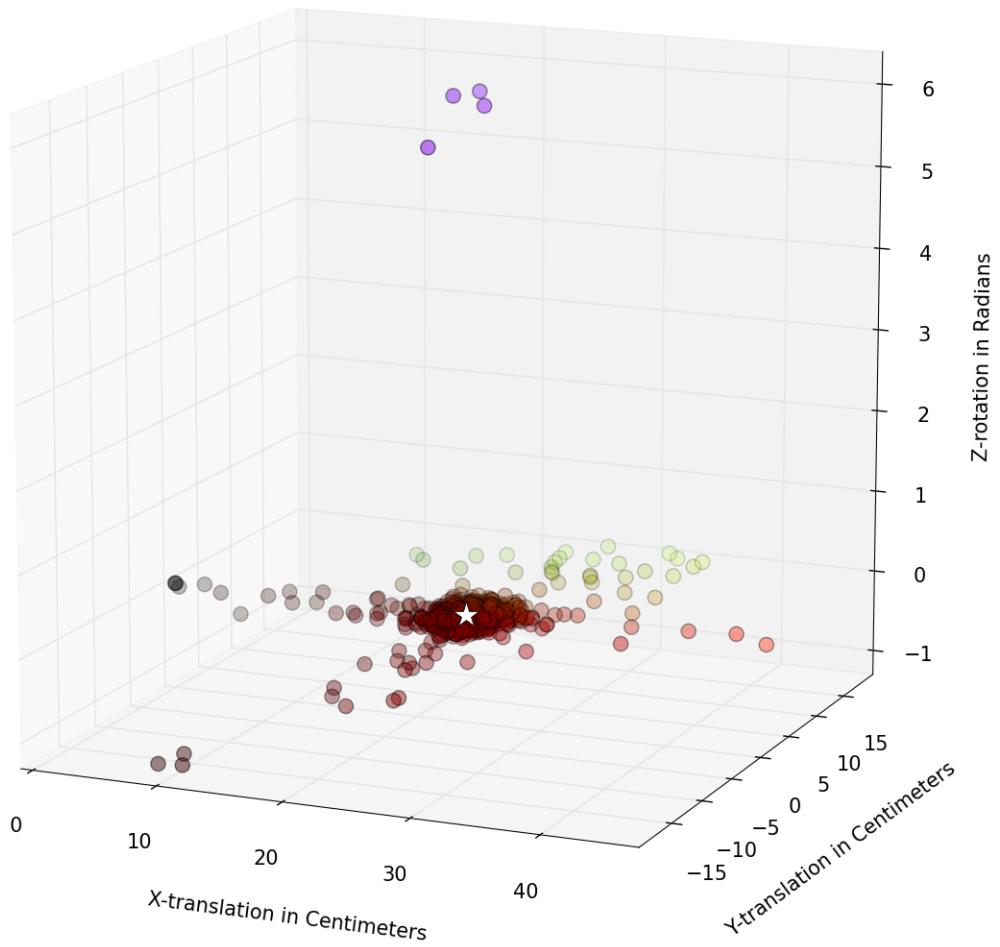


FIGURE 4.7: Here you see a 3D scatter plot of the x-translation, y-translation, and z-rotation values for the real robot after it performed the forward command. Positive values of x-translation, y-translation, and z-rotation contribute a portion of red, green, and blue respectively to each scatter plot. The star located at the center of the large mass of points is the centroid.

indistinguishable from the real motion. Various distance functions were looked at and the Mahalanobis distance was chosen as the basis for the fitness function. The Mahalanobis distance is the generalized form of the Euclidean distance such that the Mahalanobis distance accounts for the correlation in the data set since it is computed using the inverse of the variance-covariance matrix [14]. For uncorrelated data, the Mahalanobis distance reduces to the Euclidean distance [15].

Inefficiencies with the overhead camera and the timing at which the position and orientation state of the real robot was captured could have potentially skewed the real robot motion model with outliers ultimately resulting in a skewed multivariate mean location and a skewed inverse variance-covariance matrix making the Mahalanobis distance skewed. To account for the potential outliers in the real motion data set, a robust mean location and a robust variance-covariance matrix was computed from the data set using the Fast-MCD algorithm implemented in the Scikit-learn Python module [16][17]. The classical mean location was 23.8644632cm, 0.338269853cm, and -0.00417473025rad for x-translation, y-translation, and z-rotation respectively while the robust mean location was 23.9934044cm, 0.0351240536cm, and -0.0189964938rad for x-translation, y-translation, and z-rotation respectively. The left matrix below was the classical variance-covariance matrix while on the right was the robust variance-covariance matrix returned by the Fast-MCD algorithm.

$$\begin{pmatrix} 10.3960321 & 2.33264688 & 0.06849305 \\ 2.33264688 & 9.46441503 & 0.08646527 \\ 0.06849305 & 0.08646527 & 0.15246783 \end{pmatrix} \begin{pmatrix} 1.46298445 & 0.13924542 & 0.00223493 \\ 0.13924542 & 1.64197605 & 0.0168499 \\ 0.00223493 & 0.0168499 & 0.00173777 \end{pmatrix}$$

The resulting samples weighted higher than others by the Fast-MCD algorithm are shown in Figure 4.8. These samples where used to calculate the robust mean and the robust variance-covariance matrix returned by the algorithm. By substituting the robust mean and the inverse of the robust variance-covariance matrix into the Mahalanobis distance calculation, the robust distance RD for any sample point can be computed [18]. Comparisons between the Mahalanobis distance and the robust distance for each of the 1040 real robot motion data points are shown in Figure 4.9. As the data points travel away from the centroid, the robust distance increases more rapidly than the Mahalanobis distance.

Only three out of the total six degrees of freedom were recorded for the real robot. However,

BBAutoTune
Real Robot Forward Motion Robust Support Samples

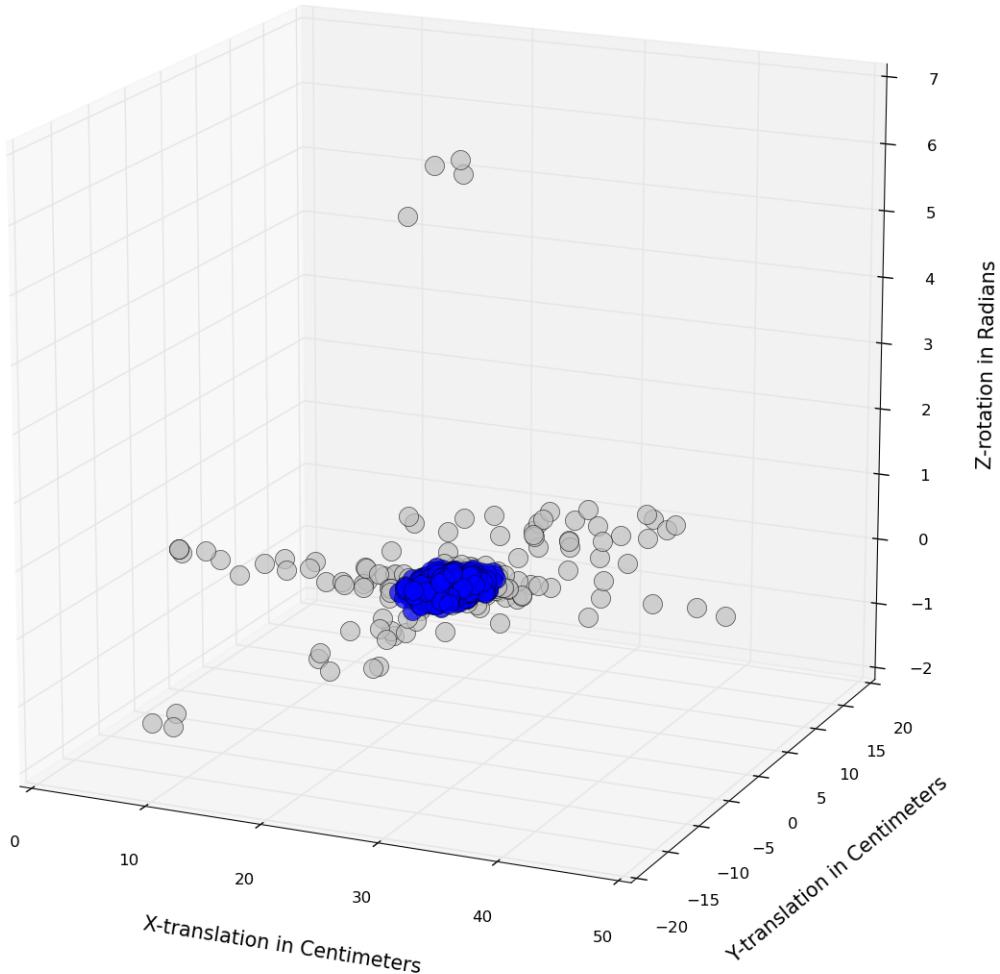


FIGURE 4.8: Here you see a 3D scatter plot of the support samples in blue used to calculate the robust mean location and the robust variance-covariance matrix as returned by the Fast-MCD algorithm.

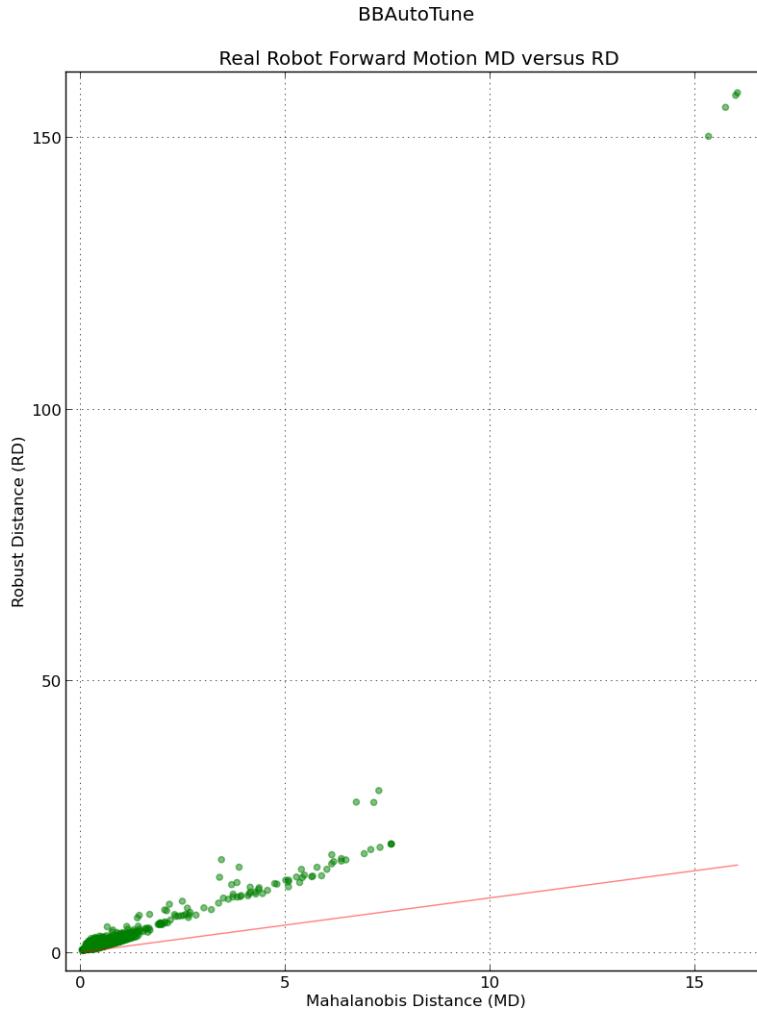


FIGURE 4.9: Here you see a scatter plot comparing the Mahalanobis distance versus the robust distance for each data point collected of the real robot motion.

in Blender there were an additional three degrees (pitch, roll, and heave) to contend with. Thus, the simulated-robot base's x-rotation, y-rotation, and z-translation were constrained. Additionally, all wheels had their z-translation constrained. As an added precaution, penalties were added onto the robust distance by the absolute amount the simulated robot's base violated x-rotation, y-rotation, and/or z-translation. Additionally, a time penalty was added onto the robust distance by the amount of time in seconds the simulated robot took to evaluate greater than one second with a max penalty of 15 seconds since any given evaluation period only lasted a total of 16 seconds.

Once the simulated robot was run through the game engine evaluation period (using the physics parameter settings decoded from the currently being evaluated genome G_i 's genes), 14 pieces of data was collected by the robot monitor for use in the fitness function. Let $S = [x_t, y_t, z_t, x_r, y_r, z_r, t]$ be the starting state of the simulated robot at the beginning of the evaluation period at time $S[6] = t$ where the subscript t refers to translation and the subscript r refers to rotation. Let $S' = [x_t, y_t, z_t, x_r, y_r, z_r, t]$ be the ending state of the simulated robot at the end of the evaluation period at time $S'[6] = t$. Also, let $RD(x\text{-}translation, y\text{-}translation, z\text{-}rotation)$ be the robust distance. The fitness function and thus the fitness of G_i was defined as $Fitness(S, S') = RD(S'[0] - S[0], S'[1] - S[1], S'[5] - S[5]) + |S'[4] - S[4]| + |S'[3] - S[3]| + |S'[2] - S[2]| + ((S'[6] - S[6]) - 1) = G_i$'s fitness. The range of this function is $[0, \infty)$. Since the goal of this thesis was to have the simulated robot move as the real robot, the desired output of this function was 0.0 implying that three objectives were met:

- The simulated robot's x-translation, y-translation, and z-rotation after moving was 23.9934044cm, 0.0351240536cm, and -0.0189964938rad respectively.
- The simulated robot stopped moving after one second.
- The simulated robot did not fall through the floor, flip over, roll over, and/or launch upward.

Thus the goal of the GA was to minimize this function whereby lower output values were a higher fitness than higher output values.

4.2.7.4 Evaluation Setup

For each evaluation period (the running of the game engine), the 3D robot model's local coordinate system was axis aligned with the world coordinate system, the robot was faced forward looking down the positive x-axis, and its local origin was placed at the world origin. See Figure 4.10. Before each evaluation period, the physics engine parameters were set to the values decoded from the genes of the currently being evaluated genome. All evaluation periods lasted no more than 16 seconds. If the robot stopped moving before 16 seconds, then the evaluation period ended immediately. The only applied force to the robot was the wheel torque where each wheel received the same amount of applied torque for the same duration. The duration of applied torque was roughly 16 milliseconds after which no further force was applied to the robot.

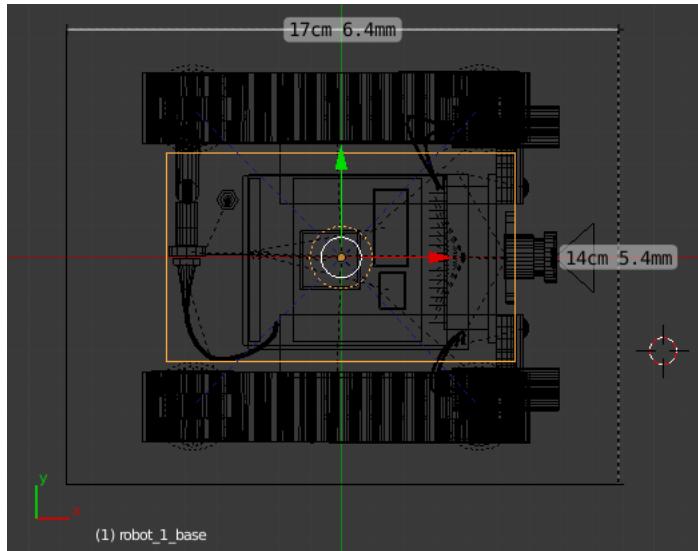


FIGURE 4.10: Here you see the simulated robot's local coordinate system axis aligned with the world coordinate system.

4.3 Platform

For all experiments, BBAutoTune was run on a 64bit Linux operating system with 32GB of RAM and an Intel Core i7-4770K four core processor running at 3.9GHz.

4.4 Experimental Designs

The purpose for experiment one was to determine a base set of physics parameters that have significant influences over the physics simulation. Experiments two through five involved actually tuning the physics engine and comparing the simulated versus real robot motion. Self-adaptation and tournament versus rank fitness selection were the experimental GA parameters for experiments two through five.

4.4.1 Experiment one: physics engine parameter influence.

The potential Blender physics engine parameter candidates—to be tuned by the GA—were analyzed for their influence over the physics simulation. To accomplish this goal, a racquetball like environment was constructed in Blender which consisted of a ball, an enclosed arena, and an automated racket controlled via a Python script. See Figure 4.11. Note that the ball was allowed to travel in all three dimensions and was completely physics based.

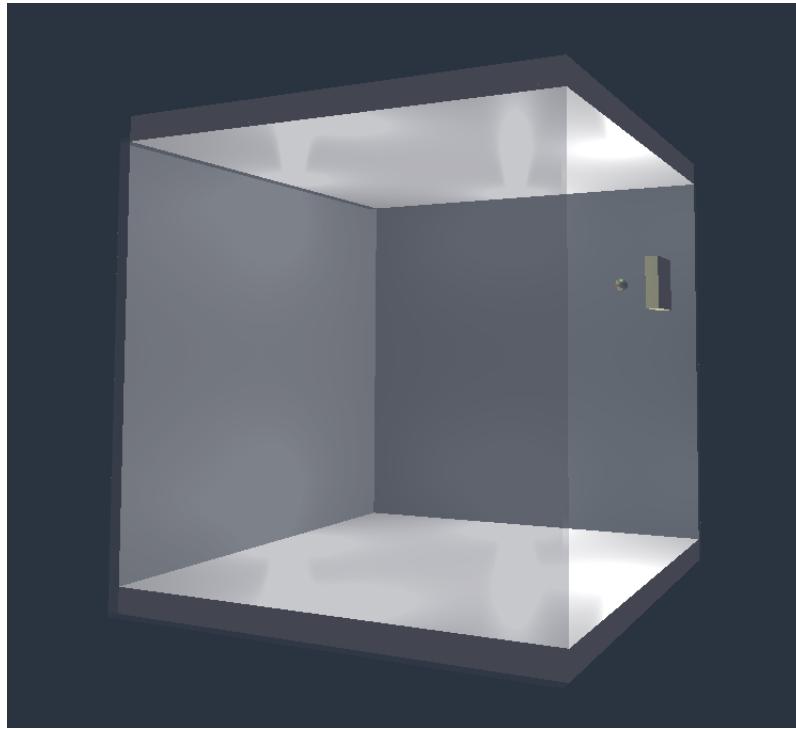


FIGURE 4.11: Here you see the racquetball environment used to test the physics engine parameter candidates' influence over the physics simulation.

44 candidate parameters were selected for testing in the racquetball environment. All parameters

tested were only associated with the ball. Before all of these parameters were tested for their influence, *nice* values were selected for each such that the ball’s behavior was visually normal. A standard was established where the ball was allowed to run for five seconds (with all parameters being set to their nice values) during which its location was recorded at roughly 60 times per second. With the standard established—with which all other runs would be compared to—a parameter was selected (from the candidate pool), its value was tweaked, the ball was run for five seconds with its location being recorded at roughly 60 times per second, and after the run was over, the tweaked parameter’s value was set back to its nice value. This sequence was repeated for all 44 candidate parameters.

4.4.2 Experiment two: tournament selection with self-adaptation.

Very early runs of BBAutoTune attempted to tune physics parameters: gravity, sub-steps, FPS, use material physics, material friction, material elasticity, mass, form factor, velocity maximum, damping translation, damping rotation, use collision bounds, collision bound type, and torque where the search space for each parameter was its entire valid range. This proved to be problematic since some of the valid ranges are quite large especially torque with a max upper bound of 3.40×10^{38} . While running the game engine, if the genome’s genes decoded to relatively high physics engine parameter values, world coordinates would return the Python values “NaN” or “inf”.

To rectify this issue, the set of physics parameters selected for tuning was pruned and for the parameters left, their ranges were shortened to reasonable upper and lower bounds found manually. The resulting set of tunable physics parameters and their ranges for experiment two were: gravity [0.0,15.0], sub-steps [1,5], FPS [30,10000], material friction [0.0,100.0], material elasticity [0.0,1.0], mass [0.010,15.0], velocity maximum [0.0,1000.0], damping translation [0.0,1.0], damping rotation [0.0,1.0], collision bound type [TRIANGLE_MESH,CONVEX_HULL,CYLINDER,SPHERE], and torque [0.0,100.0]. Use material physics and use collision bounds were set to true and held constant. Form factor was set to 1.0 and was held constant.

The GA was run for 500 generations, tournament selection was used, the population size was set to 10, elitism was set to 2, crossover and mutation were performed separately, and the crossover and mutation probabilities were self-adapted over time.

4.4.3 Experiment three: tournament selection without self-adaptation.

The set of tunable physics parameters for experiment three was the same as experiment two. The GA was run for 500 generations, tournament selection was used, the population size was set to 10, elitism was set to 2, crossover and mutation were performed separately, and the crossover and mutation probabilities were held constant at 0.8 and 0.2 respectively.

4.4.4 Experiment four: rank fitness selection with self-adaptation.

The set of tunable physics parameters for experiment four was the same as experiment two. The GA was run for 500 generations, rank fitness selection was used, the population size was set to 10, elitism was set to 2, crossover and mutation were performed separately, and the crossover and mutation probabilities were self-adapted over time.

4.4.5 Experiment five: rank fitness selection without self-adaptation.

The set of tunable physics parameters for experiment five was the same as experiment two. The GA was run for 500 generations, rank fitness selection was used, the population size was set to 10, elitism was set to 2, crossover and mutation were performed separately, and the crossover and mutation probabilities were held constant at 0.8 and 0.2 respectively.

4.5 Experimental Results

Experiment two had the highest performing genome out of experiments two through five. Experiments three through five had similar highest fitnesses. Using self-adaptation with tournament selection resulted in the highest fitness observed among experiments two through five. Comparing experiment two to three, self-adaptation was beneficial but was not beneficial when comparing experiment five to four.

4.5.1 Experiment one: physics engine parameter influence.

To compare the recorded tweaked-parameter ball paths with the recorded standard, four methods were utilized to give an indication as to how much influence any one candidate parameter had over the simulation. The first method was visual inspection. All 44 tweaked-parameter ball paths were plotted against the standard. See Figure 4.12 and Figure 4.13. The second method was an in-house algorithm, titled the *Lettier*

distance, which gives the maximum euclidean distance between two discrete curves P and Q . Informally, imagine holding a rubber band in your hands where the left hand affixes the left end of the rubber band to the first point in P and the right hand affixes the right end of the rubber band to the first point in Q . During each iteration, you advance the left end of the rubber band to the next point in P and you advance the right end of the rubber band to the next point in Q . If the distance grows between point $p_i \in P$ and point $q_j \in Q$, the rubber band stretches but never shrinks. If $|P| < |Q|$ then you hold the left end of the rubber band to the last point in P and continue advancing to the last point in Q and vice versa. Once you reach the last point in P and the last point in Q , the resulting length of the rubber band is the max euclidean distance between P and Q . See Figure 4.14. The third method was the discrete Fréchet distance and the fourth method was the Hausdorff distance [19] [20].

20 parameters out of the initial 44 showed no significant influence over the 3D physics simulation in Blender. Thus, the resulting 24 parameters which did have a significant influence were targeted for tuning by the genetic algorithm. See Table 4.3.

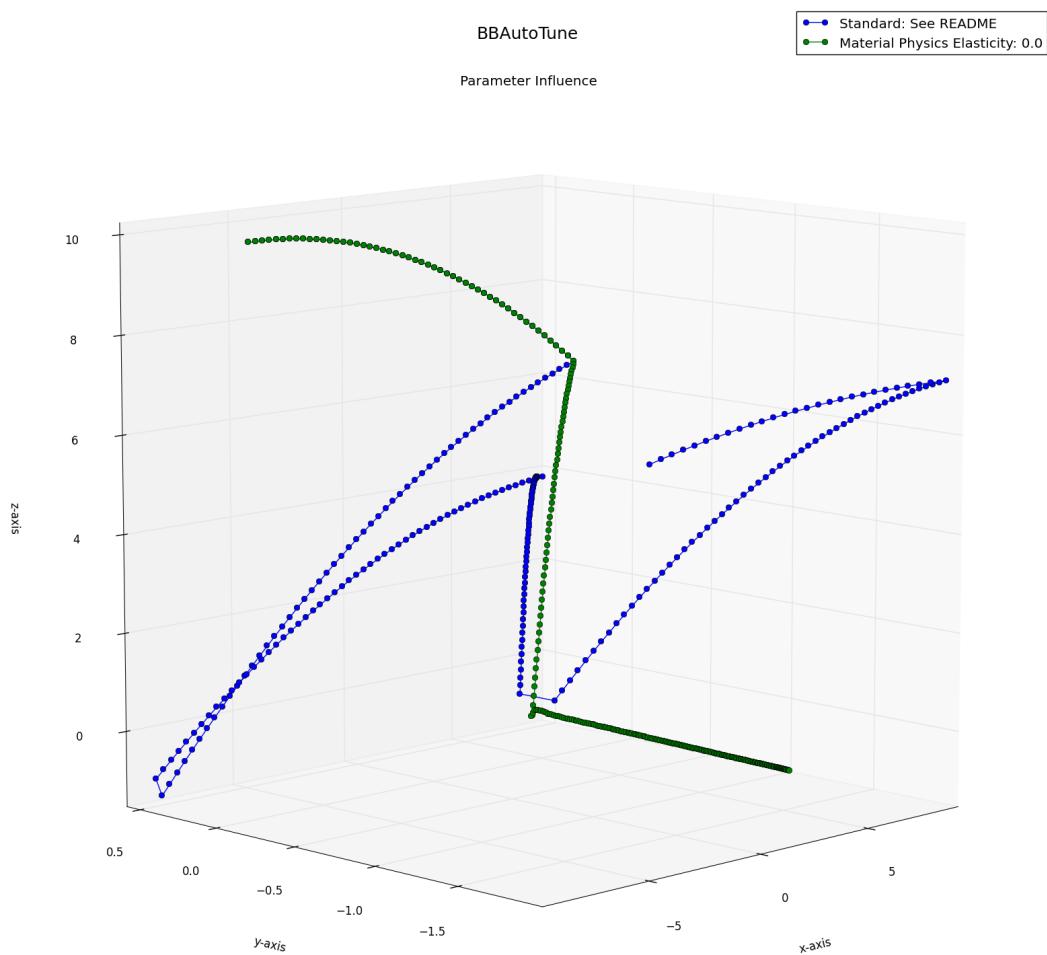


FIGURE 4.12: Here you see the dissimilarity of the ball paths between the tweaked, material-physics elasticity parameter and the standard (where no parameters were tweaked).

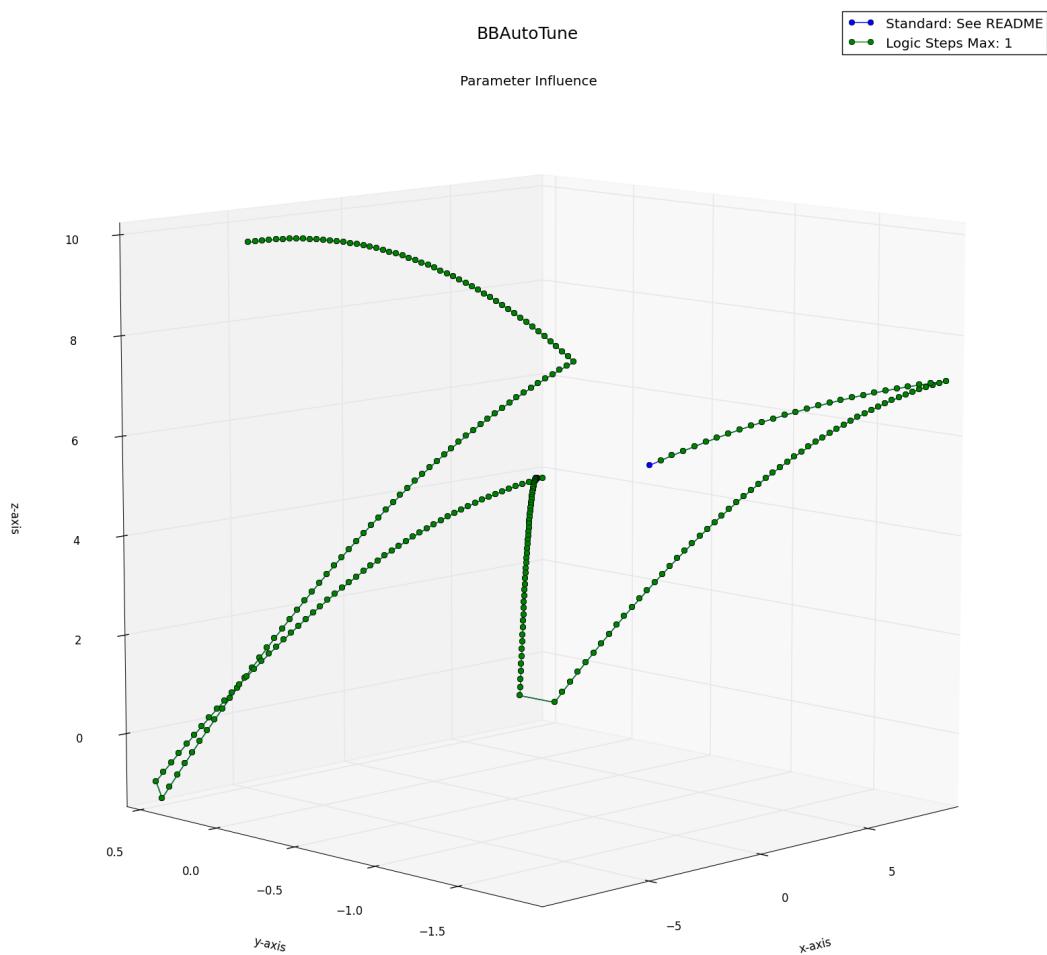


FIGURE 4.13: Here you see the similarity of the ball paths between the tweaked, logic-steps max parameter and the standard (where no parameters were tweaked).

Tweaked Parameter: Value	Lettier Distance	Discrete Fréchet Distance	Hausdorff Distance
Gravity: $1.0 \frac{m}{s^2}$	12.7741189989	20.948159542	9.86343687719
Physics Steps Max: 1	0.329232644023	0.329232644023	0.329232644023
Physics Sub-steps: 50	19.3073641073	17.0708640308	11.3180046289
Physics FPS: 1	218.037284056	211.287842927	205.848670021
Logic Steps Max: 1	0.329232644023	0.329232644023	0.329232644023
Physics Deactivation Linear Threshold: 10000.0	0.329232644023	0.329232644023	0.329232644023
Physics Deactivation Angular Threshold: 10000.0	0.329232644023	0.329232644023	0.329232644023
Physics Deactivation Time: 0.0	0.329232644023	0.329232644023	0.329232644023
Occlusion Culling: False	0.329232644023	0.329232644023	0.329232644023
Occlusion Culling Resolution: 1024	0.329232644023	0.329232644023	0.329232644023
Material Physics: False	18.0830473811	18.0830473811	18.0830473811
Material Physics Friction: 100.00	4.93924881608	4.94095279557	3.8109066203
Material Physics Elawosticity: 0.0	17.3874844829	17.0233755244	11.3180046289
Force Field Force: 1.00	0.329232644023	0.329232644023	0.329232644023
Force Field Damping: 1.00	0.329232644023	0.329232644023	0.329232644023
Force Field Distance: 20.00	0.329232644023	0.329232644023	0.329232644023
Force Field Align to Normal: True	0.329232644023	0.329232644023	0.329232644023
Physics Type: Dynamic	3.55513601614	18.1217630973	3.275450812
Actor: False	0.329232644023	0.329232644023	0.329232644023
Ghost: True	138.489494312	138.489494312	132.02387242
Use Material Force Field: True	0.329232644023	0.329232644023	0.329232644023
Rotate From Normal: True	0.329232644023	0.329232644023	0.329232644023
No Sleeping: True	0.329232644023	0.329232644023	0.329232644023
Attributes Radius: 1cm	0.329232644023	0.329232644023	0.329232644023
Attributes Mass: 10000.0	3.57800913743	3.30042073543	3.21534852885
Attributes Form Factor: 0.0	3.55513601614	18.1217630973	3.275450812
Velocity Minimum: 1.0	0.329232644023	0.329232644023	0.329232644023
Anisotropic Friction: True	0.329232644023	0.329232644023	0.329232644023
Velocity Maximum: 1.0	17.1034960114	17.130808295	16.8922077042
Damping Translation: 1.0	17.9729741832	17.9729741832	17.9729741832
Damping Rotation: 1.0	8.09457671285	5.47840838373	5.18331655137
Collision Bounds: False	8.81773036062	17.4044978663	2.80531111777
Collision Bounds Margin: 0m	18.740072391	9.79629223411	3.78119352088
Collision Bounds: False	4.28865271192	4.23797419961	3.72440427516
Launch Dynamic Object Settings Force X: 30.0	5.21169989737	5.21169989737	3.78308993247
Launch Dynamic Object Settings Torque X: 30.0	8.82943537929	8.70403741077	6.58255281735
Launch Dynamic Object Settings AngV X: 30.0	2.93219084728	2.63246279507	1.82341393368
Launch Dynamic Object Settings LinV X: 0.0	19.1625050915	19.1625050915	17.0670582483
Launch Damping Frames: -32768	0.329232644023	0.329232644023	0.329232644023
Collision Dynamic Object Settings Force X: 30.0	5.32618850819	18.9696056987	3.80913153981
Collision Dynamic Object Settings Torque X: 30.0	1.67496526126	1.52842619107	1.52842619107
Collision Dynamic Object Settings LinV X: 0.0	19.2801183238	19.1535557726	3.3918725084
Collision Dynamic Object Settings AngV X: 30.0	4.69097055763	18.7403224372	3.80913153981
Collision Damping Frames: -32768	0.329232644023	0.329232644023	0.329232644023

TABLE 4.3: Here you see the distances between each tweaked-parameter ball path and the standard ball path per the three algorithms utilized. The highlighted tweaked parameters were determined not influential.

```

BEGIN
   $P = \langle p_1, p_2, \dots, p_n \rangle$ 
   $Q = \langle q_1, q_2, \dots, q_m \rangle$ 
   $max = 0.0$ 
  For all  $p_i \in P$  and  $q_j \in Q$  do
     $d = \|p_i - q_j\|$ 
    If  $max < d$  then
       $max = d$ 
    End if
  End for
  Return  $max$ 
END

```

FIGURE 4.14: Here you see the Lettier distance algorithm.

4.5.2 Experiment two: tournament selection with self-adaptation.

The minimum values reached for the highest, average, and lowest fitness were 0.9306191001, 1.024632475, and 0.9315692954 respectively. The minimum and maximum probability for both crossover and mutation obtained over the course of the 500 generations was 0.001 and 0.999. Interestingly, as the highest, average, and lowest fitness began to converge, the crossover and mutation probability flipped causing the population to be entirely mutated resulting in essentially a randomly generated population thereby destroying the high fitness solutions found. Notice that after the probabilities flipped, the highest, average, and lowest fitness diverged. This converging and then diverging occurred twice during the 500 generation run. Up to generation 376, the highest fitness steadily declined and then afterward remained fairly stable. See Figure 4.15. Table 4.4 lists the best and worst physics parameters found by the GA corresponding to the best and worst fitness observed during the 500 generation run.

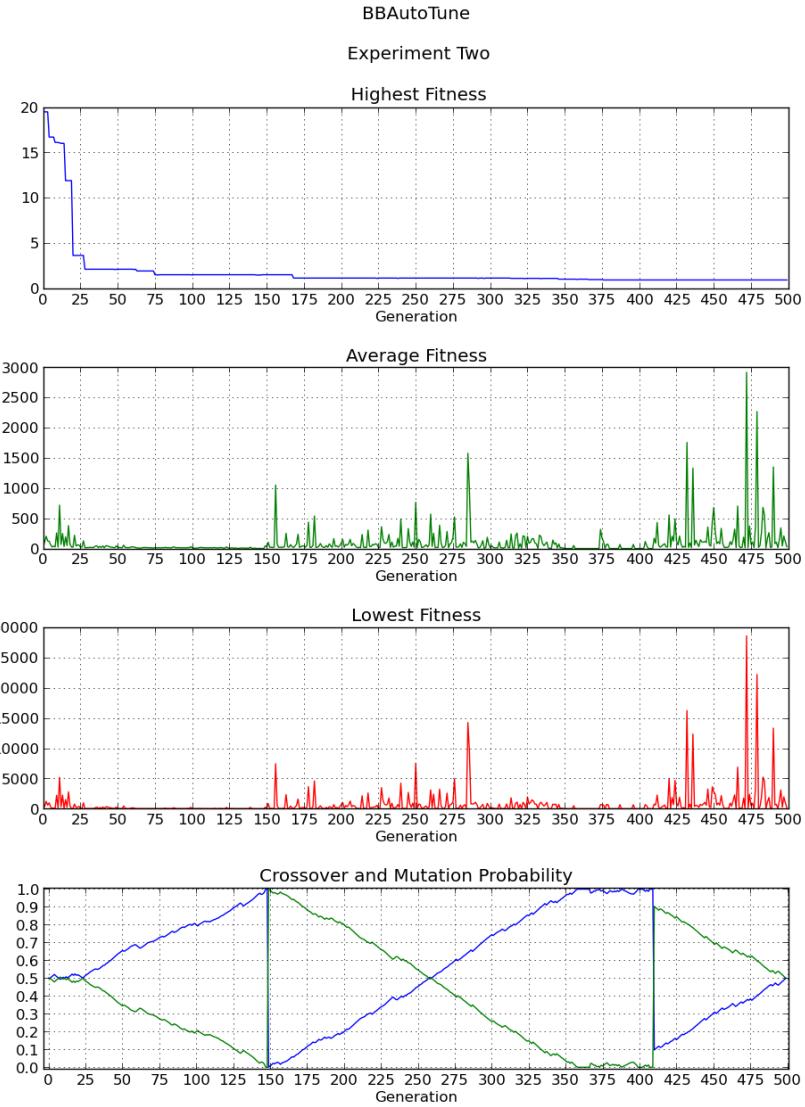


FIGURE 4.15: Here you see the highest, average, and lowest fitness in addition to the crossover and mutation probability over the course of 500 generations. For the bottom plot, the crossover probability is shown in blue while the mutation probability is shown in green.

	Best	Worst
Gravity	$2.89862416489 \frac{m}{s^2}$	$10.1989482496 \frac{m}{s^2}$
Sub-steps	5	5
FPS	30	30
Material Friction	59.5011814113	77.8151135094
Material Elasticity	0.0742521056997	0.279061300281
Mass	4.33392950881	1.76678194417
Velocity Max	900.11395466	647.638168514
Damping Translation	1.0	0.0
Damping Rotation	0.691143247902	0.648240602049
Collision Bound Type	SPHERE	SPHERE
Torque	82.7271515601	100.0
Fitness	0.930619100106	28584.2244771

TABLE 4.4: Here you see the best and worst physics parameters found by the GA corresponding to the best and worst fitness observed during the 500 generation run for experiment two.

4.5.3 Experiment three: tournament selection without self-adaptation.

The minimum values reached for the highest, average, and lowest fitness were 1.0827696957, 2.892412821, and 7.7702053307 respectively. Up until generation 146, the highest fitness steadily declined and then afterwards remained fairly stable. See Figure 4.16. In contrast to experiment two, it would seem that the self-adaptation side effect of periodically randomizing the population helped experiment two obtain a lower highest fitness than experiment three. Table 4.5 lists the best and worst physics parameters found by the GA corresponding to the best and worst fitness observed during the 500 generation run.

	Best	Worst
Gravity	$14.09509165745291 \frac{m}{s^2}$	$0.8362602682788933 \frac{m}{s^2}$
Sub-steps	1	1
FPS	30	30
Material Friction	79.87292012678728	100.0
Material Elasticity	0.7331947746415657	0.6643893038716038
Mass	15.0	12.761484385447746
Velocity Max	0.0	250.88084935511978
Damping Translation	1.0	0.33429608723965537
Damping Rotation	0.14340109658364997	0.0
Collision Bound Type	TRIANGLE_MESH	SPHERE
Torque	47.85677413931377	63.68300968863001
Fitness	1.0827696957	1617.02428027

TABLE 4.5: Here you see the best and worst physics parameters found by the GA corresponding to the best and worst fitness observed during the 500 generation run for experiment three.

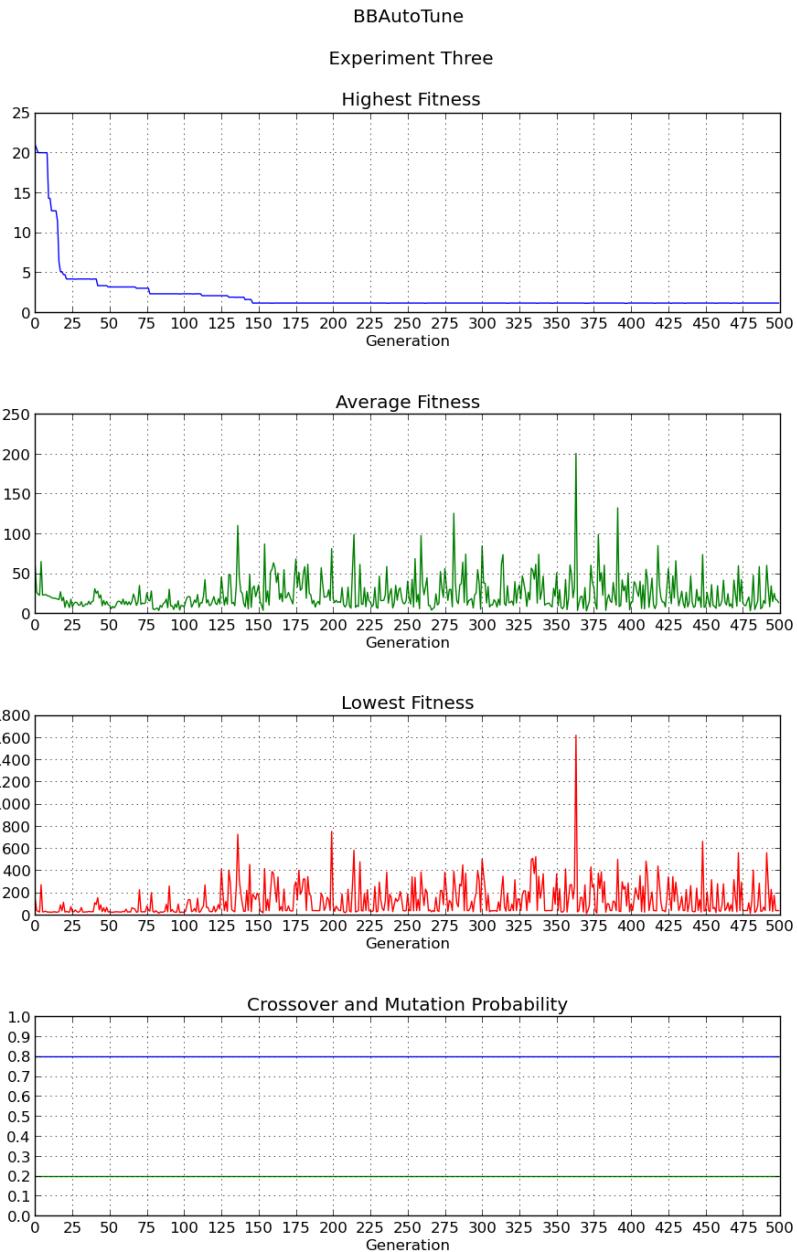


FIGURE 4.16: Here you see the highest, average, and lowest fitness in addition to the crossover and mutation probability over the course of 500 generations. For the bottom plot, the crossover probability is shown in blue while the mutation probability is shown in green.

4.5.4 Experiment four: rank fitness selection with self-adaptation.

The minimum values reached for the highest, average, and lowest fitness were 1.1309704845, 2.7714017205, and 2.5217479907 respectively. The minimum and maximum probabilities observed for crossover, during the 500 generation run, were 0.253 and 0.999 respectively. For mutation, the minimum and maximum probabilities observed were 0.001 and 0.747 respectively. From generation zero to 79, the highest fitness steadily declined and then fell more gradually afterwards for rest of the duration. The same periodic converging and then diverging seen in experiment two was observed again in experiment four. See Figure 4.17. Table 4.6 lists the best and worst physics parameters found by the GA corresponding to the best and worst fitness observed during the 500 generation run.

	Best	Worst
Gravity	$13.8256917774 \frac{m}{s^2}$	$15.0 \frac{m}{s^2}$
Sub-steps	2	5
FPS	30	30
Material Friction	61.2749944576	0.0
Material Elasticity	0.171754015461	0.649745829218
Mass	15.0	0.158414320671
Velocity Max	1000.0	1000.0
Damping Translation	0.0	0.387362543282
Damping Rotation	0.984539067046	0.687852083808
Collision Bound Type	SPHERE	CONVEX_HULL
Torque	6.70058480184	75.8262976514
Fitness	1.1309704845	16396.2145412

TABLE 4.6: Here you see the best and worst physics parameters found by the GA corresponding to the best and worst fitness observed during the 500 generation run for experiment four.

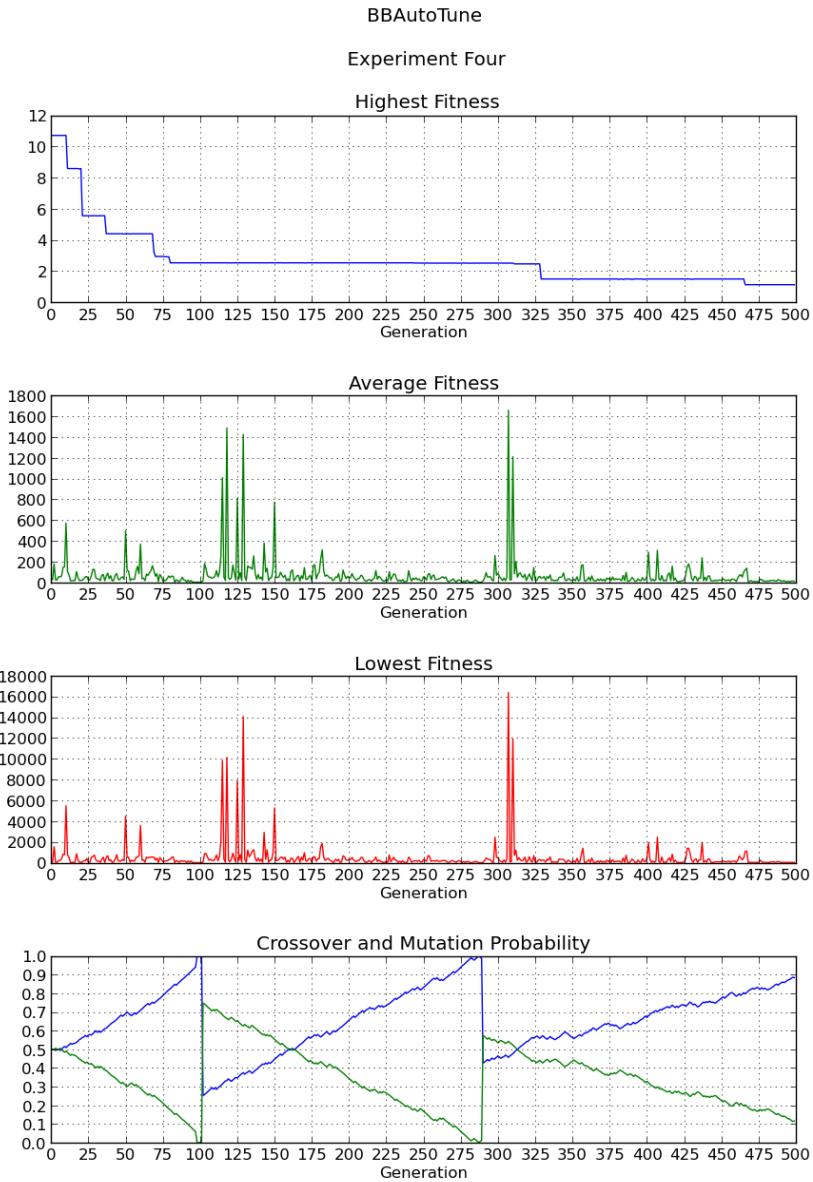


FIGURE 4.17: Here you see the highest, average, and lowest fitness in addition to the crossover and mutation probability over the course of 500 generations. For the bottom plot, the crossover probability is shown in blue while the mutation probability is shown in green.

4.5.5 Experiment five: rank fitness selection without self-adaptation.

The minimum values reached for the highest, average, and lowest fitness were 1.0638026764, 3.4974350451, and 11.6425679783 respectively. Throughout the 500 generation run, the highest fitness continued to decline reaching its lowest value at generation 469. See Figure 4.18. Table 4.7 lists the best and worst physics parameters found by the GA corresponding to the best and worst fitness observed during the 500 generation run.

	Best	Worst
Gravity	$0.0 \frac{m}{s^2}$	$12.803362098213498 \frac{m}{s^2}$
Sub-steps	3	2
FPS	30	30
Material Friction	60.71902294177607	15.76102238218593
Material Elasticity	0.5378313234044673	0.602617926833965
Mass	4.175314301157847	0.2111916960575379
Velocity Max	660.0787581868401	787.7673658611162
Damping Translation	1.0	0.6703819812309364
Damping Rotation	0.4031179325185546	0.10076651150002103
Collision Bound Type	SPHERE	SPHERE
Torque	46.465508081185064	49.46737364841879
Fitness	1.0638026764	3257.00654843

TABLE 4.7: Here you see the best and worst physics parameters found by the GA corresponding to the best and worst fitness observed during the 500 generation run for experiment five.

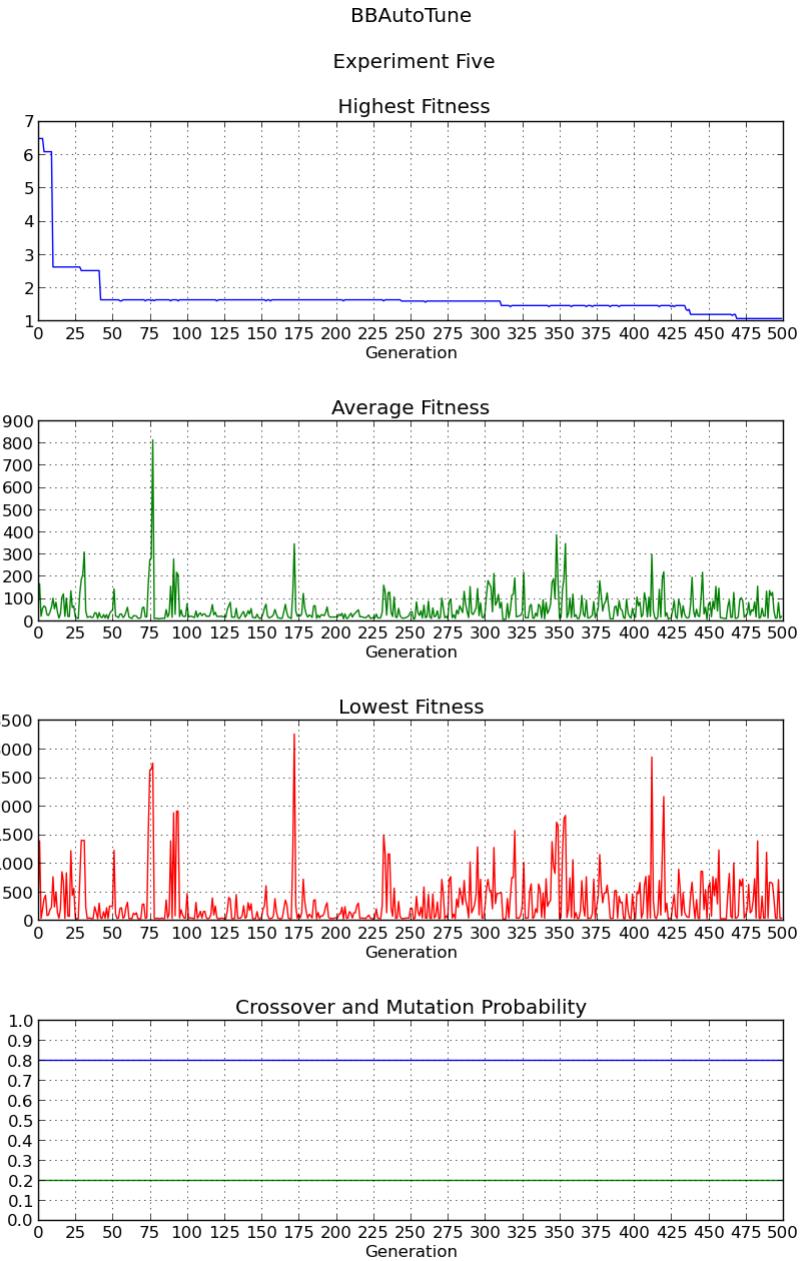


FIGURE 4.18: Here you see the highest, average, and lowest fitness in addition to the crossover and mutation probability over the course of 500 generations. For the bottom plot, the crossover probability is shown in blue while the mutation probability is shown in green.

4.6 Simulated versus Real Motion

The top phenotype for experiment two had the largest robust distance from the real robot robust mean but suffered the least amount of penalty for not stopping at one second. The top phenotypes for experiments three through five had similar robust distances and stopping times over one second. See Table 4.8, Figure 4.19, and Figure 4.20.

	Final States of Highest Performing Phenotypes				Real Robot Robust Mean
	Exp. Two	Exp. Three	Exp. Four	Exp. Five	
X-translation	23.12349975cm	23.95108938cm	23.49144816cm	24.05546605cm	23.9934044cm
Y-translation	0.00953689cm	-0.34897618cm	-0.00286334cm	0.01641194cm	0.0351240cm
Z-rotation	0.00203373rad	-0.00318596rad	-0.00428754rad	0.00118415rad	-0.0189964rad
Elapsed Time Until Coming to a Stop	1.0333563sec	1.5333535sec	1.5333476sec	1.5333496sec	
Resulting Genome Fitness	0.930619100106	1.0827696957	1.1309704845	1.0638026764	

TABLE 4.8: Here you see the comparison between the top phenotypes' final states (per experiment) and the robust mean of the real robot forward motion. The elapsed times indicate how long each top phenotype took before reaching its final at-rest state. The resulting genome fitnesses are included for reference.

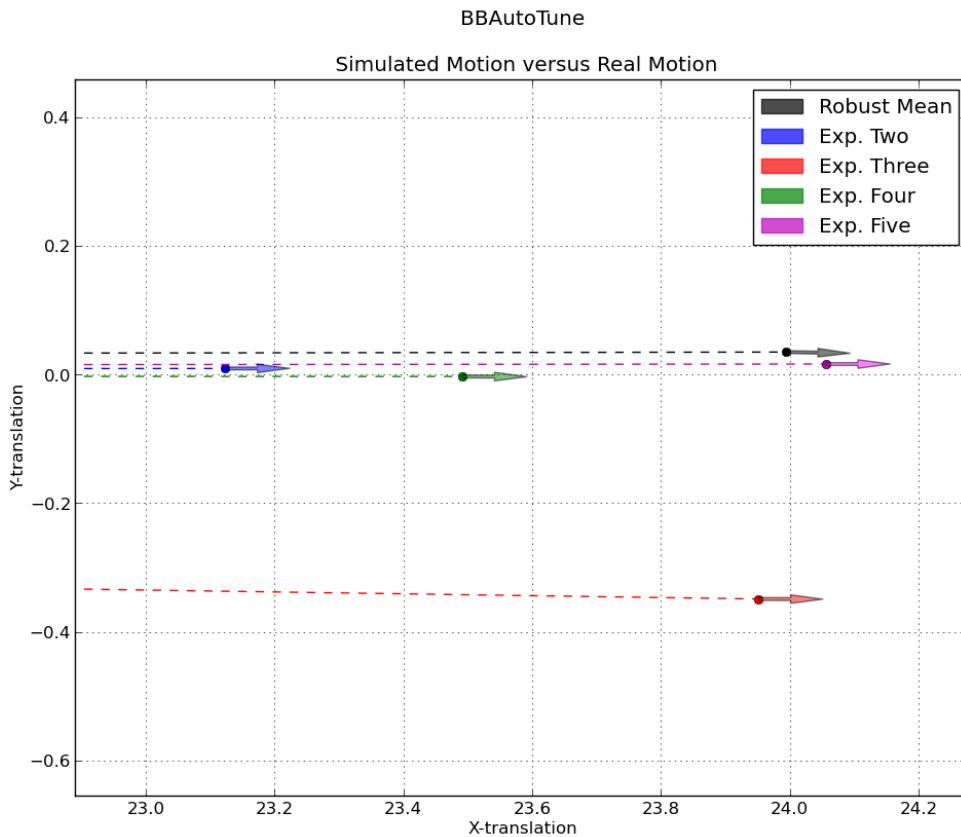


FIGURE 4.19: Here you see the final states of the top phenotypes from experiments two through five compared to the real robot robust mean.

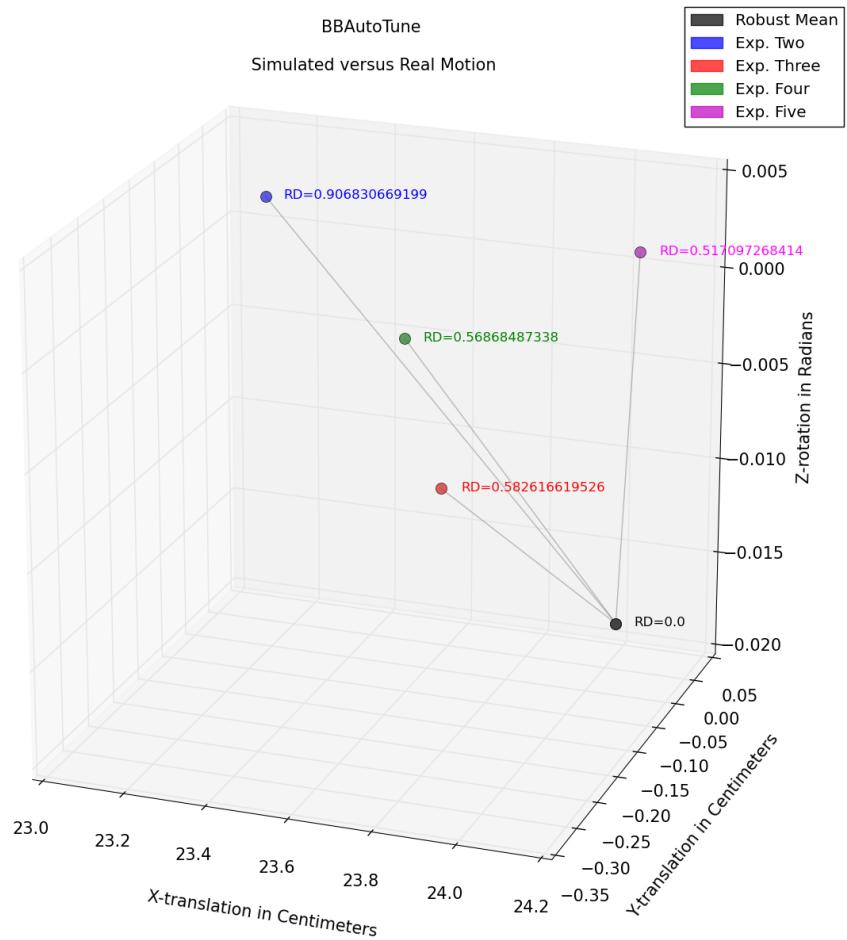


FIGURE 4.20: Here you see an alternate view of the final states of the top phenotypes from experiments two through five compared to the real robot robust mean. The robust distances are indicated near each data point.

Chapter 5

BlenderSim

5.1 Overview

Over the course of three months, a Blender based simulation was developed for HRTeam¹ titled BlenderSim.

Initially, BlenderSim was wholly physics based but soon proved to be problematic on three fronts: time, scale, and intricacy. The problems faced by using the physics engine to model the locomotion of the robots gave way to the thesis of tuning the physics engine via a genetic algorithm. In the interim, the physics engine was abandoned and a constant linear motion model was used.

As a proof-of-concept of the thesis, BlenderSim was revisited with the physics engine being restored as the motion model—using the highest-fitness physics parameters found by BBAutoTune. Using only one robot, instead of the standard four usually used in a HRTeam experiment, the motion of the simulated, physics-based robot was compared to the real robot by re-running a previously run HRTeam experiment in BlenderSim.

5.2 Problems Faced

Initial problems arose when the treads of the SRV-1 were recreated in the simulation. Even after numerous hours adjusting physics parameters and rigid-body configurations, the treads would consistently behave in erratic fashions. See Figure 5.1.

Scale was problematic as the Blender/Bullet physics engine has difficulty with collisions of objects that have a size outside of the assumed range of .05 to 10 meters [22]. Objects smaller than .05 (5cm)

¹HRTeam is a human and robot multi-agent framework developed to explore human and robot teamwork interactions [21].

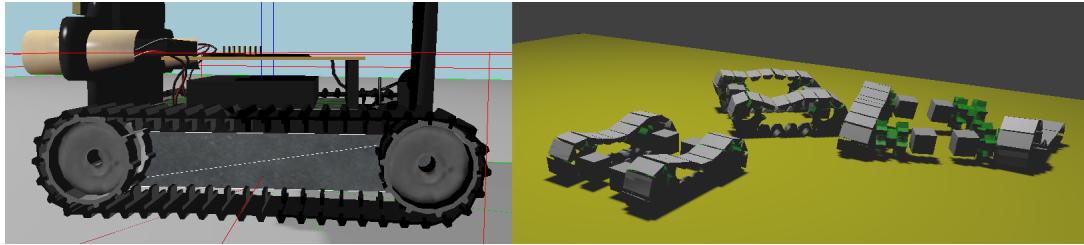


FIGURE 5.1: Here you see the to-scale treads modeled after the physical SRV-1 robot on the left and the physics-based rigid-body tracks in motion on the right.

Blender/Bullet units, in any given dimension, erratically jitter despite having no force acting upon them. As such, since the wheel dimensions of the real SRV-1 are 2.11cm x 2.45cm x 2.52cm, the to-scale 3D model of the SRV-1 was affected by this scale limitation of the physics engine.

To rectify these issues, the physics engine was largely abandoned—in terms of providing the motion model—and was only kept to keep the robots from running through each other and the arena. In its place, a constant linear and angular velocity motion model was developed of which only moves the 3D robot as if it was a single point body. See Figure 5.2 and Figure 5.3.

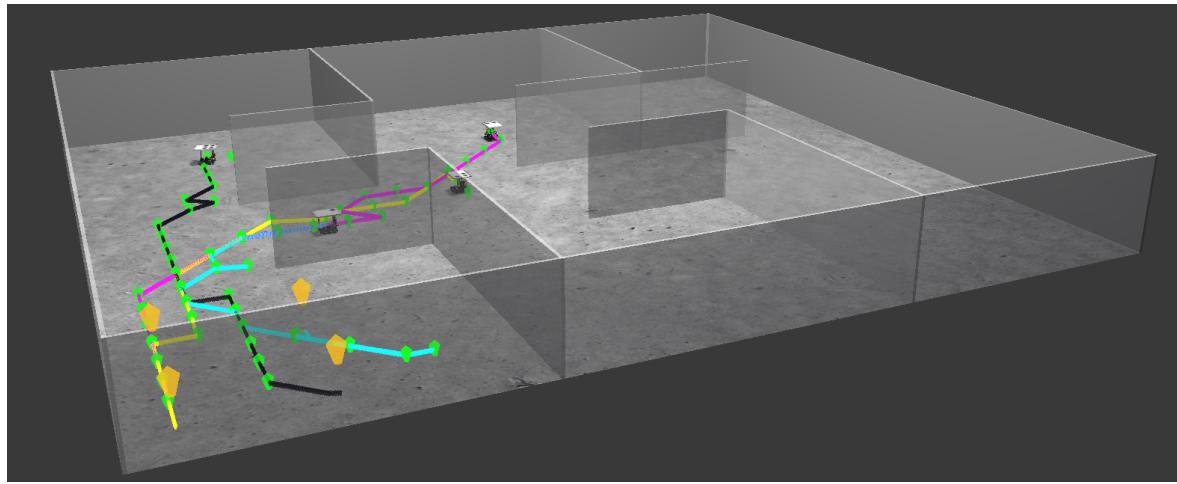


FIGURE 5.2: Here you see the constant velocity motion model at work in BlenderSim with four robots traversing their respective paths of which consist of way-points scrapped from a previously recorded physical lab experiment.

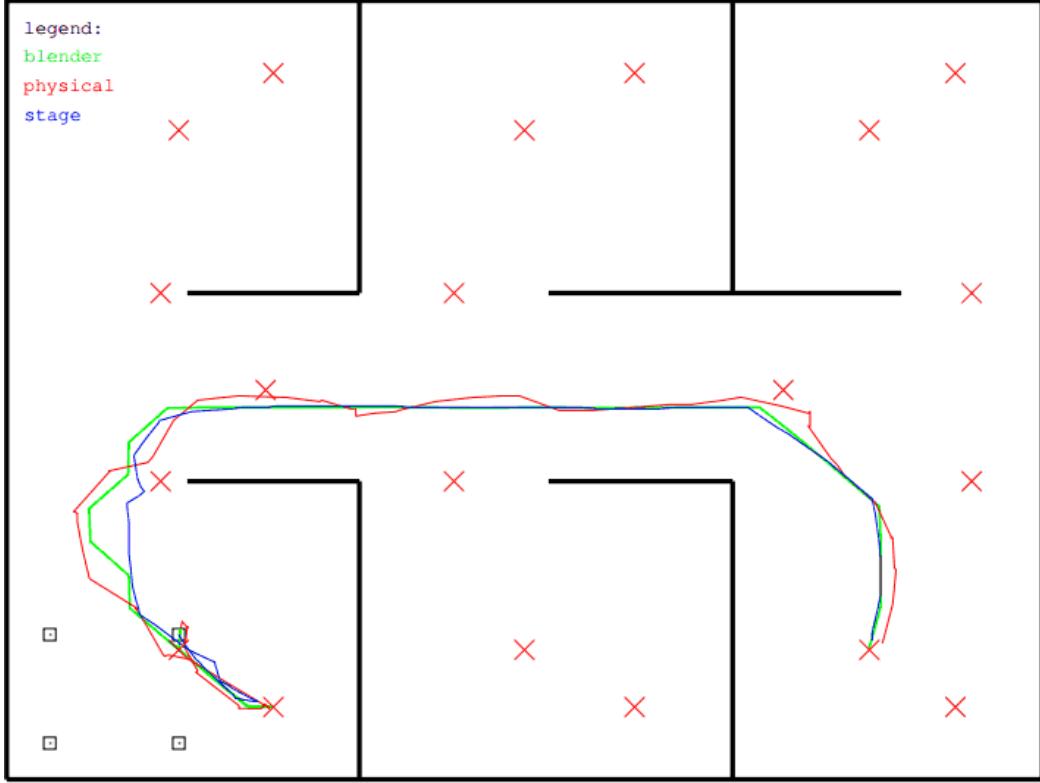


FIGURE 5.3: Here you see the path plots—as plotted by Dr. Elizabeth Sklar—of the BlenderSim robot, the physical robot, and the *Stage* (a 2D robot simulator already in use by HRTeam)[23] robot plotted in comparison.

5.3 Implementation

For the purposes of the proof-of-concept, most of the original architecture of BlenderSim was simplified. The simulation contained one robot, one robot controller, one robot path planner, and the arena.

5.3.1 Arena

The arena is a $602cm \times 538cm$ enclosure with a main hallway and six compartments. The floor and all of walls are physics based in BlenderSim and respond accordingly should any robot try to pass through them. See Figure 5.4. All dimensions and proportions of the simulated arena match the real arena used in HRTeam experiments.

5.3.2 Surveyor SRV-1 Blackfin 3D Models

The 3D model of the Surveyor SRV-1 Blackfin is dimensionally and aesthetically representative of its real-life counterpart. The extents of the model are $17.64cm \times 14.53cm \times 14.33cm$ including the braille hat that rests above the body of the robot. The base and the wheels are the only physics based objects on the model with the wheels being connected to the base via a rigid body hinge joint. See Figure 5.5. Residing $1.37cm$ off the floor from their local origins in the positive z-axis, the robot's base and wheels have their z-translation fixed. The physics parameters governing the wheels are the same highest-fitness parameters found by BBAutoTune in subsection 4.5.1.

5.3.3 Robot Path Planner

The robot path planner was greatly simplified for the proof-of-concept. Given a HRTeam experiment log and a robot number, the path planner extracts the A* path points that the real robot was instructed to follow during the HRTeam experiment. This A* path is a path from the robot's starting position to various points of interest or task points. With the path points extracted from the log file, the robot path planner populates the robot controller's waypoint queue.

5.3.4 Robot Controller

The robot controller is expressed in the simulation as the metal box base of the SRV-1 3D model. See Figure 5.6. Contained in its logic is a path or waypoint queue that it runs through. This queue is populated by the

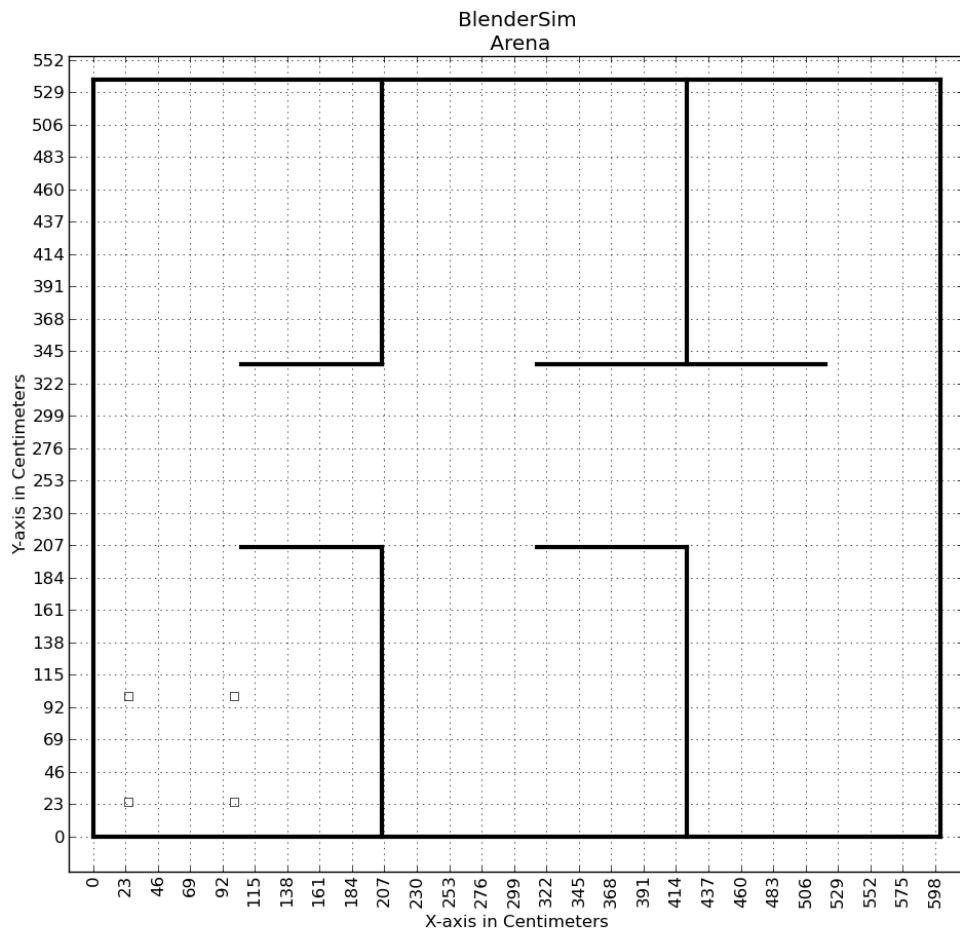


FIGURE 5.4: Here you see the layout and dimensions of the arena map both as it is in BlenderSim and in the real HRTeam lab. The smaller four squares in the lower left-hand corner represent the standard starting positions of the robots before any experiment. The grid shown in the plot is the same discrete grid used by HRTeam to calculate the A^* generated paths of the robots.

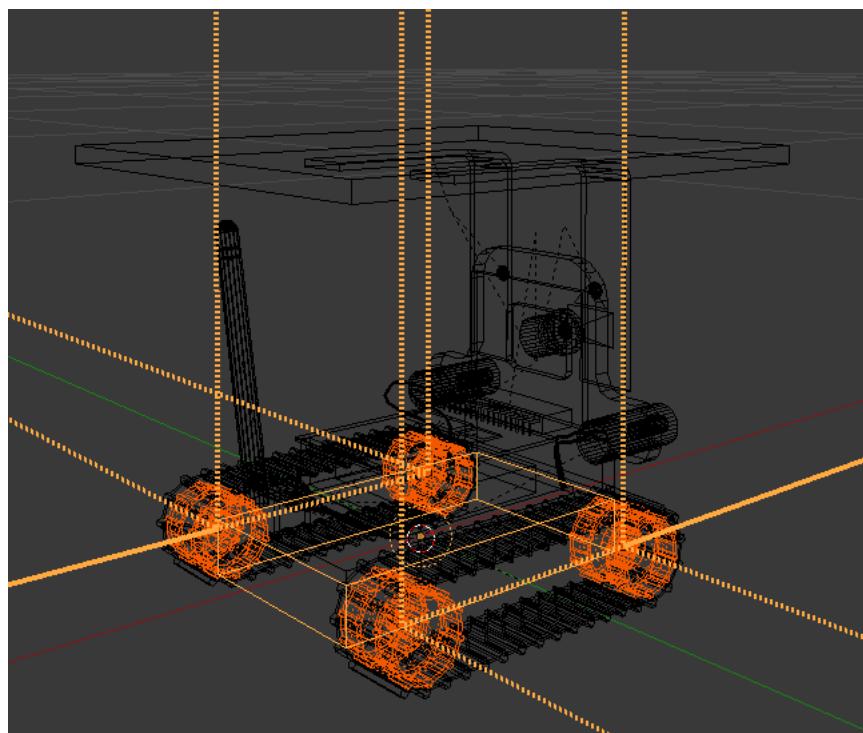


FIGURE 5.5: Here you see the 3D model of the Surveyor SRV-1 Blackfin with the wheels connected to the base via rigid body hinge joints.

robot path planner. For each waypoint in the queue, the robot controller will first rotate the robot towards the point and then move the robot forward to reach the point. Once the robot moves forward, the waypoint is removed from the queue.

For either rotating or moving the robot, the robot controller can only apply torque to the robot's wheels—no other force or mechanism is used to rotate or move the robot. Recalling from subsection 4.5.1 and section 4.6, a torque setting of 82.7271515601 resulted in the simulated robot moving forward 23.12349975cm. Since turning had not been learned by BBAutoTune, there were no baseline values of torque versus rotation. To correct for this, the torque value for the robot's left wheels was set to -20.8 while the torque value for the right wheels was set to 20.8 resulting in an in-place rotation of 44.260811 degrees.

Using these torque versus displacement values, the robot controller linearly interpolates the amount of torque needed to first rotate the robot towards a waypoint (to within 1 degree) and then move the robot forward to reach the waypoint. While rotating, if the robot over or undershoots the orientation needed to face the waypoint, the robot controller will continuously interpolate the torque needed to get the robot facing the waypoint to within 1 degree. Depending on the sign of the angle needed to orient the robot towards the waypoint, the robot controller will either rotate the robot counter-clockwise or clockwise by applying the same magnitude but opposing signs of torque to either the left or right wheels. While move forward, if the robot over or undershoots the position of the waypoint, the robot controller performs no correction but proceeds to orient and move the robot towards the next waypoint in the queue (if there is one). Note that for moving the robot forward, the robot controller applies the same torque value to all four wheels.

5.3.5 Task Points Manager

The task points manager is expressed in the simulation as a large torus located above the 3D modeled arena. Once the Blender game engine is started, it hides itself. See Figure 5.6. At the start of the simulation, the task points manager reads the task points configurations from the task points configuration directory. Controls include keys **a** through **e** where each key corresponds to the five possible task point configurations. See Figure 5.7.

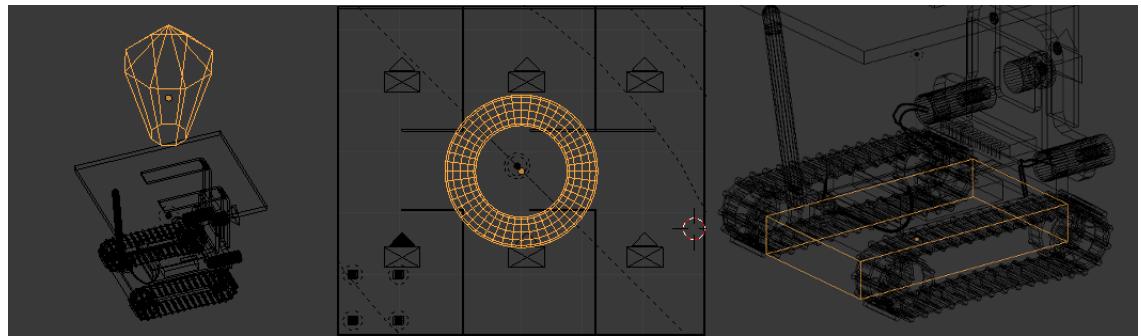


FIGURE 5.6: Here you see from left to right: the robot path planner, the task points manager, and the robot controller manifestations in the simulation.

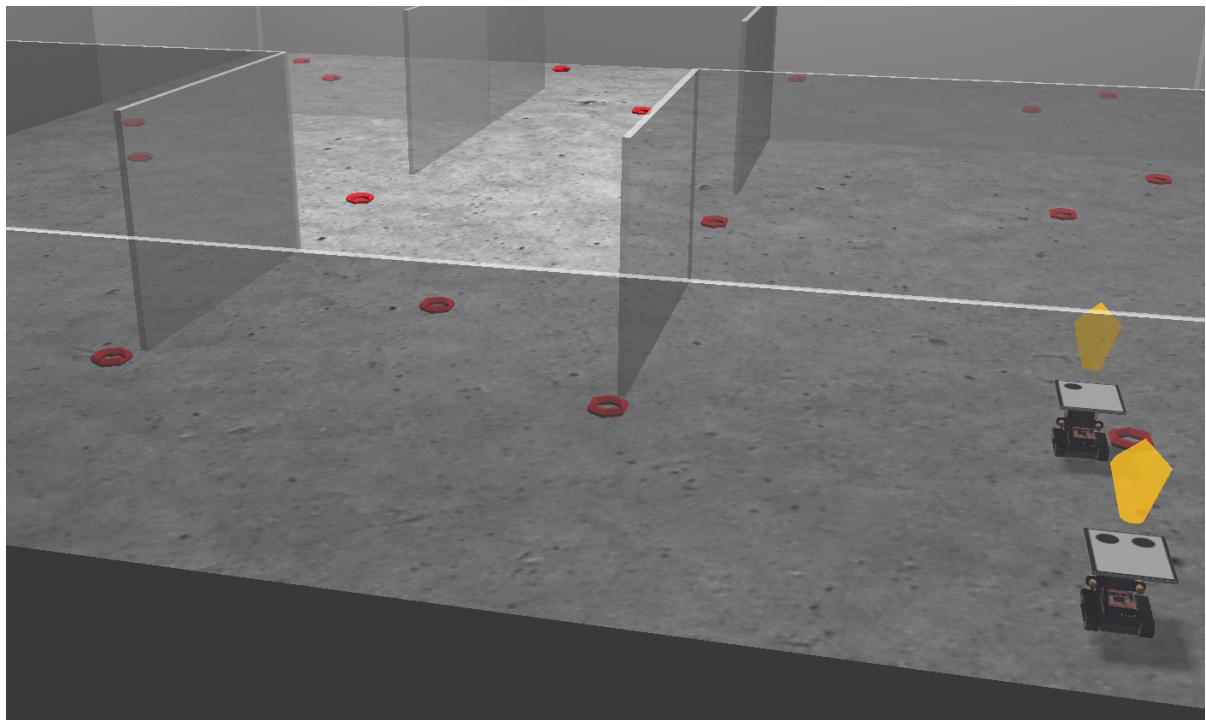


FIGURE 5.7: Here you see the red torus task points laid out in the arena.

5.4 Platform

BlenderSim was run on a 64bit Linux operating system with 32GB of RAM and an Intel Core i7-4770K four core processor running at 3.9GHz.

5.5 Experimental Design

To compare the simulated versus real robot motion (using the physics engine and the highest-fitness physics parameters found by BBAutoTune), a HRTeam experiment was re-run in BlenderSim but for only one robot—specifically robot one with a starting position at $(100cm, 100cm)$. The A* path (the waypoints) and the task points that the real robot one was instructed to travel along were extracted from the experiment’s log file. Positioning the simulated robot at the same starting position as the real robot in the experiment, the simulated robot was instructed to travel along the same set of waypoints and task points as the real robot was instructed to. See Figure 5.8.

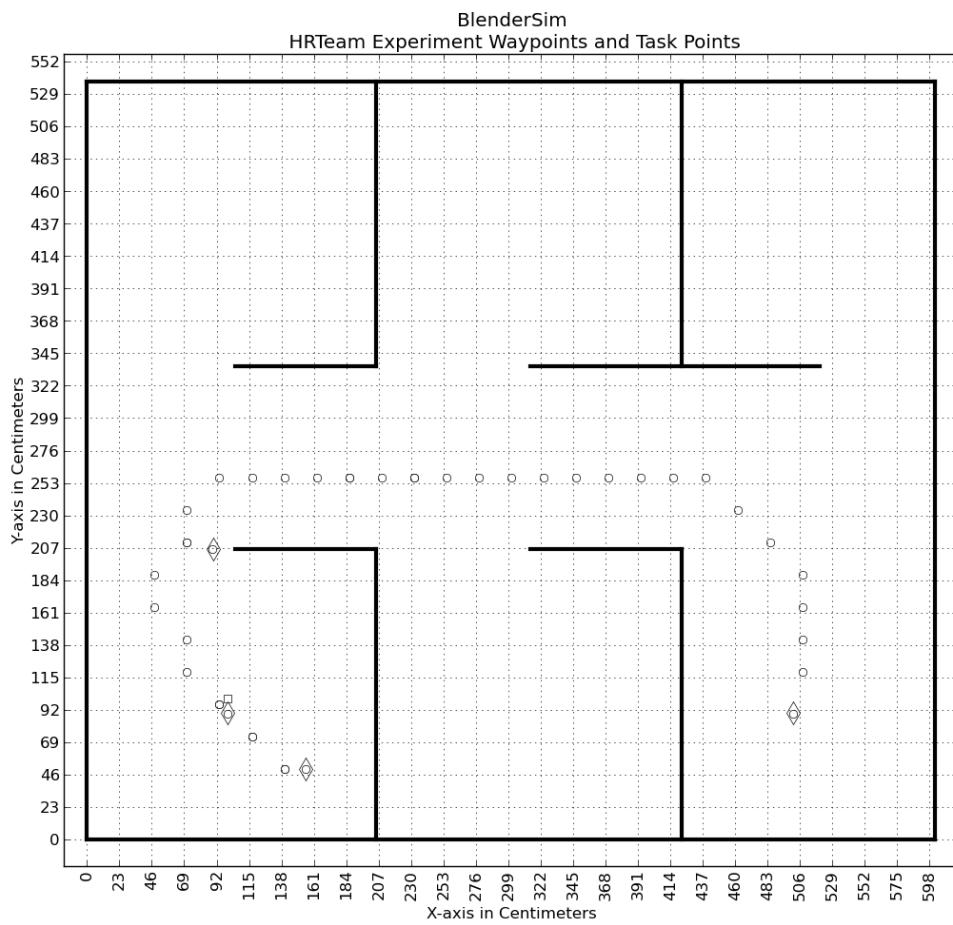


FIGURE 5.8: Here you see the arena layout with the experiment's waypoints and task points. The white square is the initial position of the robot before the experiment. The white circles are the waypoints and the white diamonds are the task points.

5.6 Experimental Result

Comparing the simulated versus real robot paths, the discrete Fréchet and Hausdorff distances between them were 40.575953633cm and 19.1208685021cm respectively. Between the simulated and real robot paths, the real robot path was the most dissimilar from the waypoint path. See Table 5.1, 5.2, and Figure 5.9.

	Simulated	Real	Waypoint
Simulated	0.0cm	40.575953633cm	16.2376801625cm
Real	40.575953633cm	0.0cm	24.0208242989cm
Waypoint	16.2376801625cm	24.0208242989cm	0.0cm

TABLE 5.1: Here you see the discrete Fréchet distances between the various paths.

	Simulated	Real	Waypoint
Simulated	0.0cm	19.1208685021cm	16.2376801625cm
Real	19.1208685021cm	0.0cm	24.0208242989cm
Waypoint	16.2376801625cm	24.0208242989cm	0.0cm

TABLE 5.2: Here you see the Hausdorff distances between the various paths.

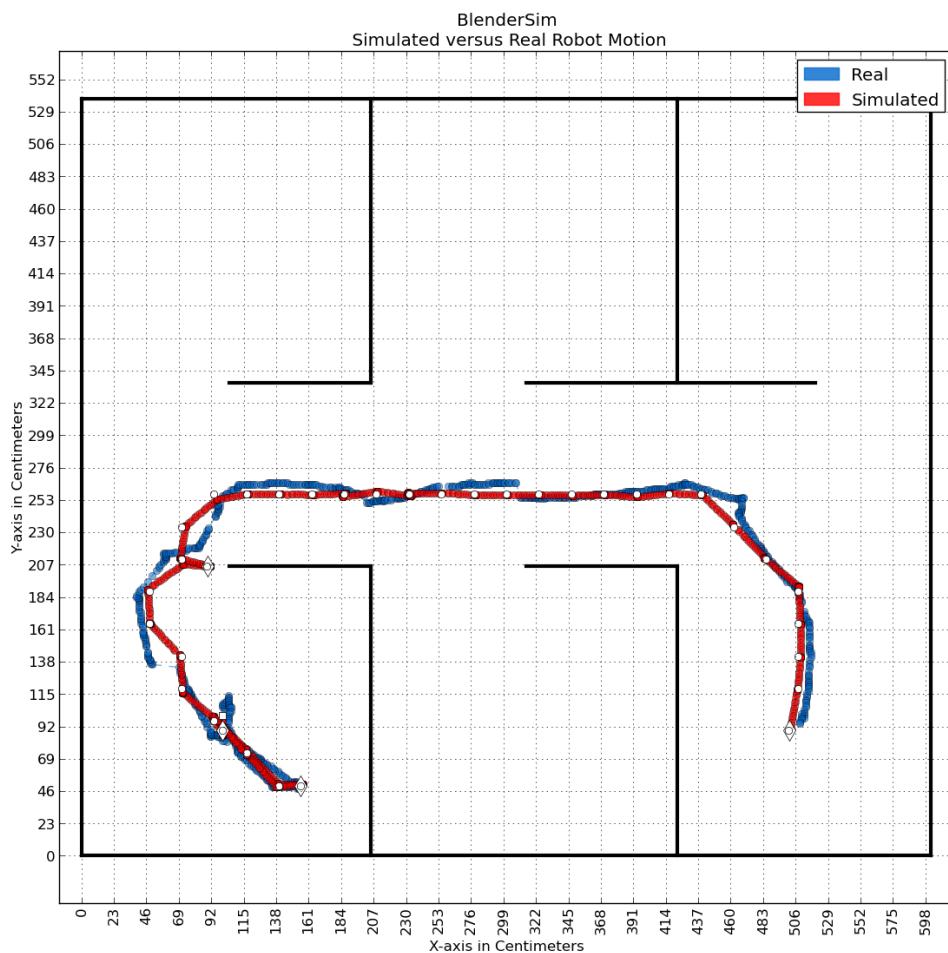


FIGURE 5.9: Here you see the simulated motion versus the real motion of the robot over the course of the experiment.

Chapter 6

Conclusion

6.1 Genetic Algorithms

The genetic algorithm outlined in this thesis employed a wide variety of techniques presented by previous works. Unfortunately, there is no unanimously recognized theory of genetic algorithms [4]. Furthermore, there is no proven optimal set of GA parameters for any given problem being solved by a GA [3]. There are, however, a few schools of thought and generally accepted guidelines when developing a GA [3][4]. As to why GAs work, a few ideas have been presented such as John Holland's schema theorem and David Goldberg's building block hypothesis [4].

6.2 SimPL

The genetic algorithm developed for SimPL proved to be a robust basis for the genetic algorithm needed to solve the harder problem of tuning a 3D physics engine (project BBAutoTune). The principles and techniques of evolutionary algorithms learned during the SimPL project certainly carried over to the more difficult project, BBAutoTune. And while the problem domain of SimPL and BBAutoTune were only somewhat similar, the problems faced and worked out during the development of SimPL alleviated problems faced while developing BBAutoTune. As the results show, the genetic algorithm for SimPL performed well, producing neural network weight solutions that had the paddle keeping the ball in the arena for almost a minute. Had it not been for the round termination criteria of the ball's velocity magnitude dropping below 100, most of the paddles (with high fitnesses) would have kept the ball in the arena indefinitely. Thus, the goal to learn about and to cultivate a genetic algorithm capable of tuning parameters with respect to a

fitness landscape was certainly accomplished.

6.3 BBAutoTune

Using the real robot forward motion data, BBAutoTune was consistently able to tune the physics engine such that the reality gap between simulation and reality was extremely small. For all runs of the GA, BBAutoTune was able to find nearly optimal physics parameters in no more than five hours. This is particularly impressive considering the state space size of the number of floating point numbers representable in the range [0.0, 1.0] on a 64-bit platform to the power of 12—12 being the number of tunable parameters. It is even more impressive considering the many days lost trying to find reasonable parameters by hand only to abandon the physics engine altogether due to consistent instability issues. Interestingly, the physics parameters found were not necessarily intuitive nor did they coincide with their real world counterparts. For example, in experiment two, gravity was $\sim 2.89 \frac{m}{s^2}$ versus earth's gravitational constant $\sim 9.81 \frac{m}{s^2}$ and the collision bounds type was sphere versus the cylindrical shape of the robot's wheels.

6.4 BlenderSim

Interpreting the plot of the simulated versus real robot motion along with the Fréchet and Hausdorff distances, one can see that the simulated motion was very close to the real robot motion. Considering the largest move the (simulated or real) robot can make at anyone time on the discrete arena grid is $\sim 32.52cm^1$ and that the largest distance between the simulated and real robots paths was a Fréchet distance of $\sim 40.57cm$, the thesis was certainly demonstrated and its hypothesis was supported.

¹Overlaid on the robot arena is a grid spaced $23cm \times 23cm$ for both the simulated and real arenas. This discrete grid is used to compute the A* paths that take the robots from their starting positions to the task points in the arena. Moving diagonally from one grid square to another requires a distance of $\sqrt{23^2cm + 23^2cm} = 32.526911935cm$.

Appendix A

Software Code

A.1 SimPL

LISTING A.1: Index.js

```
1  /*
2   *
3   * David Lettier (C) 2013.
4   *
5   * http://www.lettier.com/
6   *
7   * Simple Pong Learner or SimPL.
8   *
9   * Single sided pong clone with only one paddle which learns to follow and collide with the ball.
10  *
11 */
13 var request_animation_id = null;
15 var take_control = false;
16 var pause = true;
17 var show_debug = false;
19 var use_neural_network_paddle = true;
20 var use_perfect_paddle = false;
21 var use_random_paddle = false;
23 var run_experiment = true;
25 var top_wall = new Static_Object( "top_wall" );
26 var right_wall = new Static_Object( "right_wall" );
27 var bottom_wall = new Static_Object( "bottom_wall" );
28 var left_wall = new Static_Object( "left_wall" );
29
30 var ball_slot = document.getElementById( "ball_slot" );
31 var paddle_slot = document.getElementById( "paddle_slot" );
33 var ball_reset_center = { x: ball_slot.offsetLeft + ( ( ball_slot.offsetWidth || ball_slot.
34   clientWidth ) / 2 ),
35   y: ball_slot.offsetTop + ( ( ball_slot.offsetHeight || ball_slot.
36   clientHeight ) / 2 ) };
35 var paddle_reset_center = { x: paddle_slot.offsetLeft + ( ( paddle_slot.offsetWidth || paddle_slot.
37   clientWidth ) / 2 ),
38   y: paddle_slot.offsetTop + ( ( paddle_slot.offsetHeight || paddle_slot.
39   clientHeight ) / 2 ) };
40
41 var paddle_path = document.getElementById( "paddle_path" );
42 var ball_in_paddle_path_color = "rgba( 200, 255, 136, .2 )";
43 var ball_not_in_paddle_path_color = "rgba( 255, 98, 98, .2 )";
44 var paddle_path_background_color = ball_in_paddle_path_color;
45
46 var random_angle_range = { min: 135, max: 225 };
47 var random_angle = random_angle_range.min + ( random_angle_range.max - random_angle_range.min ) * Math.
48   random( );
49
50 var starting_magnitude = 1000.0;
51 var starting_ball_magnitude = starting_magnitude;
52 var starting_paddle_magnitude = starting_magnitude;
53 var starting_paddle_angle = 90.0;
54
55 var ball = new Physics_Object( new Dynamic_Object( "ball" ), random_angle, starting_ball_magnitude );
56 var paddle = new Physics_Object( new Dynamic_Object( "paddle", [ paddle_path, "t" ] ),
57   starting_paddle_angle, starting_paddle_magnitude );
58
59 var physics_engine = new Physics_Engine( collision_handler, paddle, ball, top_wall, right_wall,
60   bottom_wall, left_wall );
61
62 var time_delta = 0.0;
```

```

59 var fps_monitor = new FPS_Monitor( );
61 var debug_manager = new Debug_Manager( "debug" );
63 var database_manager = new Database_Manager( );
65 var neural_net_parameters = {
67   nInputs:           6,
68   nOutputs:          1,
69   nHiddenLayers:     1,
70   nNeuronsPerHiddenLayer: 5,
71   bias:              -1
73 };
75 var genetic_algorithm_parameters = {
77   popSize:            10,
78   nGenesPerGenome:    null,
79   useRankFitness:      true,
80   useSelfAdaptation:  true,
81   pCrMuSeq:           false,
82   iCProb:              0.8,
83   iMProb:              0.2,
84   nElite:              2
85 };
87 var learner_parameters = {
89   neural_net:         neural_net_parameters,
90   genetic_algorithm:  genetic_algorithm_parameters
93 };
95 var learner = {
97   parameters:        learner_parameters,
98   neural_net:         null,
99   neural_net_output:  null,
100  genetic_algorithm: null,
101  evaluation_start_time: 0,
102  evaluation_end_time:   0,
103  evaluate_current_genome: false,
104  current_genome_fitness_tracking: new Array(),
105  current_genome_paddle_hits: 0,
106  current_genome_being_evaluated: 0
107 };
109 learner.neural_net = new Neural_Net( learner.parameters.neural_net );
111 learner.parameters.genetic_algorithm.nGenesPerGenome = learner.neural_net.get_number_of_weights();
113 learner.genetic_algorithm = new Genetic_Algorithm( learner.parameters.genetic_algorithm );
115 learner.neural_net_output = 0;
117 // Attempt to load in the last generation from the database.
119 var last_generation_data_string = "population_size=" + learner.genetic_algorithm.get_population_size();
121 last_generation_data_string += "&number_of_genes_per_genome=" + learner.genetic_algorithm.get_number_of_genes_per_genome();
123 database_manager.send_and_receive( "assets/scripts/retrieve_genomes.php", last_generation_data_string,
124                                     database_manager, "last_generation", false );
125 var last_generation = database_manager.responses[ "last_generation" ];
127 if ( last_generation.charAt( 0 ) != ";" )
128 {
129   if ( window.confirm( "Start from previous generation?" ) )
130   {
133   // String received from server is generation_number;crossover_probability;mutation_probability;
134   population_genes.
135   var generation_number = parseInt( last_generation.split( ";" )[ 0 ] );
137   learner.genetic_algorithm.set_generation_number( generation_number );
139   var crossover_probability = parseFloat( last_generation.split( ";" )[ 1 ] );
141   learner.genetic_algorithm.set_crossover_probability( crossover_probability );
143   var mutation_probability = parseFloat( last_generation.split( ";" )[ 2 ] );
145   learner.genetic_algorithm.set_mutation_probability( mutation_probability );
147   var population_genes = last_generation.split( ";" )[ 3 ];
149   population_genes = population_genes.split( "," );
151   for ( var i = 0; i < population_genes.length; ++i )
152   {
153     population_genes[ i ] = parseFloat( population_genes[ i ] );
155   }
157

```

```

159     learner.genetic_algorithm.replace_population_genes( population_genes );
160     // Set neural network weights to the first genome's genes.
161     learner.neural_net.put_weights( learner.genetic_algorithm.get_genome_genes( learner.
162         current_genome_being_evaluated ) );
163     pause = false;
164 }
165 else
166 {
167     learner.neural_net.put_weights( learner.genetic_algorithm.get_genome_genes( learner.
168         current_genome_being_evaluated ) );
169     pause = false;
170 }
171 }
172 else
173 {
174     learner.neural_net.put_weights( learner.genetic_algorithm.get_genome_genes( learner.
175         current_genome_being_evaluated ) );
176     pause = false;
177 }
178 }
179 else
180 {
181     learner.neural_net.put_weights( learner.genetic_algorithm.get_genome_genes( learner.
182         current_genome_being_evaluated ) );
183     pause = false;
184 }
185 var ball_style = window.getComputedStyle( ball.dynamic_object.object, null );
186 var ball_style_transform = ball_style.getPropertyValue( "-webkit-transform" ) ||
187     ball_style.getPropertyValue( "-moz-transform" ) ||
188     ball_style.getPropertyValue( "-ms-transform" ) ||
189     ball_style.getPropertyValue( "-o-transform" ) ||
190     ball_style.getPropertyValue( "transform" );
191 var ball_div_transform_angle = 360;
192 var ball_div_transform_angle_by = 20;
193
194 function draw( time_stamp )
195 {
196     request_animation_id = window.requestAnimationFrame( draw );
197     time_delta = fps_monitor.get_time_delta( time_stamp );
198     if ( !pause )
199     {
200         debug_manager.add_or_update( "FPS", fps_monitor.get_fps( time_stamp ) );
201         debug_manager.add_or_update( "Paddle", paddle );
202         debug_manager.add_or_update( "Ball", ball );
203         handle_neural_network_ouput();
204         debug_manager.add_or_update( "Current Genome", learner.current_genome_being_evaluated );
205         debug_manager.add_or_update( "NN", learner.neural_net );
206         debug_manager.add_or_update( "NN Weights", learner.neural_net.get_weights() );
207         debug_manager.add_or_update( "GA", learner.genetic_algorithm );
208         physics_engine.update( time_delta );
209         // Spin the ball div to give the illusion of flying through the air.
210         if ( ball_style_transform != undefined && ball.get_magnitude() != 0 )
211         {
212             ball.dynamic_object.object.style.webkitTransform = "rotate(" + ball_div_transform_angle + "deg)";
213             ball.dynamic_object.object.style.MozTransform = "rotate(" + ball_div_transform_angle + "deg)";
214             ball.dynamic_object.object.style.msTransform = "rotate(" + ball_div_transform_angle + "deg)";
215             ball.dynamic_object.object.style.OTransform = "rotate(" + ball_div_transform_angle + "deg)";
216             ball.dynamic_object.object.style.transform = "rotate(" + ball_div_transform_angle + "deg)";
217             if ( ball.get_angle() >= 90 && ball.get_angle() <= 270 )
218             {
219                 // Spin counter clock-wise if it is going towards the left of the screen.
220                 ball_div_transform_angle = ball_div_transform_angle < 0 ? 360 : ball_div_transform_angle - (
221                     ball_div_transform_angle_by * ( ball.get_magnitude() / starting_ball_magnitude ) );
222             }
223             else
224             {
225                 ball_div_transform_angle = ball_div_transform_angle > 360 ? 0 : ball_div_transform_angle + (
226                     ball_div_transform_angle_by * ( ball.get_magnitude() / starting_ball_magnitude ) );
227             }
228         }
229     }
230     // Time to evaluate the current genome?
231     if ( learner.evaluate_current_genome )
232     {

```

```

255     handle_genome_evaluation( );
257   }
259   debug_manager.print( );
261 }
263 }
265 // The following stub was taken from https://gist.github.com/paulirish/1579671.
267 (
269 {
271   var lastTime = 0;
272   var vendors = ['ms', 'moz', 'webkit', 'o'];
273   for( var x = 0; x < vendors.length && !window.requestAnimationFrame; ++x )
274   {
275     window.requestAnimationFrame = window[ vendors[x] + 'RequestAnimationFrame' ];
276     window.cancelAnimationFrame = window[vendors[x]+ 'CancelAnimationFrame'] || window[ vendors[x] + 'CancelRequestAnimationFrame' ];
277   }
279   if ( !window.requestAnimationFrame )
281   {
283     window.requestAnimationFrame = function ( callback, element )
284   {
285     var currTime = new Date( ).getTime( );
286     var timeToCall = Math.max( 0, 16 - ( currTime - lastTime ) );
287     var id = window.setTimeout( function () { callback( currTime + timeToCall ); }, timeToCall );
288     lastTime = currTime + timeToCall;
289     return id;
290   };
293 }
295   if ( !window.cancelAnimationFrame )
297   {
299     window.cancelAnimationFrame = function ( id )
300   {
301     clearTimeout( id );
303   };
305 }
307 } ( ) );
309 draw( );
311 function reset( )
312 {
313   random_angle = random_angle_range.min + ( random_angle_range.max - random_angle_range.min ) * Math.
314   random( );
315   ball.dynamic_object.set_center( ball_reset_center.x, ball_reset_center.y );
316   ball.set_magnitude( starting_ball.magnitude );
317   ball.set_angle( random_angle );
318   paddle.dynamic_object.set_center( paddle_reset_center.x, paddle_reset_center.y );
319   paddle.set_magnitude( starting_ball.magnitude );
320   paddle.set_angle( starting_paddle_angle );
321
322   take_control = false;
323   pause = false;
324
325 }
327 document.onkeyup = handle_key;
329 function handle_key( kevent )
330 {
331   var key = ( window.event ) ? event.keyCode : kevent.keyCode;
333   switch ( key )
335   {
337     case 68: // [d] key.
339       show_debug = !show_debug
341       if ( show_debug ) debug_manager.show_debug( );
342       else debug_manager.hide_debug( );
343       break;
345     case 70: // [f] key.
347       debug_div = document.getElementById( "debug" );
349       debug_div.font_size = parseInt( window.getComputedStyle( debug_div, null ).getPropertyValue( 'font-
350 size' ), 10 );
351       if ( window.event.shiftKey )
352   {

```

```

355     debug_div_font_size -= 1;
357     debug_div.style.fontSize = debug_div_font_size + "px";
359 }
360 else
361 {
363     debug_div_font_size += 1;
365     debug_div.style.fontSize = debug_div_font_size + "px";
367 }
369     break;
371 case 77: // [m] key.
373     ball.set_magnitude( ball.get_magnitude() + 100 );
375     break;
377 case 80: // [p] key.
379     pause = !pause;
381     break;
383 case 82: // [r] key.
385     reset();
387     break;
389 case 84: // [t] key.
391     take_control = !take_control;
393     break;
395 }
397 document.onmousemove = handle_mouse_move;
399 function handle_mouse_move( mevent )
401 {
403     if ( !take_control ) return null;
405     mevent = ( window.event ) ? window.event : mevent;
407     if ( ( mevent.clientY - ( paddle.dynamic_object.get_height() / 2 ) ) >= ( window.innerHeight - paddle.
408         dynamic_object.get_height() ) )
409     {
410         paddle.dynamic_object.set_left_top( paddle.dynamic_object.get_left(), window.innerHeight - paddle.
411             dynamic_object.get_height() );
412         return null;
413     }
415     else if ( ( mevent.clientY - ( paddle.dynamic_object.get_height() / 2 ) ) <= 0 )
416     {
417         paddle.dynamic_object.set_left_top( paddle.dynamic_object.get_left(), 0 );
418         return null;
419     }
423     paddle.dynamic_object.set_left_top( paddle.dynamic_object.get_left(), mevent.clientY - ( paddle.
424         dynamic_object.get_height() / 2 ) );
425 }
427 function handle_neural_network_ouput()
429 {
431     if ( take_control )
432     {
433         paddle.set_magnitude( 0 );
435         paddle.set_angle( starting_paddle_angle );
437         return null;
439     }
441     // To help with fitness testing, keep track of if the ball
442     // is in the path of the paddle.
443     // 1 for in the path of the ball.
444     // Nothing for not in the path of the ball.
445     // -----
446     // In the path:
447     // -----
448     // |-----0
449     // |
450     // |-----
451     // |

```

```

453 // Not in the path:
454 // 0
455 // -----
456 // |
457 // |
458 // | -----
459 // |
460 // |
461 // -----
462 // |
463 // |
464 // | -----
465 // |
466 // 0
467

468 if ( !( ( ball.dynamic_object.get_top( ) > paddle.dynamic_object.get_bottom( ) ) || ( ball.
469     dynamic_object.get_bottom( ) < paddle.dynamic_object.get_top( ) ) ) )
470 {
471     learner.current_genome_fitness_tracking.push( 1 );
472
473     // Adjust paddle path visual.
474     paddle_path.style.background = ball_in_paddle_path_color;
475
476 }
477
478 }
479
480 else
481 {
482     // Add nothing.
483
484     // Adjust paddle path visual.
485     paddle_path.style.background = ball_not_in_paddle_path_color;
486
487 }
488
489 */
490
491 /**
492 // Experiment one tracking.
493 learner.current_genome_fitness_tracking.push( Math.abs( paddle.dynamic_object.get_center( ).y - ball.
494     dynamic_object.get_center( ).y ) );
495
496 */
497
498 // Get input values for the neural network.
499
500 var x = ball.dynamic_object.get_center( ).x - paddle.dynamic_object.get_center( ).x;
501 var y = ball.dynamic_object.get_center( ).y - paddle.dynamic_object.get_center( ).y;
502
503 var paddle_offset_from_ball = { x: x,
504     y: y,
505     h: Math.sqrt( ( x * x ) + ( y * y ) )
506 };
507
508 x = paddle.dynamic_object.get_center( ).x - 0;
509 y = paddle.dynamic_object.get_center( ).y - 0;
510
511 var paddle_offset_from_screen_origin = { x: x,
512     y: y,
513     h: Math.sqrt( ( x * x ) + ( y * y ) )
514 };
515
516 var ball_velocity = deep_copy( ball.get_velocity( ) );
517
518 // Normalize input in [-1,1].
519
520 paddle_offset_from_ball.x = paddle_offset_from_ball.x / paddle_offset_from_ball.h;
521 paddle_offset_from_ball.y = paddle_offset_from_ball.y / paddle_offset_from_ball.h;
522
523 paddle_offset_from_screen_origin.x = paddle_offset_from_screen_origin.x /
524     paddle_offset_from_screen_origin.h;
525 paddle_offset_from_screen_origin.y = paddle_offset_from_screen_origin.y /
526     paddle_offset_from_screen_origin.h;
527
528 // Turn the velocity into unit vectors.
529 ball_velocity.x = ball_velocity.x / ball.get_magnitude( );
530 ball_velocity.y = ball_velocity.y / ball.get_magnitude( );
531
532 // Send in the input and get out the output which is an array.
533
534 learner.neural_net_output = learner.neural_net.update( [ paddle_offset_from_ball.x,
535     paddle_offset_from_ball.y, ball_velocity.x, ball_velocity.y, paddle_offset_from_screen_origin.x,
536     paddle_offset_from_screen_origin.y ] );
537
537 debug_manager.add_or_update( "NN Input", paddle_offset_from_ball.x.toFixed( 3 ) + " " +
538     paddle_offset_from_ball.y.toFixed( 3 ) + " " + ball_velocity.x.toFixed( 3 ) + " " + ball_velocity.y.
539    toFixed( 3 ) + " " + paddle_offset_from_screen_origin.x.toFixed( 3 ) + " " +
540     paddle_offset_from_screen_origin.y.toFixed( 3 ) );
541
542 debug_manager.add_or_update( "NN Output", learner.neural_net_output );
543
544 if ( use_neural_network_paddle )
545 {
546     if ( learner.neural_net_output[ 0 ] < 0.0 )
547     {

```

```

547 // Output indicates that we must go down the screen
548 // by some magnitude in the range [0,MAG_MAX].
549 // Don't set a negative magnitude so multiply the output by -1 to
550 // make it positive and multiply this result by the starting paddle
551 // magnitude.
552 paddle.set_magnitude( starting_paddle_magnitude * ( -1 * learner.neural_net_output[ 0 ] ) );
553 paddle.set_angle( 270 );
554 }
555 else if ( learner.neural_net_output[ 0 ] == 0.0 )
556 {
557     // Don't move.
558     paddle.set_magnitude( 0.0 );
559     paddle.set_angle( starting_paddle_angle );
560 }
561 else if ( learner.neural_net_output[ 0 ] > 0.0 )
562 {
563     // Go up by some portion of the starting paddle magnitude.
564     paddle.set_magnitude( starting_paddle_magnitude * learner.neural_net_output[ 0 ] );
565     paddle.set_angle( 90.0 );
566 }
567 }
568 }
569 else if ( use_perfect_paddle )
570 {
571     /*
572     var difference = paddle.dynamic_object.get_center( ).y - ball.dynamic_object.get_center( ).y;
573     debug_manager.add_or_update( "Perfect Output", difference );
574     if ( difference < 0 )
575     {
576         paddle.set_magnitude( 10.0 * ( -1 * difference ) );
577         paddle.set_angle( 270 );
578     }
579     else if ( difference == 0.0 )
580     {
581         paddle.set_magnitude( 0.0 );
582         paddle.set_angle( starting_paddle_angle );
583     }
584     else if ( difference > 0 )
585     {
586         paddle.set_magnitude( 10.0 * difference );
587         paddle.set_angle( 90.0 );
588     }
589     */
590     paddle.set_magnitude( 0.0 );
591     paddle.set_angle( starting_paddle_angle );
592     var ball_center = ball.dynamic_object.get_center( );
593     var paddle_center = paddle.dynamic_object.get_center( );
594     paddle.dynamic_object.set_center( paddle_center.x, ball_center.y );
595 }
596 else if ( use_random_paddle )
597 {
598     var random_float = get_random_float( -1, 1 );
599     debug_manager.add_or_update( "Random Output", random_float );
600     if ( random_float < 0.0 )
601     {
602         paddle.set_magnitude( starting_paddle_magnitude * ( -1 * random_float ) );
603         paddle.set_angle( 270.0 );
604     }
605     else if ( random_float == 0.0 )
606     {
607         paddle.set_magnitude( 0.0 );
608         paddle.set_angle( starting_paddle_angle );
609     }

```

```

649     }
651 {
653     paddle.set_magnitude( starting_paddle_magnitude * random_float );
655     paddle.set_angle( 90.0 );
657 }
659 }
661 }

663 /* Experiment one fitness function.

665 function handle_genome_evaluation( )
{
667     pause = true;
669     var fitness = 0.0;
671     for( var i = 0; i < learner.current_genome_fitness_tracking.length - 1; ++i )
{
675         if ( learner.current_genome_fitness_tracking[ i ] < learner.current_genome_fitness_tracking[ i + 1 ] )
{
677             fitness = fitness - 0.1;
679         }
681         else if ( learner.current_genome_fitness_tracking[ i ] == learner.current_genome_fitness_tracking[ i + 1 ] )
{
683             if ( learner.current_genome_fitness_tracking[ i ] == 0 && learner.current_genome_fitness_tracking[ i + 1 ] == 0 )
{
685                 fitness = fitness + 0.1;
687             }
689             else
{
691                 fitness = fitness - 0.1;
693             }
695         }
697         else if ( learner.current_genome_fitness_tracking[ i ] > learner.current_genome_fitness_tracking[ i + 1 ] )
{
699             fitness = fitness + 0.1;
701         }
703     }
705 }

707 learner.current_genome_fitness_tracking = [ ];
709 fitness = fitness + ( 1 * learner.current_genome_paddle_hits );
711 learner.current_genome_paddle_hits = 0;
713 if ( fitness < 0.0 ) fitness = 0.0;
715 debug_manager.add_or_update( "Last Genome Fitness", fitness );
717 learner.genetic_algorithm.set_genome_fitness( learner.current_genome_being_evaluated, fitness );
719 if ( ( learner.current_genome_being_evaluated + 1 ) == learner.genetic_algorithm.get_population_size( )
)
{
721     // Evaluated all the genomes in the population.
723     // Calculate population metrics.
724     // Adjust crossover and mutation probabilities if using self-adaptation.
725     learner.genetic_algorithm.sort_population( );
727     learner.genetic_algorithm.evaluate_population( );
729     if ( learner.parameters.genetic_algorithm.useSelfAdaptation )
{
733         learner.genetic_algorithm.adjust_crossover_and_mutation_probabilities( );
735     }
737     // Package up this population and send it to the database if its generation number is higher than
738     // the highest generation number already in the database.
739     var data_string = "generation_number=" + learner.genetic_algorithm.get_generation_number( )
;
741     data_string += "&best_fitness=" + learner.genetic_algorithm.get_best_fitness( );
742     data_string += "&average_fitness=" + learner.genetic_algorithm.get_average_fitness( );
743     data_string += "&worst_fitness=" + learner.genetic_algorithm.get_worst_fitness( );
    data_string += "&population_size=" + learner.genetic_algorithm.get_population_size( );

```

```

745     data_string += "&number_of_genes_per_genome=" + learner.genetic_algorithm.
746         get_number_of_genes_per_genome( );
747     data_string += "&crossover_probability=" + learner.genetic_algorithm.
748         get_crossover_probability( );
749     data_string += "&mutation_probability=" + learner.genetic_algorithm.get_mutation_probability
750         ( );
751     data_string += "&population_genes=" + learner.genetic_algorithm.
752         get_population_genes_flattened( );
753
754     database_manager.send_and_receive( "assets/scripts/store_genomes.php", data_string, database_manager,
755         "adding_genome_population" );
756
757     if ( run_experiment )
758     {
759
760         // Experiment of first 100 generations.
761
762         if ( learner.genetic_algorithm.get_generation_number( ) < 100 )
763         {
764
765             var experiment_record_data_string = "action=record";
766             experiment_record_data_string += "&generation_number=" + learner.genetic_algorithm.
767                 get_generation_number( );
768             experiment_record_data_string += "&best_fitness=" + learner.genetic_algorithm.
769                 get_best_fitness( );
770             experiment_record_data_string += "&average_fitness=" + learner.genetic_algorithm.
771                 get_average_fitness( );
772             experiment_record_data_string += "&worst_fitness=" + learner.genetic_algorithm.
773                 get_worst_fitness( );
774             experiment_record_data_string += "&crossover_probability=" + learner.genetic_algorithm.
775                 get_crossover_probability( );
776             experiment_record_data_string += "&mutation_probability=" + learner.genetic_algorithm.
777                 get_mutation_probability( );
778
779             database_manager.send_and_receive( "assets/scripts/experiment.php", experiment_record_data_string,
780                 database_manager, "experiment_record" );
781
782             if ( ( learner.genetic_algorithm.get_generation_number( ) + 1 ) % 10 == 0 )
783             {
784
785                 var experiment_store_data_string = "action=store";
786                 experiment_store_data_string += "&generation_number=" + learner.genetic_algorithm.
787                     get_generation_number( );
788                 experiment_store_data_string += "&fitness=" + learner.genetic_algorithm.
789                     get_genome_fitness( learner.genetic_algorithm.get_fittest_genome_index( ) );
790                 experiment_store_data_string += "&crossover_probability=" + learner.genetic_algorithm.
791                     get_crossover_probability( );
792                 experiment_store_data_string += "&mutation_probability=" + learner.genetic_algorithm.
793                     get_mutation_probability( );
794                 experiment_store_data_string += "&genes=" + learner.genetic_algorithm.
795                     get_genome_genes_flattened( learner.genetic_algorithm.get_fittest_genome_index( ) );
796
797                 database_manager.send_and_receive( "assets/scripts/experiment.php", experiment_store_data_string
798                     , database_manager, "experiment_store" );
799             }
800         }
801     }
802     else
803     {
804
805         // Test the next genome in the population.
806
807         learner.current_genome_being_evaluated += 1;
808
809         learner.neural_net.put_weights( learner.genetic_algorithm.get_genome_genes( learner.
810             current_genome_being_evaluated ) );
811     }
812
813     learner.evaluate_current_genome = false;
814
815     reset( );
816
817 }
818 */
819
820     function handle_genome_evaluation( )
821     {
822
823         pause = true;

```

```

827 var fitness = 0.0;
829 for( var i = 0; i < learner.current_genome_fitness_tracking.length; ++i )
{
831   fitness += learner.current_genome_fitness_tracking[ i ];
833 }
835 learner.current_genome_fitness_tracking = [ ];
837 if ( fitness < 0.0 ) fitness = 0.0;
839 debug_manager.add_or_update( "Last Genome Fitness", fitness );
841 learner.genetic_algorithm.set_genome_fitness( learner.current_genome_being_evaluated, fitness );
843 if ( ( learner.current_genome_being_evaluated + 1 ) == learner.genetic_algorithm.get_population_size( )
)
845 {
847   // Evaluated all the genomes in the population.
848   // Calculate population metrics.
849   // Adjust crossover and mutation probabilities if using self-adaptation.
851   learner.genetic_algorithm.sort_population( );
853   learner.genetic_algorithm.evaluate_population( );
855 if ( learner.parameters.genetic_algorithm.useSelfAdaptation )
{
857   learner.genetic_algorithm.adjust_crossover_and_mutation_probabilities( );
859 }
861 // Package up this population and send it to the database if its generation number is higher than
863 // the highest generation number already in the database.
865 var data_string = "generation_number=" + learner.genetic_algorithm.get_generation_number( )
;
866 data_string += "&best_fitness=" + learner.genetic_algorithm.get_best_fitness( );
867 data_string += "&average_fitness=" + learner.genetic_algorithm.get_average_fitness( );
868 data_string += "&worst_fitness=" + learner.genetic_algorithm.get_worst_fitness( );
869 data_string += "&population_size=" + learner.genetic_algorithm.get_population_size( );
870 data_string += "&number_of_genes_per_genome=" + learner.genetic_algorithm.
get_number_of_genes_per_genome( );
871 data_string += "&crossover_probability=" + learner.genetic_algorithm.
get_crossover_probability( );
872 data_string += "&mutation_probability=" + learner.genetic_algorithm.get_mutation_probability
();
873 data_string += "&population_genes=" + learner.genetic_algorithm.
get_population_genes_flattened( );
875 database_manager.send_and_receive( "assets/scripts/store_genomes.php", data_string, database_manager,
"adding_genome_population" );
877 if ( run_experiment )
{
879   // Experiment of first 100 generations.
881   if ( learner.genetic_algorithm.get_generation_number( ) < 100 )
{
885     var experiment_record_data_string = "action=record";
886     experiment_record_data_string += "&generation_number=" + learner.genetic_algorithm.
get_generation_number( );
887     experiment_record_data_string += "&best_fitness=" + learner.genetic_algorithm.
get_best_fitness( );
888     experiment_record_data_string += "&average_fitness=" + learner.genetic_algorithm.
get_average_fitness( );
889     experiment_record_data_string += "&worst_fitness=" + learner.genetic_algorithm.
get_worst_fitness( );
890     experiment_record_data_string += "&crossover_probability=" + learner.genetic_algorithm.
get_crossover_probability( );
891     experiment_record_data_string += "&mutation_probability=" + learner.genetic_algorithm.
get_mutation_probability( );
893     database_manager.send_and_receive( "assets/scripts/experiment.php", experiment_record_data_string,
database_manager, "experiment_record" );
895     if ( ( learner.genetic_algorithm.get_generation_number( ) + 1 ) % 10 == 0 )
{
897       var experiment_store_data_string = "action=store";
898       experiment_store_data_string += "&generation_number=" + learner.genetic_algorithm.
get_generation_number( );
899       experiment_store_data_string += "&fitness=" + learner.genetic_algorithm.
get_genome_fitness( learner.genetic_algorithm.get_fittest_genome_index( ) );
900       experiment_store_data_string += "&crossover_probability=" + learner.genetic_algorithm.
get_crossover_probability( );
901       experiment_store_data_string += "&mutation_probability=" + learner.genetic_algorithm.
get_mutation_probability( );
903       experiment_store_data_string += "&genes=" + learner.genetic_algorithm.
get_genome_genes_flattened( learner.genetic_algorithm.get_fittest_genome_index( ) );
905       database_manager.send_and_receive( "assets/scripts/experiment.php", experiment_store_data_string
,
database_manager, "experiment_store" );
907 }

```

```

909     }
911   }
913   // Generate a new generation.
915   learner.genetic_algorithm.generate_new_generation( );
917   // Get the makeup of the population.
919   learner.genetic_algorithm.compute_population_makeup( );
921   // Load in the first genome's genes.
923   learner.current_genome_being_evaluated = 0;
925   learner.neural_net.put_weights( learner.genetic_algorithm.get_genome_genes( learner.
926     current_genome_being_evaluated ) );
927 }
928 else
929 {
930   // Test the next genome in the population.
931   learner.current_genome_being_evaluated += 1;
932   learner.neural_net.put_weights( learner.genetic_algorithm.get_genome_genes( learner.
933     current_genome_being_evaluated ) );
934 }
935 learner.evaluate_current_genome = false;
936 reset();
937 }
938
939 var ball_magnitude_reduction_threshold = 100;
940 var ball_magnitude_reduce_by_percentage = .5;
941
942 function collision_handler( colliding_objects )
943 {
944
945   learner.evaluate_current_genome = false;
946
947   for ( var i = 0; i < colliding_objects.length; ++i )
948   {
949
950     // Ball collisions
951
952     if ( colliding_objects[ i ][ 0 ].id == "ball" && colliding_objects[ i ][ 1 ].id == "paddle" || (
953       colliding_objects[ i ][ 0 ].id == "paddle" && colliding_objects[ i ][ 1 ].id == "ball" ) )
954     {
955
956       ball.set_magnitude( ( ball.get_magnitude( ) - ( ball.get_magnitude( ) *
957         ball_magnitude_reduce_by_percentage ) ) > ball_magnitude_reduction_threshold ? ( ball.get_magnitude(
958           ) - ( ball.get_magnitude( ) * .1 ) ) : 0 );
959
960       learner.current_genome_paddle_hits = learner.current_genome_paddle_hits + 1;
961
962       if ( ball.get_magnitude( ) == 0 )
963       {
964
965         learner.evaluate_current_genome = true;
966
967       }
968
969       // Wind the ball's position back, going along the direction opposite of the angle it was traveling
970       // along when it collided with
971       // the paddle until it no longer collides with the paddle.
972       // This is easier to calculate than finding the intersection of a line along the direction of the
973       // ball's vector with that of
974       // the paddle's top, right, bottom, and left boundary lines.
975
976       var reverse_angle = ball.mod_degrees( ( ball.get_angle( ) + 180 ) );
977       var dx = 5 * Math.cos( ball.degrees_to_radians( reverse_angle ) );
978       var dy = -5 * Math.sin( ball.degrees_to_radians( reverse_angle ) ); // Negative one because of the
979       // mirrored coordinate system.
980
981     /*
982       while ( physics_engine.rectangle_intersection( paddle.dynamic_object, ball.dynamic_object ) )
983     {
984
985       console.log( "x" );
986
987       ball.dynamic_object.move_left_top( dx, dy );
988
989     }
990
991   */
992
993   // There are two cases:
994   // 1. The ball hits the paddle from the top or bottom.
995   // 2. The ball hits the paddle from either the left or right.
996
997   if ( ( ball.dynamic_object.get_top( ) > paddle.dynamic_object.get_bottom( ) ) || ( ball.
998     dynamic_object.get_bottom( ) < paddle.dynamic_object.get_top( ) ) )
999   {
1000
1001   // Top or bottom side case.

```



```

1093     ball.set_magnitude( ( ball.get_magnitude() - ( ball.get_magnitude() *
1094         ball.magnitude_reduce_by_percentage ) ) > ball.magnitude_reduction_threshold ? ( ball.get_magnitude(
1095             ) - ( ball.get_magnitude() * .1 ) ) : 0 );
1096
1097     if ( ball.get_magnitude( ) == 0 )
1098     {
1099         learner.evaluate_current_genome = true;
1100     }
1101
1102     ball.dynamic_object.move_left_top( 0, ( -1 * ( ball.dynamic_object.get_bottom( ) - window.
1103         innerHeight ) ) -1 );
1104
1105     var reflection_angle = ball.mod_degrees( -1 * ball.get_angle( ) );
1106     ball.set_angle( reflection_angle );
1107
1108 } else if ( colliding_objects[ i ][ 0 ].id == "ball" && colliding_objects[ i ][ 1 ].id == "left_wall" ||
1109 ( colliding_objects[ i ][ 0 ].id == "left_wall" && colliding_objects[ i ][ 1 ].id == "ball" ) )
1110 {
1111     ball.set_magnitude( ( ball.get_magnitude() - ( ball.get_magnitude() *
1112         ball.magnitude_reduce_by_percentage ) ) > ball.magnitude_reduction_threshold ? ( ball.get_magnitude(
1113             ) - ( ball.get_magnitude() * .1 ) ) : 0 );
1114
1115     learner.evaluate_current_genome = true;
1116
1117     ball.dynamic_object.move_left_top( ( -1 * ( ball.dynamic_object.get_left( ) ) ) + 1, 0 );
1118
1119     var reflection_angle = -1 * ( ball.get_angle( ) - 180.0 );
1120     ball.set_angle( reflection_angle );
1121
1122 // Paddle collisions.
1123
1124 if ( colliding_objects[ i ][ 0 ].id == "paddle" && colliding_objects[ i ][ 1 ].id == "top_wall" || (
1125     colliding_objects[ i ][ 0 ].id == "top_wall" && colliding_objects[ i ][ 1 ].id == "paddle" )
1126 {
1127     paddle.dynamic_object.move_left_top( 0, ( -1 * paddle.dynamic_object.get_top( ) ) + 1 );
1128
1129 } else if ( colliding_objects[ i ][ 0 ].id == "paddle" && colliding_objects[ i ][ 1 ].id == "bottom_wall"
1130 "|| ( colliding_objects[ i ][ 0 ].id == "bottom_wall" && colliding_objects[ i ][ 1 ].id == "paddle"
1131 ) )
1132 {
1133     paddle.dynamic_object.move_left_top( 0, ( -1 * ( paddle.dynamic_object.get_bottom( ) - window.
1134         innerHeight ) ) -1 );
1135 }
1136
1137 }
1138
1139 }

```

LISTING A.2: Genetic_Algorithm.js

```

/*
2  *
3  * David Lettier (C) 2013.
4  *
5  * http://www.lettier.com/
6  *
7  * Code ported to JS and HEAVILY modified from original C++ source
8  * found at http://www.ai-junkie.com/ann/evolved/nn1.html and
9  * written by Mat Buckland.
10 *
11 * Implements a genetic algorithm for NN weight tuning.
12 *
13 */
14
15 function Genome( genes, fitness )
16 {
17
18     this.genes = null;
19     this.fitness = null;
20
21     if ( genes == undefined ) this.genes = new Array( );
22     else this.genes = genes;
23
24     if ( fitness == undefined ) this.fitness = 0.0;
25     else this.fitness = fitness;
26
27     // Used to calculate either the crossover progress or mutation progress.
28     // If this genome is created via crossover, use the weighted average
29     // based on the cross over point.
30     // So if the crossover point is say 9 and the genome length is 10,
31     // then the weighted average pf = (p1.f*.9) + (p2.f*.1).
32     // In other words the offspring received 90% of its genes from parent one
33     // and it received 10% of its genes from parent two so its parent fitness is
34     // 90% of parent one's fitness and 10% of parent two's fitness.
35
36     this.parent_fitness = 0.0;
37
38     // Created by means if this genome was generated either by randomness, crossover,
39     // mutation, both crossover and mutation, or elitism.
40     // Initially it is created from nothing so set it to -1.

```

```

42 // 0 = randomness, 1 = crossover, 2 = mutation, 3 = crossover & mutation, 4 = elitism
43 // This encoding is to facilitate crossover's and mutation's progress at producing fitter offspring than
44 // the offspring's parents.
45 this.created_by = 0;
46 }
47
48 function Genetic_Algorithm( params )
49 {
50
51 // Size of population.
52 this.population_size = params.popSize;
53
54 // Amount of genes per genome.
55 this.number_of_genes_per_genome = params.nGenesPerGenome;
56
57 // Use rank in selection?
58 this.use_rank_fitness = params.useRankFitness;
59
60 // Perform crossover and mutation sequentially or separately?
61 this.perform_crossover_and_mutation_sequentially = params.pOrMuSeq;
62
63 // Probability of genome's crossing over bits.
64 // 0.7 is pretty good.
65
66 this.crossover_probability = params.iCProb;
67 this.crossover_probability_minimum = 0.001;
68 this.crossover_probability_adjustment = 0.01;
69 this.crossover_operator_progress_average = 0.0;
70 this.observed_crossover_rate = 0.0;
71 this.total_number_of_crossovers = 0;
72 this.total_number_of_crossover_attempts = 0;
73
74 // Probability that a genomes bits will mutate.
75 // Try figures around 0.05 to 0.3-ish.
76
77 this.mutation_probability = params.iMProb;
78 this.mutation_probability_minimum = 0.001;
79 this.mutation_probability_adjustment = 0.01;
80 this.mutation_operator_progress_average = 0.0;
81 this.observed_mutation_rate = 0.0;
82 this.total_number_of_mutations = 0;
83 this.total_number_of_mutation_attempts = 0;
84
85 // Set the number of elite that go on to the next generation.
86 this.number_of_elite = params.nElite;
87
88 // This holds the entire population of genomes.
89 this.population = new Array();
90
91 // Total fitness of population.
92 this.total_fitness = 0;
93
94 // Average fitness.
95 this.average_fitness = 0;
96
97 // Best fitness this population.
98 this.best_fitness = 0;
99
100 // Worst fitness.
101 this.worst_fitness = 0;
102
103 // Keeps track of the best genome.
104 this.fittest_genome_index = -1;
105
106 // Keep track of the worst genome.
107 this.weakest_genome_index = -1;
108
109 // Generation number.
110 this.generation_number = 0;
111
112 // Current population makeup of randoms, crossovers, mutants, crossover mutants, elites.
113 this.population_makeup = "";
114
115 // Initialize population with genomes consisting of random
116 // genes and all fitness's set to zero.
117
118 for ( var i = 0; i < this.population_size; ++i )
119 {
120
121     this.population.push( new Genome() );
122
123     for ( var j = 0; j < this.number_of_genes_per_genome; ++j )
124     {
125
126         this.population[ i ].genes.push( get_random_float( -1.0, 1.0 ) );
127
128     }
129
130 }
131
132
133
134
135
136
137
138
139
140

```

```

142     }
144   }
146
147   this.replace_population_genes = function ( replacement_population_genes )
148   {
149
150     if ( replacement_population_genes === undefined ||
151         replacement_population_genes.length === 0 ||
152         replacement_population_genes.length != ( this.population_size * this.number_of_genes_per_genome ) )
153     {
154       console.error( "[Genetic_Algorithm:replace_population_genes] Replacement population genes invalid." );
155       return null;
156     }
157
158   }
159
160   // Assumes replace_population_genes is one big array.
161   // Splices the big array based on the number of genes
162   // per genome.
163
164   // Big array: [ 1,1,1,1,1,1,1,1,1,1 ] >>
165   // Population: [ [ 1, 1 ] G0
166   //                [ 1, 1 ] G1
167   //                [ 1, 1 ] ...
168   //                [ 1, 1 ] ...
169   //                [ 1, 1 ] ...
170   //                [ 1, 1 ] ...
171   //                [ 1, 1 ] ...
172   //                ]
173
173   var k = 0;
174
175   for ( var i = 0; i < this.population_size; ++i )
176   {
177
178     this.population[ i ].genes = [ ];
179
180     for ( var j = 0; j < this.number_of_genes_per_genome; ++j )
181     {
182
183       this.population[ i ].genes.push( replacement_population_genes[ k ] );
184
185       k += 1;
186
187     }
188
189   }
190
191 }
192
193 this.selection_operator = function ( number_of_indexes )
194 {
195
196   // Assumes population has been evaluated.
197
198   // Assumes population is sorted in ascending order according to fitness.
199
200   // Roulette selection of n genome indexes in the population.
201
202   if ( !this.use_rank_fitness )
203   {
204
205     // Say we have a population of 4 with these fitness values:
206     // G-1: 1
207     // G-2: 2
208     // G-3: 3
209     // G-4: 4
210
211     // Total fitness: 10
212     // Probabilities:
213     // G-1: .1
214     // G-2: .2
215     // G-3: .3
216     // G-4: .4
217
218     // Now shift them over by the running sum.
219     // This give them a portion on the number line [0.0,1.0] proportional to their
220     // fitness.
221
222     // G-1: .1
223     // G-2: G-1 + .2 = .3
224     // G-3: G-2 + .3 = .6
225     // G-4: G-3 + .4 = 1.0
226
227     // 0.0-----10-----20-----30-----40-----50-----60-----70-----80-----90-----1.0
228     //          G-1           G-2           G-3           G-4
229
230     // Now selected a random float in [0.0,1.0]:
231     // RF: .51
232
233     // 0.0-----10-----20-----30-----40-----50-----60-----70-----80-----90-----1.0
234     //          G-1           G-2           RF   G-3           G-4
235
236     // G-3 gets selected for mating.
237
238   var probabilities = new Array( );
239
240   var genome_indexes_selected = new Array( );
241
242   if ( this.total_fitness == 0 )

```

```

244     {
245         // So that we don't divide by zero.
246         // This means genomes all have zero fitness
247         // so just select random genome indexes.
248
249         for ( var i = 0; i < number_of_indexes; ++i )
250     {
251
252         genome_indexes_selected.push( get_random_integer( 0, this.population_size - 1 ) );
253     }
254
255     return genome_indexes_selected;
256 }
257
258
259     probabilities.push( this.population[ 0 ].fitness / this.total_fitness );
260
261     for ( var i = 1; i < this.population_size; ++i )
262     {
263
264         probabilities.push( probabilities[ i - 1 ] + ( this.population[ i ].fitness / this.total_fitness ) );
265     }
266
267
268     while ( genome_indexes_selected.length < number_of_indexes )
269     {
270
271         var random_number = get_random_float( 0.0, 1.0 );
272
273         for ( var i = 0; i < this.population_size; ++i )
274     {
275
276             if ( random_number <= probabilities[ i ] )
277             {
278
279                 genome_indexes_selected.push( i );
280             }
281         }
282
283     }
284
285
286     return genome_indexes_selected;
287 }
288
289 else
290 {
291
292     // Give the worst genome a rank fitness of 1.
293     // Give the second worst genome a rank fitness of 2.
294     // ...
295     // Give the best genome a rank fitness of the population size.
296
297     // Now, based on rank fitness, do a roulette selection where the
298     // probabilities are based on the rank fitness.
299
300     var probabilities = new Array( );
301
302     var genome_indexes_selected = new Array( );
303
304     // Rank fitness of the first is 1.
305     // Probability is 1/(n(n+1)/2).
306     // Where n is population size.
307     // (n(n+1)/2) = the total rank fitness.
308     // Summing the numbers from 1 to population size.
309     // Say population size is 10.
310     // Rank fitness: G-1 = 1, G-2 = 2, ..., G-10 = 10.
311     // Total rank fitness is 1+2+3+...+10 = n(n+1)/2 = (10*11)/2 = 55
312     // Probabilities:
313     // G-1: 1/55
314     // G-2: G-1 + 2/55
315     // ...
316     // G-10: G-9 + 10/55
317
318     var total_rank_fitness = ( this.population_size * ( this.population_size + 1 ) ) / 2;
319
320     probabilities.push( 1 / total_rank_fitness ); // First rank fitness probability.
321
322     // Rest of the rank fitness probabilities.
323
324     for ( var i = 1; i < this.population_size; ++i )
325     {
326
327         probabilities.push( probabilities[ i - 1 ] + ( ( i + 1 ) / total_rank_fitness ) );
328     }
329
330
331     while ( genome_indexes_selected.length < number_of_indexes )
332     {
333
334         var random_number = get_random_float( 0.0, 1.0 );
335
336         for ( var i = 0; i < this.population_size; ++i )
337     {
338
339             if ( random_number <= probabilities[ i ] )
340             {
341
342

```

```

344         genome_indexes_selected.push( i );
346     }
348 }
350 }
352     return genome_indexes_selected;
354 }
356 }
358 this.elitism_operator = function ( new_population )
{
360     if ( this.number_of_elite > this.population_size ) this.number_of_elite = this.population_size;
362     // Assumes the population is sorted in ascending order of fitness.
364     // A = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
366     // |A| = 10
367     // i = 2 check.
368     // i = 1 decrement.
369     // A[ ( 10 - 1 = 9 ) - i ] = 8 ]
370     // i = 1 check.
371     // i = 0 decrement.
372     // A[ ( 10 - 1 = 9 ) - i ] = 9 ]
373     // i = 0 check.
374     // Stop.
375
376     var i = this.number_of_elite;
377
378     while ( i-- )
379     {
380         var genome_temp = deep_copy( this.population[ ( this.population_size - 1 ) - i ] );
382         genome_temp.fitness = 0;
383         genome_temp.parent_fitness = 0;
384         genome_temp.created_by = 4;
386
387         new_population.push( genome_temp );
388
389         if ( new_population.length == this.population_size ) return;
390     }
392 }
394 this.crossover_operator = function ( parent_one_index, parent_two_index )
395 {
396     // One point crossover operator.
397
398     // Do we crossover?
399
400     if ( get_random_float( 0.0, 1.0 ) <= this.crossover_probability )
401     {
402         // If the parents are the same genome then this is not a true crossover.
403
404         if ( parent_one_index === parent_two_index ) return 0;
405
406         // Only returns one crossed offspring.
407
408         var offspring = new Genome();
409
410         // Determine a crossover point.
411
412         // Let the uniform sample be in the range of [1,n-1].
413         // If the crossover point was zero than no true crossover takes place
414         // as all of one parent's genes get copied into the offspring.
415         // If the cp = n-1 then at least you get n-1 from one parent and 1
416         // from another parent.
417
418         var crossover_point = get_random_integer( 1, ( this.number_of_genes_per_genome - 1 ) );
419
420         // Cross the parent's genes in the offspring.
421
422         offspring.genes = [];
423
424         offspring.fitness = 0;
425
426         offspring.parent_fitness = 0;
427
428         for ( var i = 0; i < crossover_point; ++i )
429         {
430
431             offspring.genes.push( deep_copy( this.population[ parent_one_index ].genes[ i ] ) );
432
433         }
434
435         for ( var i = crossover_point; i < this.number_of_genes_per_genome; ++i )
436         {
437
438             offspring.genes.push( deep_copy( this.population[ parent_two_index ].genes[ i ] ) );
439
440         }
441
442         // Determine if a crossover actually took place.
443
444     }

```

```

446 // The offspring should not match the parent one's genes and
448 // it should not match parent two's genes as the offspring
449 // should be a combination of the two.
450 if ( ( offspring.genes.toString( ) != this.population[ parent_one_index ].genes.toString( ) ) &&
451     ( offspring.genes.toString( ) != this.population[ parent_two_index ].genes.toString( ) ) )
452 {
453
454     // Weighted average fitness of the parents based on crossover point
455     // determining percentage of genes received from parent one and parent two.
456     // Let the number of genes per genome be 41 and let the crossover point be 1.
457     // Offspring gets [0,1) = 1 gene from parent one and [1,41) = 40 genes from parent two.
458     // PF = PF1 * (1/41) + PF2 * ((41-1)/41).
459
460     var parent_one_contribution = ( this.population[ parent_one_index ].fitness * ( (
461         crossover_point ) / ( this.number_of_genes_per_genome ) ) );
462     var parent_two_contribution = ( this.population[ parent_two_index ].fitness * ( ( this.
463         number_of_genes_per_genome - crossover_point ) / ( this.number_of_genes_per_genome ) ) );
464
465     offspring.parent_fitness = parent_one_contribution + parent_two_contribution;
466
467     offspring.created_by = 1;
468
469     return offspring;
470 }
471 else
472 {
473
474     return 0;
475 }
476 }
477 else
478 {
479
480     return 0;
481 }
482 }
483 }
484 }
485 }
486 }

this.mutation_operator = function ( parent_index )
{
490
491     // Mutates parent genome's genes on a whole genome basis based on the mutation probability.
492
493     // Gaussian distribution mutation.
494
495     // Do we mutate?
496
497     if ( get_random_float( 0.0, 1.0 ) <= this.mutation_probability )
498     {
499
500         // Reference: http://www.nashcoding.com/2010/07/07/evolutionary-algorithms-the-little-things-you-d-never-guess-part-1/
501
502         function gaussian_distribution( mean, standard_deviation )
503         {
504
505             // Two uniformly distributed random variable samples.
506
507             var x1 = Math.random();
508             var x2 = Math.random();
509
510             // The method requires sampling from a uniform random of (0,1]
511             // but Math.random() returns a sample of [0,1].
512
513             if ( x1 == 0.0 ) x1 = 1.0;
514             if ( x2 == 0.0 ) x2 = 1.0;
515
516             // Box-Muller transformation for Z_0.
517
518             var y1 = Math.sqrt( -2.0 * Math.log( x1 ) ) * Math.cos( 2.0 * Math.PI * x2 );
519
520             return ( y1 * standard_deviation ) + mean;
521
522         }
523
524         // Create an offspring blank.
525
526         var offspring = new Genome();
527
528         offspring.genes = [ ];
529
530         offspring.genes = deep_copy( this.population[ parent_index ].genes );
531
532         offspring.fitness = 0;
533
534         offspring.parent_fitness = 0;
535
536         // Begin to mutate.
537
538         var mutated = false;
539
540         for ( var i = 0; i < this.number_of_genes_per_genome; ++i )
541         {
542
543             // Mutate this gene by sampling a value from a normal distribution
544             // where the mean is the current gene value and the standard deviation

```

```

546     // the mutation step = mutation probability in the range [0,1].
547     // A low mutation probability will give a mutated gene value close to the original gene
548     // value (the mean) (most of the time) as the standard deviation is small and therefore the
549     // mutation step is small.
550     // A high mutation probability will give (or it can easily) a mutated gene value farther from the
551     // original gene
552     // value (the mean) as the standard deviation is large and therefore the mutation step is large.
553
554     // Note that gv = gv + *N(0,1) is the same as gv = N(gv, 1).
555
556     // Clamp the gene to range [-1,1].
557
558     var temp_gene_value = deep_copy( offspring.genes[ i ] );
559
560     offspring.genes[ i ] = gaussian_distribution( offspring.genes[ i ], this.mutation_probability );
561     offspring.genes[ i ] = get_clamped_value( offspring.genes[ i ], -1.0, 1.0 );
562
563     // Test if it was truly mutated.
564
565     if ( temp_gene_value != offspring.genes[ i ] )
566     {
567         mutated = true;
568     }
569
570     if ( mutated ) // If truly mutated.
571     {
572
573         offspring.parent_fitness = deep_copy( this.population[ parent_index ].fitness );
574
575         offspring.created_by = 2;
576
577         return offspring;
578     }
579     else
580     {
581
582         return 0;
583     }
584
585     else
586     {
587
588         return 0;
589     }
590
591     else
592     {
593
594         return 0;
595     }
596 }
597
598 this.crossover_then_mutate_operator = function ( parent_one_index, parent_two_index )
{
599
600     // Crossover and mutation done sequentially as in more traditional genetic algorithms.
601
602     // First attempts crossover and then attempts mutation.
603
604     var offspring_one = deep_copy( this.population[ parent_one_index ] );
605     var offspring_two = deep_copy( this.population[ parent_one_index ] );
606
607     offspring_one.fitness = 0.0;
608     offspring_two.fitness = 0.0;
609
610     offspring_one.parent_fitness = null;
611     offspring_two.parent_fitness = null;
612
613     offspring_one.created_by = 0;
614     offspring_two.created_by = 0;
615
616     // Attempt crossover.
617
618     var crossover_point = get_random_integer( 0, ( this.number_of_genes_per_genome - 1 ) );
619
620     if ( ( get_random_float( 0.0, 1.0 ) <= this.crossover_probability ) && ( parent_one_index !=
621         parent_two_index ) && ( crossover_point != 0 ) )
622     {
623
624         // Cross the parent's genes in the offspring.
625
626         offspring_one.genes = [ ];
627         offspring_two.genes = [ ];
628
629         for ( var i = 0; i < crossover_point; ++i )
630         {
631
632             offspring_one.genes.push( deep_copy( this.population[ parent_one_index ].genes[ i ] ) );
633             offspring_two.genes.push( deep_copy( this.population[ parent_two_index ].genes[ i ] ) );
634
635         }
636
637         for ( var i = crossover_point; i < this.number_of_genes_per_genome; ++i )
638         {
639
640             offspring_one.genes.push( deep_copy( this.population[ parent_two_index ].genes[ i ] ) );
641             offspring_two.genes.push( deep_copy( this.population[ parent_one_index ].genes[ i ] ) );
642
643         }
644
645     }
646
647 }

```

```

644     var parent_one_contribution = ( this.population[ parent_one_index ].fitness * ( (
645         crossover_point ) / ( this.number_of_genes_per_genome ) ) );
646     var parent_two_contribution = ( this.population[ parent_two_index ].fitness * ( ( this.
647         number_of_genes_per_genome - crossover_point ) / ( this.number_of_genes_per_genome ) ) );
648
648     offspring_one.parent_fitness = parent_one_contribution + parent_two_contribution;
649     offspring_one.created_by = offspring_one.created_by + 1;
650
650     parent_two_contribution = ( this.population[ parent_two_index ].fitness * ( (
651         crossover_point ) / ( this.number_of_genes_per_genome ) ) );
652     parent_one_contribution = ( this.population[ parent_one_index ].fitness * ( ( this.
653         number_of_genes_per_genome - crossover_point ) / ( this.number_of_genes_per_genome ) ) );
654
654     offspring_two.parent_fitness = parent_two_contribution + parent_one_contribution;
655     offspring_two.created_by = offspring_two.created_by + 1;
656   }
657
658   // Crossover may or may not have happened but now try to mutation.
659
660   // Normal distribution sample function.
661
662   function gaussian_distribution( mean, standard_deviation )
663   {
664
665     // Two uniformly distributed random variable samples.
666
667     var x1 = Math.random();
668     var x2 = Math.random();
669
670     // The method requires sampling from a uniform random of (0,1]
671     // but Math.random() returns a sample of [0,1].
672
673     if ( x1 == 0.0 ) x1 = 1.0;
674     if ( x2 == 0.0 ) x2 = 1.0;
675
676     // Box-Muller transformation for Z_0.
677
678     var y1 = Math.sqrt( -2.0 * Math.log( x1 ) ) * Math.cos( 2.0 * Math.PI * x2 );
679
680     return ( y1 * standard_deviation ) + mean;
681   }
682
683
684   // Attempt to mutate offspring one.
685
686   var mutated_one = false;
687
688   for ( var i = 0; i < this.number_of_genes_per_genome; ++i )
689   {
690
691     // Mutate this gene by sampling a value from a normal distribution
692     // where the mean is the current gene value and the standard deviation
693     // is the mutation step = mutation probability in the range [0,1].
694     // A low mutation probability will give a mutated gene value close to the original gene
695     // value (the mean) (most of the time) as the standard deviation is small and therefore the mutation
696     // step is small.
697     // A high mutation probability will give (or it can easily) a mutated gene value farther from the
698     // original gene
699     // value (the mean) as the standard deviation is large and therefore the mutation step is large.
700
701     // Note that gv = gv +   * N( 0, 1 ) is the same as gv = N( gv,   ).
702
703     // Clamp the gene to range [-1,1].
704
705     if ( get_random_float( 0.0, 1.0 ) <= this.mutation_probability ) // Mutate this gene?
706     {
707
708       var temp_gene_value_one = deep_copy( offspring_one.genes[ i ] );
709
710       offspring_one.genes[ i ] = gaussian_distribution( offspring_one.genes[ i ], this.
711         mutation_probability );
712
713       // offspring_one.genes[ i ] = gaussian_distribution( offspring_one.genes[ i ], 0.5 );
714       // offspring_one.genes[ i ] = offspring_one.genes[ i ] + ( get_random_float( -1.0, 1.0 ) * .3 );
715       offspring_one.genes[ i ] = get_clamped_value( offspring_one.genes[ i ], -1.0, 1.0 );
716
717       // Test if it was truly mutated.
718
719       if ( temp_gene_value_one != offspring_one.genes[ i ] )
720     {
721
722         mutated_one = true;
723
724     }
725
726   }
727
728   // Attempt to mutate offspring two.
729
730   var mutated_two = false;
731
732   for ( var i = 0; i < this.number_of_genes_per_genome; ++i )
733   {
734
735     // Mutate this gene by sampling a value from a normal distribution
736     // where the mean is the current gene value and the standard deviation
737     // is the mutation step = mutation probability in the range [0,1].
738     // A low mutation probability will give a mutated gene value close to the original gene

```

```

738 // value (the mean) (most of the time) as the standard deviation is small and therefore the mutation
739 // step is small.
740 // A high mutation probability will give (or it can easily) a mutated gene value farther from the
741 // original gene
742 // value (the mean) as the standard deviation is large and therefore the mutation step is large.
743
744 // Note that gv = gv + * N( 0, 1 ) is the same as gv = N( gv, 1 ).  

745
746 // Clamp the gene to range [-1,1].
747
748 if ( get_random_float( 0.0, 1.0 ) <= this.mutation_probability ) // Mutate this gene?
749 {
750
751     var temp_gene_value_two = deep_copy( offspring_two.genes[ i ] );
752
753     offspring_two.genes[ i ] = gaussian_distribution( offspring_two.genes[ i ], this.
754     mutation_probability );
755     // offspring_two.genes[ i ] = gaussian_distribution( offspring_two.genes[ i ], 0.5 );
756     // offspring_two.genes[ i ] = offspring_two.genes[ i ] + ( get_random_float( -1.0, 1.0 ) * .3 );
757     offspring_two.genes[ i ] = get_clamped_value( offspring_two.genes[ i ], -1.0, 1.0 );
758
759     // Test if it was truly mutated.
760
761     if ( temp_gene_value_two != offspring_two.genes[ i ] )
762     {
763         mutated_two = true;
764     }
765 }
766
767 if ( mutated_one ) // If truly mutated.
768 {
769
770     // Mutation = 2, crossover = 1, crossover + mutation = 3.
771
772     offspring_one.created_by = offspring_one.created_by + 2;
773
774     // If this offspring was only mutated, that is, it was not crossed then get its parent fitness.
775     // If it was crossed before being mutated then offspring_one.created would equal 3.
776
777     if ( offspring_one.created_by === 2 )
778     {
779
780         offspring_one.parent_fitness = deep_copy( this.population[ parent_one_index ].fitness );
781
782     }
783
784 }
785
786 if ( mutated_two ) // If truly mutated.
787 {
788
789     offspring_two.created_by = offspring_two.created_by + 2;
790
791     // If this offspring was only mutated, that is, it was not crossed then get its parent fitness.
792     // If it was crossed before being mutated then offspring_two.created would equal 3.
793
794     if ( offspring_two.created_by === 2 )
795     {
796
797         offspring_two.parent_fitness = deep_copy( this.population[ parent_two_index ].fitness );
798
799     }
800
801 }
802
803 // No parents->offspring not crossed and/or not mutated enter into the new population.
804 // Each offspring going into the new population must either crossed, mutated, or both.
805
806 if ( ( offspring_one.created_by === 0 ) || ( offspring_two.created_by === 0 ) )
807 {
808
809     return 0;
810
811 }
812
813 else if ( ( offspring_one.genes.toString() === this.population[ parent_one_index ].genes.toString()
814 ) || ( offspring_two.genes.toString() === this.population[ parent_two_index ].genes.toString() ) )
815 {
816
817     return 0;
818
819 }
820
821 else
822 {
823
824     return { one: offspring_one, two: offspring_two };
825
826 }
827
828 this.evaluate_population = function ( )
829 {
830
831     this.reset_population_evaluation();
832
833     var highest_so_far = this.population[ 0 ].fitness;
834     var lowest_so_far = this.population[ 0 ].fitness;

```

```

836     this.fittest_genome_index = 0;
838     this.weakest_genome_index = 0;
840     this.total_fitness = this.population[ 0 ].fitness;
841     this.best_fitness = this.population[ 0 ].fitness;
842     this.worst_fitness = this.population[ 0 ].fitness;
844     for ( var i = 1; i < this.population_size; ++i )
845     {
846         // Update fittest if necessary.
847         if ( highest_so_far < this.population[ i ].fitness )
848         {
849             highest_so_far = this.population[ i ].fitness;
850             this.fittest_genome_index = i;
851             this.best_fitness = highest_so_far;
852         }
853         // Update worst if necessary.
854         if ( lowest_so_far > this.population[ i ].fitness )
855         {
856             lowest_so_far = this.population[ i ].fitness;
857             this.weakest_genome_index = i;
858             this.worst_fitness = lowest_so_far;
859         }
860         this.total_fitness += this.population[ i ].fitness;
861     }
862     this.average_fitness = this.total_fitness / this.population_size;
863 }
864 this.reset_population_evaluation = function ( )
865 {
866     this.total_fitness = 0;
867     this.best_fitness = 0;
868     this.worst_fitness = 0;
869     this.average_fitness = 0;
870     this.fittest_genome_index = -1;
871     this.weakest_genome_index = -1;
872 }
873 this.compute_population_makeup = function ( )
874 {
875     var randoms = 0;
876     var crossovers = 0;
877     var mutants = 0;
878     var crossover_mutants = 0;
879     var elites = 0;
880     for ( var i = 0; i < this.population_size; ++i )
881     {
882         if ( this.population[ i ].created_by === 0 )
883         {
884             randoms = randoms + 1;
885         }
886         else if ( this.population[ i ].created_by === 1 )
887         {
888             crossovers = crossovers + 1;
889         }
890         else if ( this.population[ i ].created_by === 2 )
891         {
892             mutants = mutants + 1;
893         }
894         else if ( this.population[ i ].created_by === 3 )
895         {
896             crossover_mutants = crossover_mutants + 1;
897         }
898         else if ( this.population[ i ].created_by === 4 )
899         {
900             elites = elites + 1;
901         }
902     }
903 }

```

```

938     this.population_makeup = randoms + " " + crossovers + " " + mutants + " " + crossover_mutants + " " +
939     elites;
940 }
942 this.adjust_crossover_and_mutation_probabilities = function ( )
944 {
946     // Calculate the crossover and mutation operators' progress where
947     // their progress is based on how well they produced offspring that
948     // had a better fitness than their parent.
949
950     var crossover_operator_progress_sum = 0;
951     var number_of_crossovers = 0;
952
953     var mutation_operator_progress_sum = 0;
954     var number_of_mutations = 0;
955
956     // Sum all of the progresses.
957
958     for ( var i = 0; i < this.population_size; ++i )
959     {
960         if ( this.population[ i ].created_by === 1 ) // Created by crossover.
961         {
962             crossover_operator_progress_sum += ( this.population[ i ].fitness - this.population[ i ].parent_fitness );
963
964             number_of_crossovers += 1;
965
966         }
967         else if ( this.population[ i ].created_by === 2 ) // Created by mutation.
968         {
969             mutation_operator_progress_sum += ( this.population[ i ].fitness - this.population[ i ].parent_fitness );
970
971             number_of_mutations += 1;
972
973         }
974     }
975
976     // Now calculate the average crossover and mutation progress for the population.
977
978     this.crossover_operator_progress_average = 0.0;
979     this.mutation_operator_progress_average = 0.0;
980
981     if ( number_of_crossovers != 0 )
982     {
983
984         this.crossover_operator_progress_average = ( crossover_operator_progress_sum ) / (
985             number_of_crossovers );
986
987     }
988
989     if ( number_of_mutations != 0 )
990     {
991
992         this.mutation_operator_progress_average = ( mutation_operator_progress_sum ) / (
993             number_of_mutations );
994
995     }
996
997     // Adjust crossover and mutation rate adjustments.
998
999     if ( this.best_fitness > this.worst_fitness )
1000     {
1001
1002         this.crossover_probability_adjustment = 0.01 * ( ( this.best_fitness - this.average_fitness ) / (
1003             this.best_fitness - this.worst_fitness ) );
1004
1005         this.mutation_probability_adjustment = 0.01 * ( ( this.best_fitness - this.average_fitness ) / (
1006             this.best_fitness - this.worst_fitness ) );
1007
1008     }
1009     else if ( this.best_fitness == this.average_fitness )
1010     {
1011
1012         this.crossover_probability_adjustment = 0.01;
1013
1014         this.mutation_probability_adjustment = 0.01;
1015
1016     }
1017
1018     // Adjust crossover and mutation rates.
1019
1020     if ( this.crossover_operator_progress_average > this.mutation_operator_progress_average )
1021     {
1022
1023         this.crossover_probability = this.crossover_probability + this.crossover_probability_adjustment;
1024
1025         this.mutation_probability = this.mutation_probability - this.mutation_probability_adjustment;
1026
1027     }
1028     else if ( this.crossover_operator_progress_average < this.mutation_operator_progress_average )
1029     {
1030
1031         this.crossover_probability = this.crossover_probability - this.crossover_probability_adjustment;
1032

```

```

1034     this.mutation_probability = this.mutation_probability + this.mutation_probability_adjustment;
1036 }
1038 else if ( this.crossover_operator_progress_average == this.mutation_operator_progress_average )
{
1039     // Do not adjust.
1040 }
1042 this.crossover_probability = get_clamped_value( this.crossover_probability, this.
1043     crossover_probability_minimum, 1.0 );
1044 this.mutation_probability = get_clamped_value( this.mutation_probability, this.
1045     mutation_probability_minimum, 1.0 );
1046 }
1048 this.sort_population = function ( descending )
{
1049     if ( descending == undefined || descending == false )
{
1050         this.population.sort( function ( a, b ) { return a.fitness - b.fitness; } );
1051     }
1052     else
{
1053         this.population.sort( function ( a, b ) { return b.fitness - a.fitness; } );
1054     }
1055 }
1056 this.generate_new_generation = function ( )
{
1057     // Assumes population is sorted in ascending order.
1058     // Assumes population is completely evaluated.
1059     // Assumes crossover probability and mutation probability have been adjusted if using self-adaptation.
1060     // Create a temporary population to store newly created generation.
1061     var new_population = new Array( );
1062     // Allow the top N elite to pass into the next generation.
1063     this.elitism_operator( new_population );
1064     // Perform crossover and mutation separately?
1065     if ( !this.perform_crossover_and_mutation_sequentially )
{
1066         // Now we enter the GA loop.
1067         // Repeat until a new population is generated.
1068         while ( new_population.length < this.population_size )
{
1069             // Perform crossover and mutation separately.
1070             // Try to generate an offspring via crossover first.
1071             this.total_number_of_crossover_attempts += 1;
1072             // Select two genome indexes.
1073             var parents = this.selection_operator( 2 );
1074             var crossover_offspring = this.crossover_operator( parents[ 0 ], parents[ 1 ] );
1075             if ( crossover_offspring != 0 )
{
1076                 new_population.push( crossover_offspring );
1077                 this.total_number_of_crossovers += 1;
1078             }
1079             // There is the possibility of adding up to two
1080             // offspring per while loop.
1081             // Don't create more than the population size.
1082             if ( new_population.length === this.population_size ) break;
1083             // Try to generate an offspring via mutation second.
1084             this.total_number_of_mutation_attempts += 1;
1085             // Select one genome index.
1086             var parent = this.selection_operator( 1 );
1087             var mutation_offspring = this.mutation_operator( parent[ 0 ] );
1088         }
1089     }
1090 }
1091

```

```

1134     if ( mutation_offspring != 0 )
1135     {
1136         new_population.push( mutation_offspring );
1137         this.total_number_of_mutations += 1;
1138     }
1139 }
1140 }
1141 if ( new_population.length > this.population_size )
1142 {
1143     console.warn( "[Genetic_Algorithm:generate_new_generation] New population larger than population size." );
1144 }
1145 // Finished so assign new pop to the current population.
1146 this.population = [ ];
1147 this.population = deep_copy( new_population );
1148 new_population = [ ];
1149 }
1150 else // Perform crossover and mutation in sequence.
1151 {
1152     // Now we enter the GA loop.
1153     // Repeat until a new population is generated.
1154     while ( new_population.length < this.population_size )
1155     {
1156         // Attempt crossover and then mutation in sequence.
1157         var parents = this.selection_operator( 2 );
1158         var offspring = this.crossover_then_mutation_operator( parents[ 0 ], parents[ 1 ] );
1159         if ( offspring != 0 )
1160         {
1161             // First offspring.
1162             if ( offspring.one.created_by === 1 )
1163             {
1164                 this.total_number_of_crossovers += 1;
1165                 this.total_number_of_crossover_attempts += 1;
1166                 this.total_number_of_mutation_attempts += 1;
1167                 new_population.push( offspring.one );
1168             }
1169             else if ( offspring.one.created_by === 2 )
1170             {
1171                 this.total_number_of_mutations += 1;
1172                 this.total_number_of_crossover_attempts += 1;
1173                 this.total_number_of_mutation_attempts += 1;
1174                 new_population.push( offspring.one );
1175             }
1176             else if ( offspring.one.created_by === 3 )
1177             {
1178                 this.total_number_of_crossovers += 1;
1179                 this.total_number_of_mutations += 1;
1180                 this.total_number_of_crossover_attempts += 1;
1181                 this.total_number_of_mutation_attempts += 1;
1182                 new_population.push( offspring.one );
1183             }
1184         }
1185         // Old population size should match this new population size.
1186         if ( new_population.length === this.population_size ) break;
1187         // Second offspring.
1188         if ( offspring.two.created_by === 1 )
1189         {
1190             this.total_number_of_crossovers += 1;
1191             this.total_number_of_crossover_attempts += 1;
1192             this.total_number_of_mutation_attempts += 1;
1193             new_population.push( offspring.two );
1194         }
1195         else if ( offspring.two.created_by === 2 )
1196         {
1197             this.total_number_of_mutations += 1;
1198             this.total_number_of_crossover_attempts += 1;
1199             this.total_number_of_mutation_attempts += 1;
1200         }
1201     }
1202 }
1203 
```

```

1234         new_population.push( offspring.two );
1236     }
1238     else if ( offspring.two.created_by === 3 )
1239     {
1240         this.total_number_of_crossovers += 1;
1241         this.total_number_of_mutations += 1;
1242         this.total_number_of_crossover_attempts += 1;
1243         this.total_number_of_mutation_attempts += 1;
1244
1245         new_population.push( offspring.two );
1246     }
1247 }
1248 }
1249
1250 if ( new_population.length > this.population_size )
1251 {
1252     console.warn( "[Genetic_Algorithm:generate_new_generation] New population larger than population size." );
1253 }
1254
1255 // Finished so assign new pop to the current population.
1256
1257 this.population = [ ];
1258 this.population = deep_copy( new_population );
1259 new_population = [ ];
1260
1261 }
1262
1263 // Calculate the observed rates.
1264
1265 this.observed_crossover_rate = this.total_number_of_crossovers / this.
1266     total_number_of_crossover_attempts;
1267 this.observed_mutation_rate = this.total_number_of_mutations / this.
1268     total_number_of_mutation_attempts;
1269
1270 // Advance generation counter.
1271
1272 this.generation_number += 1;
1273
1274 }
1275
1276 // Getter methods.
1277
1278 this.get_population = function ( )
1279 {
1280     return deep_copy( this.population );
1281 }
1282
1283 this.get_population_size = function ( )
1284 {
1285     return deep_copy( this.population_size );
1286 }
1287
1288 this.get_number_of_genes_per_genome = function ( )
1289 {
1290     return deep_copy( this.number_of_genes_per_genome );
1291 }
1292
1293 this.get_genome_fitness = function ( index )
1294 {
1295     index = parseInt( index );
1296
1297     if ( ( index > ( this.population_size - 1 ) ) || ( index < 0 ) )
1298     {
1299         console.error( "[Genetic_Algorithm:get_genome_fitness] Index out of bounds of population size." );
1300
1301         return;
1302     }
1303
1304     return deep_copy( this.population[ index ].fitness );
1305 }
1306
1307 this.get_genome_genes = function ( index )
1308 {
1309     index = parseInt( index );
1310
1311     if ( ( index > ( this.population_size - 1 ) ) || ( index < 0 ) )
1312     {
1313         console.error( "[Genetic_Algorithm:get_genome_genes] Index out of bounds of population size." );
1314
1315         return;
1316     }

```

```

1334     return deep_copy( this.population[ index ].genes );
1336 }
1338 this.get_genome_genes_flattened = function ( index )
{
1340     index = parseInt( index );
1342     if ( ( index > ( this.population_size - 1 ) ) || ( index < 0 ) )
1344     {
1346         console.error( "[Genetic_Algorithm:get_genome_genes_flattened] Index out of bounds of population size." );
1348         return;
1350     }
1352     return deep_copy( this.population[ index ].genes.join( "," ) );
1354 }
1356 this.get_population_genes_flattened = function ( )
{
1358     var population_genes = "";
1360     for ( var i = 0; i < this.population_size - 1; ++i )
1362     {
1364         population_genes += deep_copy( this.population[ i ].genes.join( "," ) + "," );
1366     }
1368     population_genes += deep_copy( this.population[ this.population_size - 1 ].genes.join( "," ) );
1370     return population_genes;
1372 }
1374 this.get_best_fitness = function ( )
{
1376     this.evaluate_population( );
1378     return deep_copy( this.best_fitness );
1380 }
1382 this.get_average_fitness = function ( )
1384 {
1386     this.evaluate_population( );
1388     return deep_copy( this.average_fitness );
1390 }
1392 this.get_worst_fitness = function ( )
{
1394     this.evaluate_population( );
1396     return deep_copy( this.worst_fitness );
1398 }
1400 this.get_fittest_genome_index = function ( )
1402 {
1404     this.evaluate_population( );
1406     return deep_copy( this.fittest_genome_index );
1408 }
1410 this.get_weakest_genome_index = function ( )
{
1412     this.evaluate_population( );
1414     return deep_copy( this.weakest_genome_index );
1416 }
1418 this.get_crossover_probability = function ( )
{
1420     return deep_copy( this.crossover_probability );
1422 }
1424 this.get_mutation_probability = function ( )
{
1426     return deep_copy( this.mutation_probability );
1428 }
1430 this.get_generation_number = function ( )

```

```

1434  {
1436      return deep_copy( this.generation_number );
1438  }
1440 // Setter methods.
1442 this.set_crossover_probability = function ( rate )
{
1444     this.crossover_probability = parseFloat( rate );
1446 }
1448 this.set_mutation_probability = function ( rate )
{
1450     this.mutation_probability = parseFloat( rate );
1452 }
1454 this.set_generation_number = function ( number )
{
1456     number = parseInt( number );
1458     this.generation_number = number;
1460 }
1462 this.set_genome_fitness = function ( index, fitness )
{
1464     index = parseInt( index );
1466     fitness = parseFloat( fitness );
1468     if ( ( index > ( this.population_size - 1 ) ) || ( index < 0 ) )
1470     {
1472         console.error( "[Genetic_Algorithm:set_genome_fitness] Index out of bounds of population size." );
1474         return;
1476     }
1478     this.population[ index ].fitness = fitness;
1480 }
1482 }

```

LISTING A.3: Neural_Network.js

```

1 /*
2 *
3 * David Lettier (C) 2013.
4 *
5 * http://www.lettier.com/
6 *
7 * Code ported to JS and HEAVILY modified from original C++ source
8 * found at http://www.ai-junkie.com/ann/evolved/nnt1.html and
9 * written by Mat Buckland.
10 *
11 * Implements a neural network for learning.
12 *
13 */
14
15 function Neuron( nInputs )
16 {
17
18     // Plus one for the bias/threshold.
19     // x_1*w_1+x_2*w_2+...+x_n*w_n >= t
20     // x_1*w_1+x_2*w_2+...+x_n*w_n+(-1)*t >= 0
21     // Where t=w_(n+1).
22
23     this.number_of_inputs = nInputs + 1;
24
25     if ( this.number_of_inputs < 0 ) this.number_of_inputs = 1;
26
27     this.weights = new Array( );
28
29     for ( var i = 0; i < this.number_of_inputs; ++i )
30     {
31
32         this.weights.push( get_random_float( -1.0, 1.0 ) );
33
34     }
35
36 }
37
38 function Neuron_Layer( nNumberNeurons, nNumberInputsPerNeuron )
39 {
40
41     this.number_of_neurons = nNumberNeurons;
42
43     this.neurons = new Array( );
44
45     for ( var i = 0; i < nNumberNeurons; ++i )
46     {
47

```

```

49     this.neurons.push( new Neuron( nNumberInputsPerNeuron ) );
51   }
53 }
55 function Neural_Net( params )
{
57   this.number_of_inputs = params.nInputs;
59   this.number_of_outputs = params.nOutputs;
61   this.number_of_hidden_layers = params.nHiddenLayers;
63   this.neurons_per_hidden_layer = params.nNeuronsPerHiddenLayer;
65   this.bias = params.bias;
67   this.layers = new Array( );
69   this.create_network = function ( )
71 {
73     // Create the layers of the network.
74     if ( this.number_of_hidden_layers > 0 )
75     {
77       // Create first hidden layer.
78       this.layers.push( new Neuron_Layer( this.neurons_per_hidden_layer, this.number_of_inputs ) );
79       // Create the subsequent hidden layers.
80       for ( var i = 0; i < this.number_of_hidden_layers - 1; ++i )
81     {
83         this.layers.push( new Neuron_Layer( this.neurons_per_hidden_layer, this.neurons_per_hidden_layer ) );
84       };
85     }
87     // Create output layer.
88     this.layers.push( new Neuron_Layer( this.number_of_outputs, this.neurons_per_hidden_layer ) );
89   }
91   else
93   {
95     // Create output layer.
96     this.layers.push( new Neuron_Layer( this.number_of_outputs, this.number_of_inputs ) );
97   }
99 }
101 this.create_network( );
103 this.get_weights = function ( )
105 {
107   var weights = new Array( );
109   // For each layer.
111   for ( var i = 0; i < this.number_of_hidden_layers + 1; ++i ) // Plus one for the output layer.
113   {
115     // For each neuron.
116     for ( var j = 0; j < this.layers[ i ].number_of_neurons; ++j )
117     {
119       // For each weight.
120       for ( var k = 0; k < this.layers[ i ].neurons[ j ].number_of_inputs; ++k )
121     {
123       weights.push( this.layers[ i ].neurons[ j ].weights[ k ] );
125     };
127   }
129 }
131 return weights;
133 }
135 this.get_number_of_weights = function ( )
137 {
139   var weights = 0;
141   // For each layer.
143   for ( var i = 0; i < this.number_of_hidden_layers + 1; ++i ) // Plus one for the output layer.
145   {
147     // For each neuron.
148     for ( var j = 0; j < this.layers[ i ].number_of_neurons; ++j )

```

```

149      {
151          // For each weight.
153          for ( var k = 0; k < this.layers[ i ].neurons[ j ].number_of_inputs; ++k )
155          {
156              weights += 1;
157          }
159      }
161  }
163  return weights;
165 }
167 this.put_weights = function ( weights )
{
171     var weight = 0;
173     // For each layer.
174     for ( var i = 0; i < this.number_of_hidden_layers + 1; ++i ) // Plus one for the output layer.
175     {
177         // For each neuron.
178         for ( var j = 0; j < this.layers[ i ].number_of_neurons; ++j )
179         {
181             // For each weight.
182             for ( var k = 0; k < this.layers[ i ].neurons[ j ].number_of_inputs; ++k )
183             {
185                 this.layers[ i ].neurons[ j ].weights[ k ] = weights[ weight++ ];
187             }
189         }
191     }
193     return null;
195 }
197 this.update = function ( inputs )
{
199     // Stores the resultant outputs from each layer.
201     var outputs = new Array( );
203     var weight = 0;
205     // First check that we have the correct amount of inputs.
207     if ( inputs.length != this.number_of_inputs )
209     {
211         console.error( "[Neural_Net:update] Passed inputs length does not equal neural network number of inputs setting." );
213         // Just return an empty vector if incorrect.
214         return outputs;
215     }
217     // For each layer.
218     for ( var i = 0; i < this.number_of_hidden_layers + 1; ++i )
219     {
221         if ( i > 0 )
222         {
223             inputs = deep_copy( outputs ); // Deep copy outputs.
225         }
227         outputs.length = 0;
229         weight = 0;
231         // For each neuron sum the ( inputs * corresponding weights ).
232         // Throw the total at our sigmoid function to get the output.
233         for ( var j = 0; j < this.layers[ i ].number_of_neurons; ++j )
234         {
235             var net_input = 0.0;
236
237             var number_of_inputs = this.layers[ i ].neurons[ j ].number_of_inputs;
238
239             // For each weight.
240             for ( var k = 0; k < number_of_inputs - 1; ++k )
241             {
242                 // Sum the weights x inputs.
243
244                 net_input += this.layers[ i ].neurons[ j ].weights[ k ] * inputs[ weight++ ];
245             }
246         }
247     }

```

```

251     // Add in the bias/threshold.
252     //  $x_1*w_1+x_2*w_2+\dots+x_n*w_n \geq t$ 
253     //  $x_1*w_1+x_2*w_2+\dots+x_n*w_n+(-1)*t \geq 0$ 
254     // Where  $t=w_{n+1}$ .
255
256     net_input += this.bias * this.layers[ i ].neurons[ j ].weights[ number_of_inputs - 1 ];
257
258     // We can store the outputs from each layer as we generate them.
259     // The combined activation is first filtered through the Sigmoid function.
260
261     outputs.push( this.sigmoid( net_input ) );
262
263     weight = 0;
264 }
265
266 return outputs;
267
268 }
269
270 this.sigmoid = function ( input )
271 {
272
273     // Use the hyperbolic tangent function to get a range of outputs [-1,1].
274
275     var math_exp = Math.exp;
276
277     var numerator = math_exp( 2 * input ) - 1;
278     var denominator = math_exp( 2 * input ) + 1;
279
280     return numerator / denominator;
281
282     /*
283     Old way.
284
285     // Returns [ -1, 1 ].
286
287     var numerator = 1.0;
288     var denominator = 1.0 + Math.pow( Math.E, ( -input / response ) );
289
290     return ( ( numerator / denominator ) - 0.5 ) * 2.0;
291
292     */
293
294 }
295
296
297 }
298
299 }

```

A.2 BBAutoTune

LISTING A.4: GA.py

```

1  #! /usr/bin/env python
2
3  """
5  David Lettier (C) 2014.
7  http://www.lettier.com/
9  The main GA python file.
11 """
13 import sys;
14 import atexit;
15 import random;
16 import math;
17 import copy;
18 import signal;
19 import time;
20 import itertools;
21 import subprocess;
22 import mysql.connector;
23 import os;
24 import bpy;
25 import pickle;
26 import scipy;
27 import numpy;
28 import sklearn.covariance;
29 from bpy.props import *
31 """
33 Helper functions.
35 """
37 def get_clamped_value( value, minimum, maximum ):
38     return max( min( maximum, value ), minimum );
40 def get_scripts_location( ):
41     current_working_directory = os.getcwd();

```

```

45     current_working_directory = current_working_directory.rsplit( "/", 1 )
47     while ( current_working_directory[ 1 ] != "bbautotune" ):
49         current_working_directory = current_working_directory[ 0 ].rsplit( "/", 1 );
51     return current_working_directory[ 0 ] + "/bbautotune/source/scripts/";
53     """
55 Creates the Blender properties for the BBAutoTune UI panel.
57     """
59 def initialize_bbautotune_parameter_properties( ):
61     bpy.types.Scene.GA_POPULATION_SIZE = IntProperty(
63         name = "Population Size",
64         description = "Population size.",
65         min = 0
66     );
67     bpy.context.scene[ "GA_POPULATION_SIZE" ] = 10;
69     bpy.types.Scene.GA_MAX_GENERATIONS = IntProperty(
71         name = "Max Generations",
72         description = "Max generations.",
73         min = 0
74     );
75     bpy.context.scene[ "GA_MAX_GENERATIONS" ] = 100;
77     bpy.types.Scene.GA_NUMBER_OF_ELITE = IntProperty(
79         name = "Number of Elite",
80         description = "Number of elite.",
81         min = 0
82     );
83     bpy.context.scene[ "GA_NUMBER_OF_ELITE" ] = 2;
84     bpy.types.Scene.GA_CROSSOVER_PROBABILITY = FloatProperty(
86         name = "Crossover Probability",
87         description = "Crossover probability.",
88         default = 0.8,
89         min = 0.0,
90         max = 1.0
91     );
92     bpy.types.Scene.GA_MUTATION_PROBABILITY = FloatProperty(
94         name = "Mutation Probability",
95         description = "Mutation probability.",
96         default = 0.2,
97         min = 0.0,
98         max = 1.0
99     );
100    bpy.types.Scene.GA_MAX_TORQUE = FloatProperty(
101        name = "Max Torque",
102        description = "Maximum torque value possible during search.",
103        default = 50.0,
104        min = 0.0,
105        max = 340282346638528859811704183484516925440.0
106    );
107    bpy.types.Scene.GA_USE_RANK_FITNESS_SELECTION = BoolProperty(
108        name = "Use Rank Fitness Selection",
109        description = "Use rank fitness selection otherwise tournament selection will be used."
110    );
111    bpy.context.scene[ "GA_USE_RANK_FITNESS_SELECTION" ] = False;
112    bpy.types.Scene.GA_PERFORM_CROSSOVER_AND_MUTATION_SEQUENTIALLY = BoolProperty(
113        name = "Perform Crossover and Mutation Sequentially",
114        description = "Perform crossover and mutation sequentially."
115    );
116    bpy.context.scene[ "GA_PERFORM_CROSSOVER_AND_MUTATION_SEQUENTIALLY" ] = False;
117    bpy.types.Scene.GA_USE_SELF_ADAPTATION = BoolProperty(
118        name = "Use Self-adaptation",
119        description = "Adapt the crossover and mutation probabilities."
120    );
121    bpy.context.scene[ "GA_USE_SELF_ADAPTATION" ] = False;
122    bpy.types.Scene.BBAUTOTUNE_OPEN_GA_MONITOR_BROWSER_WINDOW = BoolProperty(
123        name = "Open GA Monitor Browser Window",
124        description = "Open a browser window to the GA monitor."
125    );

```

```

147 );
149 bpy.context.scene[ "BBAUTOTUNE_OPEN_GA_MONITOR_BROWSER_WINDOW" ] = False;
150 bpy.types.Scene.BBAUTOTUNE_DEBUG = BoolProperty(
151     name      = "Debug",
152     description = "Log debug information."
153 );
154 bpy.context.scene[ "BBAUTOTUNE_DEBUG" ] = False;
155 initialize_bbautotune_parameter_properties( );
156 /**
157 * The start button operator on the BBAutoTune UI panel.
158 * Starts BBAutoTune with the values from the UI panel properties.
159 */
160
161 /**
162 * The BBAutoTune UI panel and its layout in Blender.
163 */
164
165 /**
166 * Class: BBAUTOTUNE_UI_START_BUTTON_OPERATOR< bpy.types.Operator >
167 */
168 class BBAUTOTUNE_UI_START_BUTTON_OPERATOR( bpy.types.Operator ):
169     bl_idname = "bbautotune.start";
170     bl_label   = "Start"
171
172     def execute( self, context ):
173         bpy.bbautotune.start(
174             bpy.context.scene.GA_POPULATION_SIZE,
175             bpy.context.scene.GA_MAX_GENERATIONS,
176             bpy.context.scene.GA_NUMBER_OF_ELITE,
177             bpy.context.scene.GA_CROSSOVER_PROBABILITY,
178             bpy.context.scene.GA_MUTATION_PROBABILITY,
179             bpy.context.scene.GA_MAX_TORQUE,
180             bpy.context.scene.GA_USE_RANK_FITNESS_SELECTION,
181             bpy.context.scene.GA_PERFORM_CROSSOVER_AND_MUTATION_SEQUENTIALLY,
182             bpy.context.scene.GA_USE_SELF_ADAPTATION,
183             bpy.context.scene.BBAUTOTUNE_OPEN_GA_MONITOR_BROWSER_WINDOW,
184             bpy.context.scene.BBAUTOTUNE_DEBUG
185         );
186
187         # Clean up on premature exit.
188         atexit.register( bpy.bbautotune.stop );
189
190         return { 'FINISHED' };
191
192 /**
193 * Class: GA_UI_PANEL< bpy.types.Panel >
194 */
195 class GA_UI_PANEL( bpy.types.Panel ):
196     bl_label       = "BBAutoTune Parameters";
197     bl_space_type  = "PROPERTIES";
198     bl_region_type = "WINDOW";
199     bl_context     = "render";
200
201     def draw( self, context ):
202
203         self.layout.prop( context.scene, "GA_POPULATION_SIZE" );
204         self.layout.prop( context.scene, "GA_MAX_GENERATIONS" );
205         self.layout.prop( context.scene, "GA_NUMBER_OF_ELITE" );
206         self.layout.prop( context.scene, "GA_CROSSOVER_PROBABILITY" );
207         self.layout.prop( context.scene, "GA_MUTATION_PROBABILITY" );
208         self.layout.prop( context.scene, "GA_MAX_TORQUE" );
209         self.layout.prop( context.scene, "GA_USE_RANK_FITNESS_SELECTION" );
210         self.layout.prop( context.scene, "GA_PERFORM_CROSSOVER_AND_MUTATION_SEQUENTIALLY" );
211         self.layout.prop( context.scene, "GA_USE_SELF_ADAPTATION" );
212         self.layout.prop( context.scene, "BBAUTOTUNE_OPEN_GA_MONITOR_BROWSER_WINDOW" );
213         self.layout.prop( context.scene, "BBAUTOTUNE_DEBUG" );
214
215         self.layout.operator( "bbautotune.start" );
216
217 /**
218 * A single genome.
219 */
220
221 class Genome( ):
222
223     new_id = itertools.count().__next__;
224
225     def __init__( self, genes = None, fitness = None ):
226
227         self.id = Genome.new_id();
228
229         if ( not genes == None ):
230             self.genes = list( genes );
231
232         else:
233             self.genes = [ ];
234
235         self.fitness = fitness or 0.0;

```

```

249     # Used to calculate either the crossover progress or mutation progress.
250     # If this genome is created via crossover, use the weighted average
251     # based on the cross over point.
252     # So if the crossover point is say 9 and the genome length is 10,
253     # then the weighted average pf = (p1.f*.9) + (p2.f*.1).
254     # In other words the offspring received 90% of its genes from parent one
255     # and it received 10% of its genes from parent two so its parent fitness is
256     # 90% of parent one's fitness and 10% of parent two's fitness.
257
258     self.parent_fitness = 0.0;
259
260     # Created by means if this genome was generated either by randomness, crossover,
261     # mutation, both crossover and mutation, or elitism.
262     # Initially it is created from nothing so set it to -1.
263     # 0 = randomness, 1 = crossover, 2 = mutation, 3 = crossover & mutation, 4 = elitism
264
265     # This encoding is to facilitate crossover's and mutation's progress at producing fitter
266     # offspring than the offspring's parents.
267
268     self.created_by = 0;
269
270     def set_genes( self, genes = None ):
271
272         if ( not genes == None ):
273
273             self.genes = list( genes );
274
275         else:
276
276             self.genes = [ ];
277
278     def get_genes( self ):
279
280
281         return list( self.genes );
282
283     def get_genes_as_string( self ):
284
285         return ",".join( map( str, self.genes ) );
286
287     def set_fitness( self, fitness = None ):
288
289         self.fitness = fitness or 0.0;
290
291     def get_fitness( self ):
292
293
294         return self.fitness;
295
296     def set_parent_fitness( self, parent_fitness = None ):
297
298         self.parent_fitness = parent_fitness or 0.0;
299
300     def get_parent_fitness( self ):
301
302
303         return self.parent_fitness;
304
305     def set_created_by( self, created_by = None ):
306
307         self.created_by = created_by or 0;
308
309     def get_created_by( self ):
310
311
312         return self.created_by;
313
314     def __repr__( self ):
315
316         return repr( ( self.id, self.created_by, self.fitness, self.parent_fitness, self.genes ) );
317
318     """
319
320     The main genetic algorithm object.
321
322     """
323
324     class Genetic_Algorithm( ):
325
326         def __init__(self,
327
328             self,
329             population_size = None,
330             max_generations = None,
331             number_of_elite = None,
332             crossover_probability = None,
333             mutation_probability = None,
334             use_rank_fitness_selection = None,
335             perform_crossover_and_mutation_sequentially = None,
336             use_self_adaptation = None
337         ):
338
339
340         """
341
342         Begin parameters.
343
344         """
345
346         # Size of population.
347
348         self.population_size = population_size or 0;
349
350         # Number of generations to run until termination of the algorithm.

```

```

    self.max_generations = max_generations or 0;
351
# Set the number of elite that go on to the next generation.
353 self.number_of_elite = number_of_elite or 0;
355
# Probability of genome's crossing over bits.
357 # 0.7 is pretty good.
358
359 self.crossover_probability      = crossover_probability or 0.0;
360 self.crossover_probability_minimum = 0.001;
361 self.crossover_probability_adjustment = 0.01;
362 self.crossover_operator_progress_average = 0.0;
363 self.observed_crossover_rate     = 0.0;
364 self.total_number_of_crossovers = 0;
365 self.total_number_of_crossover_attempts = 0;
366
367 # Probability that a genomes bits will mutate.
368 # Try figures around 0.05 to 0.3-ish.
369
370 self.mutation_probability      = mutation_probability or 0.0;
371 self.mutation_probability_minimum = 0.001;
372 self.mutation_probability_adjustment = 0.01;
373 self.mutation_operator_progress_average = 0.0;
374 self.observed_mutation_rate     = 0.0;
375 self.total_number_of_mutations = 0;
376 self.total_number_of_mutation_attempts = 0;
377
378 # Use rank fitness selection?
379 self.use_rank_fitness_selection = use_rank_fitness_selection or False;
380
381 # Perform crossover and mutation sequentially or separately?
382 self.perform_crossover_and_mutation_sequentially = perform_crossover_and_mutation_sequentially or
383 False;
384
385 # Use self-adaptation?
386 self.use_self_adaptation = use_self_adaptation or False;
387
388 """
389 End parameters.
390 """
391
392 # Log file if debugging.
393 self.log_file_name = "";
394
395 # The amount of genes per genome.
396 self.number_of_genes_per_genome = 18;
397
398 # This holds the entire population of genomes.
399 self.population = [];
400
401 # Total fitness of population.
402 self.total_fitness = 0.0;
403
404 # Average fitness.
405 self.average_fitness = 0.0;
406
407 # Highest fitness.
408 self.highest_fitness = 0.0;
409
410 # Lowest fitness.
411 self.lowest_fitness = 0.0;
412
413 # Keeps track of the best genome.
414 self.fittest_genome_index = -1;
415
416 # Keep track of the worst genome.
417 self.weakest_genome_index = -1;
418
419 # Generation number.
420 self.generation_number = 0;
421
422 # Current population makeup of randoms, crossovers, mutants, crossover mutants, and elites.
423 self.population_makeup = "";
424
425 def set_population_size( self, size = None ):
426     self.population_size = size or 0;
427
428 def get_population_size( self ):
429     return self.population_size;
430
431 def set_max_generations( self, maximum = None ):
432     self.max_generations = maximum or 0;

```

```

451     def get_max_generations( self ):
453         return self.max_generations;
455     def set_crossover_probability( self, probability = None ):
457         self.crossover_probability = probability or 0;
459     def get_crossover_probability( self ):
461         return self.crossover_probability;
463     def set_mutation_probability( self, probability = None ):
465         self.mutation_probability = probability or 0;
467     def get_mutation_probability( self ):
469         return self.mutation_probability;
471     def set_number_of_genes_per_genome( self, number_of = None ):
473         self.number_of_genes_per_genome = number_of or 0;
475     def get_number_of_genes_per_genome( self ):
477         return self.number_of_genes_per_genome;
479     def set_use_rank_fitness_selection( self, boolean = None ):
481         self.use_rank_fitness_selection = boolean or 0;
483     def get_use_rank_fitness_selection( self ):
485         return self.use_rank_fitness_selection;
487     def set_perform_crossover_and_mutation_sequentially( self, boolean = None ):
489         self.perform_crossover_and_mutation_sequentially = boolean or 0;
491     def get_perform_crossover_and_mutation_sequentially( self ):
493         return self.perform_crossover_and_mutation_sequentially;
495     def set_use_self_adaptation( self, use_self_adaptation = None ):
497         self.use_self_adaptation = use_self_adaptation or False;
499     def get_use_self_adaptation( self ):
501         return self.use_self_adaptation;
503     def set_number_of_elite( self, number_of = None ):
505         self.number_of_elite = number_of or 0;
507         if ( self.number_of_elite > self.population_size ):
509             self.number_of_elite = self.population_size;
511     def get_number_of_elite( self ):
513         return self.number_of_elite;
515     def set_log_file_name( self, log_file_name ):
517         self.log_file_name = log_file_name;
519     def get_log_file_name( self ):
521         return self.log_file_name;
523     def log( self, log_string ):
525         if ( self.log_file_name != "" ):
527             log_file = open( self.log_file_name, "a+" );
529             log_file.write( log_string + "\n" );
531             log_file.close();
533     def set_total_fitness( self, total_fitness = None ):
535         self.total_fitness = total_fitness or 0;
537     def get_total_fitness( self ):
539         return self.total_fitness;
541     def set_average_fitness( self, average_fitness = None ):
543         self.average_fitness = average_fitness or 0;
545     def get_average_fitness( self ):
547         return self.average_fitness;
549     def set_highest_fitness( self, highest_fitness = None ):
551         self.highest_fitness = highest_fitness or 0;

```

```

553     def get_highest_fitness( self ):
555         return self.highest_fitness;
557     def set_lowest_fitness( self, lowest_fitness = None ):
559         self.lowest_fitness = lowest_fitness or 0;
561     def get_lowest_fitness( self ):
563         return self.lowest_fitness;
565     def set_fittest_genome_index( self, fittest_genome_index = None ):
567         self.fittest_genome_index = fittest_genome_index or -1;
569     def get_fittest_genome_index( self ):
571         return self.fittest_genome_index;
573     def set_weakest_genome_index( self, weakest_genome_index = None ):
575         self.weakest_genome_index = weakest_genome_index or -1;
577     def get_weakest_genome_index( self ):
579         return self.weakest_genome_index;
581     def set_generation_number( self, generation_number = None ):
583         self.generation_number = generation_number or 0;
585     def get_generation_number( self ):
587         return self.generation_number;
589     def get_population_makeup( self ):
591         self.compute_population_makeup();
593         return self.population_makeup;
595     def get_genome( self, index ):
597         assert index < self.population_size and index >= 0, "Genome index out of bounds.";
599         return copy.deepcopy( self.population[ index ] );
601     def get_genome_fitness( self, index ):
603         assert index < self.population_size and index >= 0, "Genome index out of bounds.";
605         return self.population[ index ].fitness;
607     def set_genome_fitness( self, index, fitness ):
609         assert index < self.population_size and index >= 0, "Genome index out of bounds.";
611         self.population[ index ].set_fitness( fitness );
613     def get_genome_genes_as_string( self, index ):
615         assert index < self.population_size and index >= 0, "Genome index out of bounds.";
617         return self.population[ index ].get_genes_as_string();
619     def get_population_genes_as_string( self ):
621         if ( self.population_size == 0 ):
623             return "";
625         else:
627             gene_string = self.population[ 0 ].get_genes_as_string();
629             for i in range( self.population_size ):
631                 gene_string += "," + self.population[ i ].get_genes_as_string();
633             return gene_string;
635     def create_randomized_population( self ):
637         self.log( "Generating random population." );
639         self.population = [ ];
641         # Initialize population with genomes consisting of random
643         # genes and all fitness's set to zero.
645         for i in range( self.population_size ):
647             self.population.append( Genome() );
649             self.log( "Genome: " + str( i ) );
651             for j in range( self.number_of_genes_per_genome ):
653                 random_gene = random.uniform( 0.0, 1.0 );

```

```

655         self.log( "Random gene: " + str( j ) );
657         self.log( str( random_gene ) );
659         self.population[ i ].genes.append( random_gene );
661         self.set_generation_number( 0 );
663         self.update_population_metrics();
665     def replace_population_genes( self, replacement_population_genes ):
667
668         # Assumes replacement_population_genes is one big array.
669         # Splices the big array based on the number of genes per genome.
670
671         # Big array: [ 1,2,3,4,5,6,7,8,9,10 ]
672         # Number of genes per genome: 2
673         # Population: [ [ 1, 2 ] G0
674         #                 [ 3, 4 ] G1
675         #                 [ 5, 6 ] ...
676         #                 [ 7, 8 ] ...
677         #                 [ 9, 10 ] GN-1
678         #             ]
679
680         assert len( replacement_population_genes ) != 0, "Replacement gene size is zero.";
681         assert len( replacement_population_genes ) == self.population_size * self.number_of_genes_per_genome,
682             "Too few or too many replacement genes."
683
684         k = 0;
685
686         for i in range( self.population_size ):
687
688             self.population[ i ].genes = [ ];
689
690             for j in range( self.number_of_genes_per_genome ):
691
692                 self.population[ i ].genes.append( replacement_population_genes[ k ] );
693
694                 k += 1;
695
696     def selection_operator( self, number_of_indexes ):
697
698         self.log( "Entering selection operator." );
699
700         self.log( str( number_of_indexes ) + " genomes requested." );
701
702         # Assumes the population has been evaluated.
703
704         # Assumes the population is sorted in descending order according to fitness.
705
706         if ( not self.use_rank_fitness_selection ):
707
708             tournament_size = number_of_indexes + 1;
709
710             assert tournament_size <= self.population_size and tournament_size >= 0, "Tournament size too large/small.";
711
712             genome_indexes_selected = [ ];
713
714             self.log( "Selecting random players." );
715
716             for i in range( number_of_indexes ):
717
718                 tournament_players = [ ];
719
720                 for j in range( tournament_size ):
721
722                     random_int = random.randint( 0, self.population_size - 1 );
723
724                     tournament_players.append( [ random_int, self.population[ random_int ].get_fitness() ] );
725
726                     self.log( "Random players selected." );
727
728                     self.log( str( tournament_players ) );
729
730                     # Remember, lower fitness values are a higher fitness.
731                     # Sorts list in ascending orderer.
732                     # tournament_players = [ [genome_index,genome_fitness], ... ]
733                     # [ 0 ][ 0 ] = get the index with the lowest fitness value (thus the highest fitness).
734
735                     player_ranking = sorted( tournament_players, key = lambda x: x[ 1 ] );
736
737                     self.log( "Player ranking." );
738
739                     self.log( str( player_ranking ) );
740
741                     winner = player_ranking[ 0 ][ 0 ];
742
743                     self.log( "Winner." );
744
745                     self.log( str( winner ) );
746
747                     genome_indexes_selected.append( winner );
748
749                     self.log( "Selected genomes with tournament selection." );
750
751                     self.log( str( genome_indexes_selected ) );
752
753                     return genome_indexes_selected;
754
755         else:

```

```

755     self.log( "Sorted?" );
757     self.log( str( self.population[ 0 ].get_fitness( ) ) + " " + str( self.population[ -1 ].get_fitness(
758         ) ) );
759     # Assume the genomes are sorted in non-increasing order of fitness.
761     # Give the worst genome a rank fitness of 1.
763     # Give the second worst genome a rank fitness of 2.
764     # ...
765     # Give the best genome a rank fitness of the population size.
766
767     genome_indexes_selected = [ ];
768
769     # Genomes:          1, 2, 3, 4
770     # Genomes fitnesses: 4, 2, 3, 1 (Lowest is highest.)
771     # Rank fitnesses:   1, 2, 3, 4 (Highest is highest.)
772     # Partial sums:    1, 3, 6, 10
773     # Random number U(0,10): 7
774     # Genome 4 is selected.
775     # Since all random numbers are uniform,
776     # Genome 1 has a probability of being selected: (1-0)*(1/10) = 10%
777     # Genome 2 has a probability of being selected: (3-1)*(1/10) = 20%
778     # Genome 3 has a probability of being selected: (6-3)*(1/10) = 30%
779     # Genome 4 has a probability of being selected: (10-6)*(1/10) = 40%
780     # Total of any being selected:                = 100%
781
782     #
783     # 0.10 -----
784     # | | | |
785     # | | | |
786     # 0---1---2---3---4---5---6---7---8---9---10
787     #      G1      G2      G3      G4
788
789     # Individual rank fitnesses.
790
791     rank_fitnesses = [ ];
792
793     for i in range( self.population_size ):
794         rank_fitnesses.append( i + 1 );
795
796     total_rank_fitness = sum( rank_fitnesses );
797
798     self.log( "Total rank fitness." );
799
800     self.log( str( total_rank_fitness ) );
801
802     # Partial sum. P[i] = sum( P[1:i] ) where i is in range [1,n].
803
804     partial_sums = list( itertools.accumulate( rank_fitnesses ) );
805
806     self.log( "Partial sums." );
807
808     self.log( str( partial_sums ) );
809
810     while ( len( genome_indexes_selected ) < number_of_indexes ):
811
812         random_number = random.randint( 0, total_rank_fitness );
813
814         self.log( "Random number: " + str( random_number ) );
815
816         for i in range( self.population_size ):
817
818             if ( partial_sums[ i ] >= random_number ):
819
820                 self.log( "Partial sum: " + str( partial_sums[ i ] ) );
821
822                 self.log( "Genome index selected: " + str( i ) );
823
824                 genome_indexes_selected.append( i );
825
826                 break;
827
828             if ( len( genome_indexes_selected ) == number_of_indexes ):
829
830                 break;
831
832         self.log( "Selected genomes with rank selection." );
833
834         self.log( str( genome_indexes_selected ) );
835
836         return genome_indexes_selected;
837
838     def elitism_operator( self, new_population ):
839
840         self.log( "Attempting to add elite genomes." );
841
842         if ( self.number_of_elite > self.population_size ):
843
844             self.number_of_elite = self.population_size;
845
846             # Assumes the population is sorted in ascending order of fitness.
847
848             # A = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
849             # |A| = 10
850             # i = 2 check.
851             # i = 1 decrement.
852             # A[ ( 10 - i = 9 ) - i ] = 8 ]
853             # i = 1 check.
854             # i = 0 decrement.
855

```

```

857     # A[ ( ( 10 - i = 9 ) - i ) = 9 ]
859     # i = 0 check.
860     # Stop.
861     i = self.number_of_elite;
862     while ( i ):
863         i -= 1;
864         genome_temp = copy.deepcopy( self.population[ ( self.population_size - 1 ) - i ] );
865         self.log( "Adding elite genome. Its fitness: " + str( genome_temp.get_fitness( ) ) );
866         genome_temp.fitness          = 0.0;
867         genome_temp.parent_fitness   = 0.0;
868         genome_temp.created_by      = 4;
869         new_population.append( genome_temp );
870         if ( len( new_population ) == self.population_size ):
871             break;
872     def crossover_operator( self, parent_one_index, parent_two_index ):
873         # One point crossover operator.
874         # Only returns one crossed offspring.
875         # Do we crossover?
876         self.log( "Attempting crossover." );
877         if ( random.uniform( 0.0, 1.0 ) <= self.crossover_probability ):
878             # If the parents are the same genome then this is not a true crossover.
879             if ( parent_one_index == parent_two_index ):
880                 self.log( "Parents are the same. Crossover failed." );
881                 return 0;
882             self.log( "Parent indexes." );
883             self.log( str( parent_one_index ) + " " + str( parent_two_index ) );
884             # Create a blank offspring.
885             offspring = Genome();
886             offspring.genes        = [ ];
887             offspring.fitness      = 0.0;
888             offspring.parent_fitness = 0.0;
889             offspring.created_by   = 0;
890             # Determine a crossover point.
891             # Let the uniform sample be in the range of [1,n-1].
892             # If the crossover point was zero than no true crossover takes place
893             # as all of one parent's genes get copied into the offspring.
894             # If the cp = n-1 then at least you get n-1 from one parent and 1
895             # from another parent.
896             crossover_point = random.randint( 1, ( self.number_of_genes_per_genome - 1 ) );
897             self.log( "Crossover point." );
898             self.log( str( crossover_point ) );
899             # Cross the parent's genes into the offspring's genes.
900             for i in range( crossover_point ):
901                 offspring.genes.append( copy.deepcopy( self.population[ parent_one_index ].genes[ i ] ) );
902             for i in range( crossover_point, self.number_of_genes_per_genome ):
903                 offspring.genes.append( copy.deepcopy( self.population[ parent_two_index ].genes[ i ] ) );
904             # Determine if a crossover actually took place.
905             # The offspring should not match the parent one's genes and
906             # it should not match parent two's genes as the offspring
907             # should be a combination of the two.
908             if ( ( offspring.genes != self.population[ parent_one_index ].genes ) and
909                  ( offspring.genes != self.population[ parent_two_index ].genes ) ):
910                 # Weighted average fitness of the parents based on crossover point
911                 # determining percentage of genes received from parent one and parent two.
912                 # Let the number of genes per genome be 41 and let the crossover point be 1.
913                 # Offspring gets [0,1] = 1 gene from parent one and [1,41) = 40 genes from parent two.
914                 # PF = PF1 * (1/41) + PF2 * ((41-1)/41).
915                 parent_one_contribution = ( self.population[ parent_one_index ].fitness * ( (
916                     crossover_point ) / ( self.number_of_genes_per_genome ) ) );
917                 parent_two_contribution = ( self.population[ parent_two_index ].fitness * ( ( self.
918                     number_of_genes_per_genome - crossover_point ) / ( self.number_of_genes_per_genome ) ) );
919                 offspring.parent_fitness = parent_one_contribution + parent_two_contribution;
920

```

```

957         self.log( "Parent fitnesses." );
959         self.log( str( self.population[ parent_one_index ].fitness ) );
961         self.log( str( self.population[ parent_two_index ].fitness ) );
963         self.log( "Parent fitness contributions." );
965         self.log( str( parent_one_contribution ) );
967         self.log( str( parent_two_contribution ) );
968         offspring.created_by = 1;
969
970         self.log( "Returning crossed offspring." );
971
972         return offspring;
973
974     else:
975
976         self.log( "Did not actually perform crossover. Offspring genes match parents." );
977
978         return 0;
979
980     else:
981
982         self.log( "Greater than crossover probability." );
983
984         return 0;
985
986     def mutation_operator( self, parent_index ):
987
988         # Mutates parent genome's genes on a whole genome basis based on the mutation probability.
989
990         # Gaussian distribution mutation.
991
992         # Do we mutate?
993
994         self.log( "Attempting mutation." );
995
996         if ( random.uniform( 0.0, 1.0 ) <= self.mutation_probability ):
997
998             # Create a blank offspring and fill its genes with the parent's genes.
999
1000            offspring           = Genome( );
1001            offspring.genes    = [ ];
1002            offspring.genes   = copy.deepcopy( self.population[ parent_index ].genes );
1003            offspring.fitness  = 0.0;
1004            offspring.parent_fitness = 0.0;
1005            offspring.created_by = 0;
1006
1007            # Begin to mutate.
1008
1009            mutated = False;
1010
1011            for i in range( self.number_of_genes_per_genome ):
1012
1013                # Mutate this gene by sampling a value from a normal distribution
1014                # where the mean is the current gene value and the standard deviation
1015                # is the mutation step equal to the mutation probability in the range [0,1].
1016                # A low mutation probability will give a mutated gene value closer to the original gene
1017                # value (the mean) (most of the time) as the standard deviation is small and therefore the
1018                # mutation step is small.
1019                # A high mutation probability will give a mutated gene value farther from the original gene
1020                # value (the mean) as the standard deviation is large and therefore the mutation step is large.
1021
1022                # Note that gv = gv + *N(0,1) is the same as gv = N(gv, ).
1023
1024                # Clamp the gene to range [-1,1].
1025
1026                temp_gene_value = copy.deepcopy( offspring.genes[ i ] );
1027
1028                offspring.genes[ i ] = random.gauss( offspring.genes[ i ], self.mutation_probability );
1029                offspring.genes[ i ] = get_clamped_value( offspring.genes[ i ], 0.0, 1.0 );
1030
1031                self.log( "Mutation value before/after." );
1032
1033                self.log( str( temp_gene_value ) + " " + str( offspring.genes[ i ] ) );
1034
1035                # Test if it was truly mutated.
1036
1037                if ( temp_gene_value != offspring.genes[ i ] ):
1038
1039                    mutated = True;
1040
1041                if ( mutated ): # If truly mutated.
1042
1043                    offspring.parent_fitness = self.population[ parent_index ].fitness;
1044
1045                    offspring.created_by = 2;
1046
1047                    self.log( "Returning mutated offspring." );
1048
1049                    return offspring;
1050
1051    else:
1052
1053        self.log( "Mutation did not actually take place." );
1054
1055        return 0;
1056
1057 else:

```

```

1057     self.log( "Greater than mutation probability." );
1059     return 0;
1061
1062     def crossover_then_mutate_operator( self, parent_one_index, parent_two_index ):
1063         # Crossover and mutation done sequentially as in more traditional genetic algorithms.
1064
1065         # First attempts crossover and then attempts mutation.
1066
1067         self.log( "Attempting crossover and then mutation." );
1068
1068         offspring_one = copy.deepcopy( self.population[ parent_one_index ] );
1069         offspring_two = copy.deepcopy( self.population[ parent_one_index ] );
1070
1071         offspring_one.fitness = 0.0;
1072         offspring_two.fitness = 0.0;
1073
1074         offspring_one.parent_fitness = 0.0;
1075         offspring_two.parent_fitness = 0.0;
1076
1077         offspring_one.created_by = 0;
1078         offspring_two.created_by = 0;
1079
1080         self.log( "Attempting crossover first." );
1081
1082         # Attempt crossover.
1083
1084         crossover_point = random.randint( 0, ( self.number_of_genes_per_genome - 1 ) );
1085
1086         self.log( "Crossover point." );
1087
1088         self.log( str( crossover_point ) );
1089
1090         if ( ( random.uniform( 0.0, 1.0 ) <= self.crossover_probability ) and ( parent_one_index != parent_two_index ) and ( crossover_point != 0 ) ):
1091
1092             # Cross the parent's genes in the offspring.
1093
1094             offspring_one.genes = [ ];
1095             offspring_two.genes = [ ];
1096
1097             for i in range( crossover_point ):
1098
1098                 offspring_one.genes.append( copy.deepcopy( self.population[ parent_one_index ].genes[ i ] ) );
1099                 offspring_two.genes.append( copy.deepcopy( self.population[ parent_two_index ].genes[ i ] ) );
1100
1101             for i in range( crossover_point, self.number_of_genes_per_genome ):
1102
1102                 offspring_one.genes.append( copy.deepcopy( self.population[ parent_two_index ].genes[ i ] ) );
1103                 offspring_two.genes.append( copy.deepcopy( self.population[ parent_one_index ].genes[ i ] ) );
1104
1105             parent_one_contribution = ( self.population[ parent_one_index ].fitness * ( (
1106                     crossover_point ) / ( self.number_of_genes_per_genome ) ) );
1107             parent_two_contribution = ( self.population[ parent_two_index ].fitness * ( ( self.
1108                     number_of_genes_per_genome - crossover_point ) / ( self.number_of_genes_per_genome ) ) );
1109
1110             offspring_one.parent_fitness = parent_one_contribution + parent_two_contribution;
1111             offspring_one.created_by     = offspring_one.created_by + 1;
1112
1113             parent_two_contribution = ( self.population[ parent_two_index ].fitness * ( (
1114                     crossover_point ) / ( self.number_of_genes_per_genome ) ) );
1115             parent_one_contribution = ( self.population[ parent_one_index ].fitness * ( ( self.
1116                     number_of_genes_per_genome - crossover_point ) / ( self.number_of_genes_per_genome ) ) );
1117
1118             offspring_two.parent_fitness = parent_two_contribution + parent_one_contribution;
1119             offspring_two.created_by     = offspring_two.created_by + 1;
1120
1121             self.log( "Crossover complete." );
1122
1123             self.log( str( offspring_one.created_by ) + " " + str( offspring_two.created_by ) );
1124
1125         else:
1126
1127             self.log( "Crossover failed." );
1128
1129         # Crossover may or may not have happened but now try mutation.
1130
1131         self.log( "Attempting mutation second." );
1132
1133         # Attempt to mutate offspring one.
1134
1135         self.log( "Attempting to mutate the first offspring." );
1136
1137         mutated_one = False;
1138
1139         for i in range( self.number_of_genes_per_genome ):
1140
1140             self.log( "Attempting to mutate gene: " + str( i ) );
1141
1142             # Mutate this gene by sampling a value from a normal distribution where the mean
1143             # is the current gene value and the standard deviation is mutation step equal to the
1144             # mutation probability in the range [0,1]. A low mutation probability will give a mutated gene
1145             # value close to the original gene value (the mean) (most of the time) as the standard
1146             # deviation is small and therefore the mutation step is small. A high mutation probability
1147             # will give, or it can more easily, mutate a gene value farther from the original gene value
1148             # (the mean) as the standard deviation is large and therefore the mutation step is large.
1149
1150             # Note that gv = gv +   * N( 0, 1 ) is the same as gv = N( gv,   );
1151
1152             # Clamp the gene to range [0,1].

```

```

1155     if ( random.uniform( 0.0, 1.0 ) <= self.mutation_probability ): # Mutate this gene?
1156         temp_gene_value_one = copy.deepcopy( offspring_one.genes[ i ] );
1157
1158         offspring_one.genes[ i ] = random.gauss( offspring_one.genes[ i ], self.mutation_probability );
1159         # offspring_one.genes[ i ] = gaussian_distribution( offspring_one.genes[ i ], 0.5 );
1160         # offspring_one.genes[ i ] = offspring_one.genes[ i ] + ( get_random_float( -1.0, 1.0 ) * .3 );
1161
1162         offspring_one.genes[ i ] = get_clamped_value( offspring_one.genes[ i ], 0.0, 1.0 );
1163
1164         self.log( "Mutation value before/after." );
1165
1166         self.log( str( temp_gene_value_one ) + " " + str( offspring_one.genes[ i ] ) );
1167
1168         # Test if it was truly mutated.
1169
1170         if ( temp_gene_value_one != offspring_one.genes[ i ] ):
1171             mutated_one = True;
1172
1173         else:
1174
1175             self.log( "Greater than mutation probability." );
1176
1177         # Attempt to mutate offspring two.
1178
1179         self.log( "Attempting to mutate the second offspring." );
1180
1181         mutated_two = False;
1182
1183         for i in range( self.number_of_genes_per_genome ):
1184
1185             self.log( "Attempting to mutate gene: " + str( i ) );
1186
1187             # Mutate this gene by sampling a value from a normal distribution where the mean
1188             # is the current gene value and the standard deviation is mutation step equal to the
1189             # mutation probability in the range [0,1]. A low mutation probability will give a mutated gene
1190             # value close to the original gene value (the mean) (most of the time) as the standard
1191             # deviation is small and therefore the mutation step is small. A high mutation probability
1192             # will give (or it can more easily) mutate a gene value farther from the original gene value
1193             # (the mean) as the standard deviation is large and therefore the mutation step is large.
1194
1195             # Note that gv = gv +   * N( 0, 1 ) is the same as gv = N( gv,   );
1196
1197             # Clamp the gene to range [0,1].
1198
1199             if ( random.uniform( 0.0, 1.0 ) <= self.mutation_probability ): # Mutate this gene?
1200
1201                 temp_gene_value_two = copy.deepcopy( offspring_two.genes[ i ] );
1202
1203                 offspring_two.genes[ i ] = random.gauss( offspring_two.genes[ i ], self.mutation_probability );
1204                 # offspring_two.genes[ i ] = gaussian_distribution( offspring_two.genes[ i ], 0.5 );
1205                 # offspring_two.genes[ i ] = offspring_two.genes[ i ] + ( get_random_float( -1.0, 1.0 ) * .3 );
1206
1207                 offspring_two.genes[ i ] = get_clamped_value( offspring_two.genes[ i ], 0.0, 1.0 );
1208
1209                 self.log( "Mutation value before/after." );
1210
1211                 self.log( str( temp_gene_value_two ) + " " + str( offspring_one.genes[ i ] ) );
1212
1213                 # Test if it was truly mutated.
1214
1215                 if ( temp_gene_value_two != offspring_two.genes[ i ] ):
1216                     mutated_two = True;
1217
1218                 else:
1219
1220                     self.log( "Greater than mutation probability." );
1221
1222         if ( mutated_one ): # If truly mutated.
1223
1224             # Mutation = 2, crossover = 1, crossover + mutation = 3.
1225
1226             offspring_one.created_by = offspring_one.created_by + 2;
1227
1228             # If this offspring was only mutated, that is, it was not crossed then it gets its parent fitness.
1229             # If it was crossed before being mutated then offspring_one.created_by would equal 3.
1230
1231             if ( offspring_one.created_by == 2 ):
1232
1233                 self.log( "Offspring one only mutated." );
1234
1235                 offspring_one.parent_fitness = copy.deepcopy( self.population[ parent_one_index ].fitness );
1236
1237             if ( mutated_two ): # If truly mutated.
1238
1239                 offspring_two.created_by = offspring_two.created_by + 2;
1240
1241                 # If this offspring was only mutated, that is, it was not crossed then it gets its parent fitness.
1242                 # If it was crossed before being mutated then offspring_two.created_by would equal 3.
1243
1244                 if ( offspring_two.created_by == 2 ):
1245
1246                     self.log( "Offspring two only mutated." );
1247
1248                     offspring_two.parent_fitness = copy.deepcopy( self.population[ parent_two_index ].fitness );
1249
1250
1251             # No parents->offspring not crossed and/or not mutated enter into the new population.
1252             # Each offspring going into the new population must either be crossed, mutated, or both.
1253
1254
1255

```

```

1257     if ( ( offspring_one.created_by == 0 ) or ( offspring_two.created_by == 0 ) ):
1258         self.log( "No crossover and/or mutation occurred." );
1259         return 0;
1260
1261     elif ( ( offspring_one.genes == self.population[ parent_one_index ].genes ) or ( offspring_two.genes
1262           == self.population[ parent_two_index ].genes ) ):
1263
1264         self.log( "Offspring genes not different from parents." );
1265
1266         return 0;
1267
1268     else:
1269
1270         self.log( "Returning offspring." );
1271
1272         return { "one": offspring_one, "two": offspring_two };
1273
1274 def reset_population_metrics( self ):
1275
1276     self.total_fitness      = 0.0;
1277     self.highest_fitness    = 0.0;
1278     self.lowest_fitness     = 0.0;
1279     self.average_fitness   = 0.0;
1280     self.fittest_genome_index = -1;
1281     self.weakest_genome_index = -1;
1282
1283 def update_population_metrics( self ):
1284
1285     # 0.0 is the highest fitness
1286     # 1.7976931348623157e+308 is lowest fitness.
1287
1288     self.reset_population_metrics();
1289
1290     highest_so_far = self.population[ 0 ].fitness;
1291     lowest_so_far = self.population[ 0 ].fitness;
1292
1293     self.fittest_genome_index = 0;
1294     self.weakest_genome_index = 0;
1295
1296     self.total_fitness = self.population[ 0 ].fitness;
1297     self.highest_fitness = self.population[ 0 ].fitness;
1298     self.lowest_fitness = self.population[ 0 ].fitness;
1299
1300     for i in range( self.population_size ):
1301
1302         # Update fittest if necessary.
1303
1304         if ( highest_so_far > self.population[ i ].fitness ):
1305
1306             highest_so_far = self.population[ i ].fitness;
1307
1308             self.fittest_genome_index = i;
1309
1310             self.highest_fitness = highest_so_far;
1311
1312         # Update worst if necessary.
1313
1314         if ( lowest_so_far < self.population[ i ].fitness ):
1315
1316             lowest_so_far = self.population[ i ].fitness;
1317
1318             self.weakest_genome_index = i;
1319
1320             self.lowest_fitness = lowest_so_far;
1321
1322             self.total_fitness += self.population[ i ].fitness;
1323
1324         # Next genome.
1325
1326         self.average_fitness = self.total_fitness / float( self.population_size );
1327
1328         self.log( "Population metrics. T H L A Fi Wi." );
1329
1330         self.log( str( self.total_fitness ) );
1331         self.log( str( self.highest_fitness ) );
1332         self.log( str( self.lowest_fitness ) );
1333         self.log( str( self.average_fitness ) );
1334         self.log( str( self.fittest_genome_index ) );
1335         self.log( str( self.weakest_genome_index ) );
1336
1337 def compute_population_makeup( self ):
1338
1339     randoms      = 0;
1340     crossovers   = 0;
1341     mutants      = 0;
1342     crossover_mutants = 0;
1343     elites       = 0;
1344
1345     for i in range( self.population_size ):
1346
1347         if ( self.population[ i ].created_by == 0 ):
1348
1349             randoms = randoms + 1;
1350
1351         elif ( self.population[ i ].created_by == 1 ):
1352
1353             crossovers = crossovers + 1;
1354
1355         elif ( self.population[ i ].created_by == 2 ):
1356
1357             mutants = mutants + 1;
1358
1359         elif ( self.population[ i ].created_by == 3 ):
1360
1361             crossover_mutants = crossover_mutants + 1;
1362
1363         elif ( self.population[ i ].created_by == 4 ):
1364
1365             elites = elites + 1;
1366
1367         else:
1368
1369             print( "Unknown created_by value: " + str( self.population[ i ].created_by ) );
1370
1371     print( "Randoms: " + str( randoms ) );
1372     print( "Crossovers: " + str( crossovers ) );
1373     print( "Mutants: " + str( mutants ) );
1374     print( "Crossover Mutants: " + str( crossover_mutants ) );
1375     print( "Elites: " + str( elites ) );
1376
1377     print( "Total Population Size: " + str( self.population_size ) );
1378
1379     print( "Randoms Percentage: " + str( ( randoms / self.population_size ) * 100 ) );
1380     print( "Crossovers Percentage: " + str( ( crossovers / self.population_size ) * 100 ) );
1381     print( "Mutants Percentage: " + str( ( mutants / self.population_size ) * 100 ) );
1382     print( "Crossover Mutants Percentage: " + str( ( crossover_mutants / self.population_size ) * 100 ) );
1383     print( "Elites Percentage: " + str( ( elites / self.population_size ) * 100 ) );

```

```

1357         mutants = mutants + 1;
1359     elif ( self.population[ i ].created_by == 3 ):
1361         crossover_mutants = crossover_mutants + 1;
1363     elif ( self.population[ i ].created_by == 4 ):
1365         elites = elites + 1;
1367     self.population_makeup = str( randoms ) + " " + str( crossovers ) + " " + str( mutants ) + " " + str(
1368         crossover_mutants ) + " " + str( elites );
1369     self.log( "Population makeup. R C M CM E." );
1371     self.log( self.population_makeup );
1373 def adapt_crossover_and_mutation_probabilities( self ):
1375     # Calculate the crossover and mutation operators' progress where
1376     # their progress is based on how well they produced offspring that
1377     # had a better fitness than their parent.
1379     self.log( "Adapting crossover and mutation probabilities." );
1381     crossover_operator_progress_sum = 0.0;
1382     number_of_crossovers = 0;
1383     mutation_operator_progress_sum = 0.0;
1384     number_of_mutations = 0;
1387     # Sum all of the progresses.
1389     # Since the GA is looking to minimize the fitness function, progress is when the offspring has a lower
1390     # fitness score
1391     # than its parent.
1392     for i in range( self.population_size ):
1393         if ( self.population[ i ].created_by == 1 ): # Created by crossover.
1394             crossover_operator_progress_sum += ( self.population[ i ].parent_fitness - self.population[ i ].fitness );
1395             number_of_crossovers += 1;
1396         elif ( self.population[ i ].created_by == 2 ): # Created by mutation.
1397             mutation_operator_progress_sum += ( self.population[ i ].parent_fitness - self.population[ i ].fitness );
1398             number_of_mutations += 1;
1400     # Now calculate the average crossover and mutation progress for the population.
1401     self.crossover_operator_progress_average = 0.0;
1402     self.mutation_operator_progress_average = 0.0;
1403     if ( number_of_crossovers != 0 ):
1404         self.crossover_operator_progress_average = ( crossover_operator_progress_sum ) / float(
1405             number_of_crossovers );
1406     if ( number_of_mutations != 0 ):
1407         self.mutation_operator_progress_average = ( mutation_operator_progress_sum ) / float(
1408             number_of_mutations );
1409     # Adjust crossover and mutation rate adjustments.
1410     self.log( "Adjusting crossover and mutation rate adjustments." );
1411     if ( self.lowest_fitness > self.highest_fitness ):
1412         self.log( "L > H" );
1413         self.crossover_probability_adjustment = 0.01 * ( ( self.lowest_fitness - self.average_fitness ) / (
1414             self.lowest_fitness - self.highest_fitness ) );
1415         self.mutation_probability_adjustment = 0.01 * ( ( self.lowest_fitness - self.average_fitness ) / (
1416             self.lowest_fitness - self.highest_fitness ) );
1417         self.log( str( self.crossover_probability_adjustment ) + " " + str( self.
1418             mutation_probability_adjustment ) );
1419     elif ( self.lowest_fitness == self.average_fitness ):
1420         self.log( "L == A" );
1421         self.crossover_probability_adjustment = 0.01;
1422         self.mutation_probability_adjustment = 0.01;
1423         self.log( "Crossover progress average." );
1424         self.log( str( self.crossover_operator_progress_average ) );
1425         self.log( "Mutation progress average." );
1426         self.log( str( self.mutation_operator_progress_average ) );
1427     # Adjust crossover and mutation rates.

```

```

1451     if ( self.crossover_operator_progress_average > self.mutation_operator_progress_average ):
1452         self.log( "Adjusting crossover/mutation probabilities. CPA > MPA." );
1453         self.crossover_probability = self.crossover_probability + self.crossover_probability_adjustment;
1454         self.mutation_probability = self.mutation_probability - self.mutation_probability_adjustment;
1455     elif ( self.crossover_operator_progress_average < self.mutation_operator_progress_average ):
1456         self.log( "Adjusting crossover/mutation probabilities. CPA < MPA." );
1457         self.crossover_probability = self.crossover_probability - self.crossover_probability_adjustment;
1458         self.mutation_probability = self.mutation_probability + self.mutation_probability_adjustment;
1459     elif ( self.crossover_operator_progress_average == self.mutation_operator_progress_average ):
1460         self.log( "Not adjusting crossover/mutation probabilities. CPA == MPA." );
1461         # Do not adjust.
1462         pass;
1463     self.crossover_probability = get_clamped_value( self.crossover_probability, self.
1464         crossover_probability_minimum, 1.0 - self.mutation_probability_minimum );
1465     self.mutation_probability = get_clamped_value( self.mutation_probability, self.
1466         mutation_probability_minimum, 1.0 - self.crossover_probability_minimum );
1467
1468 def sort_population( self, descending = None ):
1469     self.log( "Sorting population. Descending: " + str( descending ) );
1470     if ( descending == None or descending == False ):
1471         self.population = sorted( self.population, key = lambda genome: genome.fitness, reverse = False );
1472     elif ( descending == True ):
1473         self.population = sorted( self.population, key = lambda genome: genome.fitness, reverse = True );
1474     self.log( "Sorted." );
1475     self.log( str( self.population[ 0 ].fitness ) + " " + str( self.population[ -1 ].fitness ) );
1476
1477 def generate_new_generation( self ):
1478     self.log( "Creating a new generation." );
1479
1480     # Sort the population based on fitness in ascending order.
1481     # 0.0 is the highest fitness and infinity is the lowest fitness.
1482     # So sort in non-increasing order.
1483
1484     self.log( "Sorting population." );
1485     self.sort_population( descending = True );
1486
1487     # Update population metrics.
1488
1489     self.log( "Updating population metrics." );
1490     self.update_population_metrics();
1491
1492     # Adapt crossover and mutation probabilities if using self-adaptation.
1493
1494     if ( self.use_self_adaptation == True ):
1495         self.log( "Adapting crossover/mutation probabilities." );
1496         self.adapt_crossover_and_mutation_probabilities();
1497
1498     # Calculate current population makeup.
1499
1500     self.log( "Computing population makeup." );
1501     self.compute_population_makeup();
1502
1503     # Create a temporary population to store newly created generation.
1504
1505     self.log( "Creating empty new population." );
1506     new_population = [ ];
1507
1508     # Allow the top N elite to pass into the next generation.
1509
1510     self.log( "Performing elitism." );
1511     self.elitism_operator( new_population );
1512
1513     # Perform crossover and mutation separately?
1514
1515     if ( not self.perform_crossover_and_mutation_sequentially ):
1516         self.log( "Performing crossover and mutation separately." );
1517
1518     self.log( "Entering the loop." );
1519
1520     # Now we enter the GA loop.
1521
1522     # Repeat until a new population is generated.

```

```

1551     while ( len( new_population ) < self.population_size ):
1553         # Perform crossover and mutation separately.
1555         # Try to generate an offspring via crossover first.
1557         self.total_number_of_crossover_attempts += 1;
1559         # Select two genome indexes.
1561         parents = self.selection_operator( 2 );
1563         crossover_offspring = self.crossover_operator( parents[ 0 ], parents[ 1 ] );
1565         if ( crossover_offspring != 0 ):
1567             self.log( "Adding crossover offspring." );
1569             new_population.append( crossover_offspring );
1571             self.total_number_of_crossovers += 1;
1573             # There is the possibility of adding up to two
1574             # offspring per while loop.
1575             # Don't create more than the population size.
1577             if ( len( new_population ) == self.population_size ):
1579                 break;
1581             # Try to generate an offspring via mutation second.
1583             self.total_number_of_mutation_attempts += 1;
1585             # Select one genome index.
1587             parent = self.selection_operator( 1 );
1589             mutation_offspring = self.mutation_operator( parent[ 0 ] );
1591             if ( mutation_offspring != 0 ):
1593                 self.log( "Adding mutation offspring." );
1595                 new_population.append( mutation_offspring );
1597                 self.total_number_of_mutations += 1;
1599             assert len( new_population ) == self.population_size, "New population size does not equal population
size setting.";
1601             # Finished so assign new pop to the current population.
1603             self.population = [ ];
1604             self.population = copy.deepcopy( new_population );
1605             new_population = [ ];
1607     else: # Perform crossover and mutation in sequence.
1609         self.log( "Performing crossover and mutation in sequence." );
1611         self.log( "Entering the loop." );
1613         # Now we enter the GA loop.
1615         # Repeat until a new population is generated.
1617         while ( len( new_population ) < self.population_size ):
1619             # Attempt crossover and then mutation in sequence.
1621             parents = self.selection_operator( 2 );
1623             self.log( "Parents selected." );
1625             self.log( str( parents ) );
1627             offspring = self.crossover_then_mutate_operator( parents[ 0 ], parents[ 1 ] );
1629             if ( offspring != 0 ):
1631                 self.log( "Adding first offspring." );
1633                 # First offspring.
1635                 if ( offspring[ "one" ].created_by == 1 ):
1637                     self.log( "Adding crossover offspring." );
1639                     self.total_number_of_crossovers += 1;
1640                     self.total_number_of_crossover_attempts += 1;
1641                     self.total_number_of_mutation_attempts += 1;
1643                     new_population.append( offspring[ "one" ] );
1645                 elif ( offspring[ "one" ].created_by == 2 ):
1647                     self.log( "Adding mutation offspring." );
1649                     self.total_number_of_mutations += 1;
                     self.total_number_of_crossover_attempts += 1;

```

```

1651         self.total_number_of_mutation_attempts += 1;
1653         new_population.append( offspring[ "one" ] );
1655     elif ( offspring[ "one" ].created_by == 3 ):
1657         self.log( "Adding crossed over and mutated offspring." );
1659         self.total_number_of_crossovers += 1;
1660         self.total_number_of_mutations += 1;
1661         self.total_number_of_crossover_attempts += 1;
1662         self.total_number_of_mutation_attempts += 1;
1663         new_population.append( offspring[ "one" ] );
1665     # Old population size should match this new population size.
1667     if ( len( new_population ) == self.population_size ):
1669         break;
1671     self.log( "Adding second offspring." );
1673     # Second offspring.
1675     if ( offspring[ "two" ].created_by == 1 ):
1677         self.log( "Adding crossover offspring." );
1679         self.total_number_of_crossovers += 1;
1680         self.total_number_of_crossover_attempts += 1;
1681         self.total_number_of_mutation_attempts += 1;
1683         new_population.append( offspring[ "two" ] );
1685     elif ( offspring[ "two" ].created_by == 2 ):
1687         self.log( "Adding mutation offspring." );
1689         self.total_number_of_mutations += 1;
1690         self.total_number_of_crossover_attempts += 1;
1691         self.total_number_of_mutation_attempts += 1;
1693         new_population.append( offspring[ "two" ] );
1695     elif ( offspring[ "two" ].created_by == 3 ):
1697         self.log( "Adding crossed over and mutated offspring." );
1699         self.total_number_of_crossovers += 1;
1700         self.total_number_of_mutations += 1;
1701         self.total_number_of_crossover_attempts += 1;
1702         self.total_number_of_mutation_attempts += 1;
1703         new_population.append( offspring[ "two" ] );
1705     new_population.append( offspring[ "two" ] );
1707 assert len( new_population ) == self.population_size, "New population size does not equal population size setting.";
1709 # Finished so assign new pop to the current population.
1711 self.population = [ ];
1712 self.population = copy.deepcopy( new_population );
1713 new_population = [ ];
1715 # Calculate the observed rates.
1717 self.observed_crossover_rate = self.total_number_of_crossovers / self.
1718     total_number_of_crossover_attempts;
1719 self.observed_mutation_rate = self.total_number_of_mutations / self.
1720     total_number_of_mutation_attempts;
1721 self.log( "Observed cross/mut rates." );
1722 self.log( str( self.observed_crossover_rate ) + " " + str( self.observed_mutation_rate ) );
1723 # Advance generation counter.
1725 self.generation_number += 1;
1727 ,,
1729 The BBAutoTune object.
1731 ,,
1733 class BBAutoTune( ):
1735     def __init__( self, ga, dbm ):
1737         # The genetic algorithm and the database manager.
1739         self.ga = ga;
1740         self.dbm = dbm;
1742         # Run ID.
1744         self.run_id = None;
1746         # The current genome being evaluated.
1748         self.current_genome = 0;

```

```

1751     # Log debug statements.
1753     self.debug = False
1755     self.log_file_name = "";
1757     # Open a browser window to the GA monitor.
1759     self.open_ga_monitor_browser_window = False;
1761     # Load in real robot data.
1763     real_forward_x_primes = pickle.load( open( get_scripts_location() + "data/real_robot_motion/forward/
1764         x_p_values.pkl", "rb" ) );
1765     real_forward_y_primes = pickle.load( open( get_scripts_location() + "data/real_robot_motion/forward/
1766         y_p_values.pkl", "rb" ) );
1767     real_forward_t_primes = pickle.load( open( get_scripts_location() + "data/real_robot_motion/forward/
1768         t_p_values.pkl", "rb" ) );
1769     real_forward_motion = [ ];
1770     for i in range( len( real_forward_x_primes ) ):
1771         x = real_forward_x_primes[ i ];
1772         y = real_forward_y_primes[ i ];
1773         t = real_forward_t_primes[ i ];
1774         real_forward_motion.append(
1775             [ x, y, t ]
1776         );
1777     real_forward_motion = numpy.array( real_forward_motion );
1778     # Calculate the robust covariance matrix and the robust mean (location).
1779     self.mcd_fitted = sklearn.covariance.MinCovDet( assume_centered = False, support_fraction = 0.5 * (
1780         len( real_forward_motion ) + 3.0 + 1.0 ) ).fit( real_forward_motion );
1781     self.rcm = self.mcd_fitted.covariance_;
1782     self.rcm_inv = numpy.linalg.inv( self.rcm );
1783     self.rm = self.mcd_fitted.location_;
1784     # The threshold for Chi-square with 3 degrees of freedom (x,y,t) and an alpha value of 0.005.
1785     self.genome_fitness_threshold = math.sqrt( scipy.stats.chi2.isf( 1.0 - 0.995, 3 ) );
1786     def start(
1787         self,
1788         population_size,
1789         max_generations,
1790         number_of_elite,
1791         crossover_probability,
1792         mutation_probability,
1793         max_torque,
1794         use_rank_fitness_selection,
1795         perform_crossover_and_mutation_sequentially,
1796         use_self_adaptation,
1797         open_ga_monitor_browser_window,
1798         debug
1799     ):
1800         self.run_id = int( round( time.time() * 1000 ) );
1801         self.debug = debug or False;
1802         if ( self.debug == True ):
1803             logs_location = get_scripts_location() + "logs/";
1804             self.log_file_name = logs_location + "log_" + str( self.run_id ) + ".dat";
1805             self.ga.set_log_file_name( self.log_file_name );
1806             bpy.data.objects[ "robot_monitor" ].game.properties[ "log_file_name" ].value = self.log_file_name;
1807             self.log( "Run ID." );
1808             self.log( str( self.run_id ) );
1809             # Pass the file name and directory where the robot
1810             # monitor will record the robot's initial and final state.
1811             self.log( "Setting the robot monitor's shared data file name." );
1812             bpy.data.objects[ "robot_monitor" ].game.properties[ "shared_data_file_name" ].value =
1813                 get_scripts_location() + "shared_data/genome_I.F.dat";
1814             self.log( "Connecting to the database." );
1815             self.dbm.connect_to_database();
1816             self.log( "Starting the GA manager." );
1817             self.open_ga_monitor_browser_window = open_ga_monitor_browser_window;

```

```

1847     self.start_ga_monitor( );
1849     self.log( "Setting the GA parameters. PS MG NE CP MP URFS PCMS USA." );
1851     self.log( str( population_size ) );
1852     self.log( str( max_generations ) );
1853     self.log( str( number_of_elite ) );
1854     self.log( str( crossover_probability ) );
1855     self.log( str( mutation_probability ) );
1856     self.log( str( use_rank_fitness_selection ) );
1857     self.log( str( perform_crossover_and_mutation_sequentially ) );
1858     self.log( str( use_self_adaptation ) );
1859
1860     self.log( "Max torque." );
1861     self.max_torque = max_torque;
1862
1863     self.log( str( self.max_torque ) );
1864
1865     self.ga.set_population_size( population_size );
1866     self.ga.set_max_generations( max_generations );
1867     self.ga.set_number_of_elite( number_of_elite );
1868     self.ga.set_crossover_probability( crossover_probability );
1869     self.ga.set_mutation_probability( mutation_probability );
1870     self.ga.set_use_rank_fitness_selection( use_rank_fitness_selection );
1871     self.ga.set_perform_crossover_and_mutation_sequentially( perform_crossover_and_mutation_sequentially )
1872     ;
1873     self.ga.set_use_self_adaptation( use_self_adaptation );
1874
1875     self.log( "Creating a random GA population." );
1876
1877     self.ga.create_randomized_population( );
1878
1879     self.log( "Setting the current genome to 0." );
1880
1881     self.current_genome = 0;
1882
1883     self.log( "Entering loop." );
1884
1885     while ( self.ga.get_generation_number( ) < self.ga.get_max_generations( ) ):
1886
1887         self.log( "Current genome." );
1888
1889         self.log( str( self.ga.get_genome( self.current_genome ) ) );
1890
1891         # Populate physics engine parameters.
1892
1893         self.log( "Populating the physics engine parameters." );
1894
1895         self.populate_physics_engine_parameters( self.ga.get_genome( self.current_genome ).get_genes( ) );
1896
1897         # Run game engine.
1898
1899         self.log( "Starting the game engine." );
1900
1901         self.start_game_engine( );
1902
1903         # Calculate current genome fitness.
1904
1905         # Read in I=(x_pos,y_pos,z_pos,x_ori,y_ori,z_ori,s_time) and F=(x_pos,y_pos,z_pos,x_ori,y_ori,z_ori,
1906         # e_time)
1907         # which was recorded by the robot monitor while the game engine was running.
1908
1909         self.log( "Game engine stopped." );
1910
1911         self.log( "Getting genome (I)initial and (F)inal state." );
1912         self.log( "x_pos , y_pos , z_pos , x_ori , y_ori , z_ori , s/e_time" );
1913
1914         shared_data_file = open( get_scripts_location( ) + "shared_data/genome_I_F.dat", "r" );
1915
1916         I = shared_data_file.readline( ).rstrip( );
1917         I = I.split( " " );
1918         I[ 0 ] = float( I[ 0 ] ); # x position.
1919         I[ 1 ] = float( I[ 1 ] ); # y position.
1920         I[ 2 ] = float( I[ 2 ] ); # z position.
1921         I[ 3 ] = float( I[ 3 ] ); # x orientation.
1922         I[ 4 ] = float( I[ 4 ] ); # y orientation.
1923         I[ 5 ] = float( I[ 5 ] ); # z orientation.
1924         I[ 6 ] = float( I[ 6 ] ); # Start time.
1925
1926         F = shared_data_file.readline( ).rstrip( );
1927         F = F.split( " " );
1928         F[ 0 ] = float( F[ 0 ] ); # x' position.
1929         F[ 1 ] = float( F[ 1 ] ); # y' position.
1930         F[ 2 ] = float( F[ 2 ] ); # z' position.
1931         F[ 3 ] = float( F[ 3 ] ); # x' orientation.
1932         F[ 4 ] = float( F[ 4 ] ); # y' orientation.
1933         F[ 5 ] = float( F[ 5 ] ); # z' orientation.
1934         F[ 6 ] = float( F[ 6 ] ); # End time.
1935
1936         shared_data_file.close( );
1937
1938         os.remove( get_scripts_location( ) + "shared_data/genome_I_F.dat" );
1939
1940         # Record simulated robot motion and its resulting fitness.
1941
1942         simulated_robot_motion_file = open( get_scripts_location( ) + "data/"
1943         simulated_robot_motion_file = open( get_scripts_location( ) + "data/" + "simulated_robot_motion_with_fitness/forward/" + "srmfwf_" + str( self.run_id ) + ".dat", "a" );
1944
1945         write_string = str( I[ 0 ] ) + ",";
1946         write_string += str( I[ 1 ] ) + ",";
1947         write_string += str( I[ 2 ] ) + ",";

```

```

1947     write_string += str( I[ 3 ] ) + ",";
1948     write_string += str( I[ 4 ] ) + ",";
1949     write_string += str( I[ 5 ] ) + ";";
1950
1951     write_string += str( F[ 0 ] ) + ",";
1952     write_string += str( F[ 1 ] ) + ",";
1953     write_string += str( F[ 2 ] ) + ",";
1954     write_string += str( F[ 3 ] ) + ",";
1955     write_string += str( F[ 4 ] ) + ",";
1956     write_string += str( F[ 5 ] ) + ";";
1957
1958     simulated_robot_motion_file.write( write_string );
1959
1960     simulated_robot_motion_file.close();
1961
1962     self.log( str( I ) );
1963
1964     self.log( str( F ) );
1965
1966     self.log( "Calculating genome fitness." );
1967
1968     current_genome_fitness = self.calculate_genome_fitness( I, F );
1969
1970     self.log( "Genome fitness." );
1971
1972     self.log( str( current_genome_fitness ) );
1973
1974     self.ga.set_genome_fitness( self.current_genome, current_genome_fitness );
1975
1976     # Add the fitness to the last line of the simulated robot motion recorded.
1977
1978     simulated_robot_motion_file = open( get_scripts_location() + "data/
1979                                         simulated_robot_motion_with_fitness/forward/" + "srwf_" + str( self.run_id ) + ".dat", "a" );
1980
1981     simulated_robot_motion_file.write( str( current_genome_fitness ) + "\n" );
1982
1983     simulated_robot_motion_file.close();
1984
1985     # Record the genome's phenotype (the physics parameters) and its corresponding fitness.
1986
1987     physics_parameters_with_fitness_file = open( get_scripts_location() + "data/
1988                                         physics_parameters_with_fitness/" + "ppwf_" + str( self.run_id ) + ".dat", "a" );
1989
1990     physics_parameters_with_fitness_file.write( "fitness," + str( current_genome_fitness ) + "\n\n" );
1991
1992     physics_parameters_with_fitness_file.close();
1993
1994     # Increase current genome + 1.
1995
1996     self.current_genome += 1;
1997
1998     self.log( str( self.current_genome ) );
1999
2000     # If current genome is equal to the population size.
2001
2002     if ( self.current_genome == self.ga.get_population_size() ):
2003
2004         self.log( "Evaluated all genomes in population." );
2005
2006         # Separate recorded simulated robot motion by generation.
2007
2008         simulated_robot_motion_file = open( get_scripts_location() + "data/
2009                                         simulated_robot_motion_with_fitness/forward/" + "srwf_" + str( self.run_id ) + ".dat", "a" );
2010
2011         simulated_robot_motion_file.write( "\n" );
2012
2013         simulated_robot_motion_file.close();
2014
2015         self.log( "Current generation number: " + str( self.ga.get_generation_number() ) );
2016
2017         # Update the population metrics.
2018
2019         self.log( "Updating population metrics." );
2020
2021         self.ga.update_population_metrics();
2022
2023         # Store the population metrics in the database.
2024
2025         self.log( "Storing population metrics in the database." );
2026
2027         a = str( self.ga.get_generation_number() );
2028         b = str( self.ga.get_highest_fitness() );
2029         c = str( self.ga.get_average_fitness() );
2030         d = str( self.ga.get_lowest_fitness() );
2031         e = str( self.ga.get_crossover_probability() );
2032         f = str( self.ga.get_mutation_probability() );
2033
2034         mysql_string = "INSERT INTO `population_metrics` ( `generation_number`, `highest_fitness`, `average_fitness`, `lowest_fitness`, `crossover_probability`, `mutation_probability` ) ";
2035         mysql_string += "VALUES( " + a + ", " + b + ", " + c + ", " + d + ", " + e + ", " + f + " );";
2036
2037         self.dbm.execute( mysql_string );
2038
2039         self.log( mysql_string );
2040
2041         # Generate a new population.
2042
2043         self.log( "Generating a new generation." );
2044         self.ga.generate_new_generation();

```

```

2045     # Set current genome to 0.
2047     self.log( "Setting the current genome to 0." );
2049     self.current_genome = 0;
2051     self.log( "Exited loop." );
2053     self.log( "Stopping." );
2055     self.stop( );
2057     def stop( self ):
2059         if ( self.run_id != None ):
2061             self.dbm.close_database_connection( );
2063             self.stop_ga_monitor( );
2065             self.run_id = None;
2067     def start_ga_monitor( self ):
2069         scripts_location = get_scripts_location( );
2071         if ( self.open_ga_monitor_browser_window == True ):
2073             self.cgi_http_server = subprocess.Popen( [ scripts_location + "cgi_http_server.py", "-w" ] );
2075         else:
2077             self.cgi_http_server = subprocess.Popen( scripts_location + "cgi_http_server.py" );
2079     def stop_ga_monitor( self ):
2081         self.cgi_http_server.kill( );
2083     def start_game_engine( self ):
2085         bpy.ops.view3d.game_start( );
2087     def log( self, log_string ):
2089         if ( self.log_file_name != "" ):
2091             log_file = open( self.log_file_name, "a+" );
2093             log_file.write( log_string + "\n" );
2095             log_file.close( );
2097     def calculate_mahalanobis_distance( self, point ):
2099         self.log( "Point." );
2101         self.log( str( point ) );
2103         md = scipy.spatial.distance.mahalanobis( point, self.rm, self.rcm_inv );
2105         md2 = md * md;
2107         return md, md2;
2109     def calculate_genome_fitness( self, initial, final ):
2111         # Blender returns NaN for large positions/orientations for x, y, and z.
2112         # If this is the case, set the fitness to some large value.
2113         # initial/final structure:
2114         #   x_pos   0
2115         #   y_pos   1
2116         #   z_pos   2
2117         #   x_rot   3
2118         #   y_rot   4
2119         #   z_rot   5
2120         #   s/e_tim 6
2121
2122         fitness = 9999999999.0;
2123
2124         for i in range( len( final ) ):
2125             if ( numpy.isnan( final[ i ] ) ):
2126                 return fitness;
2127
2128             # Only 3 dof was recorded for the real robot.
2129             # So assemble for the simulated robot its x' position, y' position, and z' orientation.
2130
2131             final_trimmed = [ final[ 0 ], final[ 1 ], final[ 5 ] ];
2132
2133             md, md2 = self.calculate_mahalanobis_distance( final_trimmed );
2134
2135             self.log( "Genome mahalanobis distance." );
2136             self.log( str( md ) );
2137
2138             # Penalties?
2139             self.log( "Penalties." );
2140
2141
2142
2143
2144
2145

```

```

2147     # Time > 1 seconds.
2148     elapsed_time = abs( ( final[ 6 ] - initial[ 6 ] ) / 1000.0 ) - 1.0 );
2149     self.log( "Elapsed time > 1 seconds: " + str( elapsed_time ) );
2150
2151     # Rotation in x.
2152     rotation_x = abs( final[ 3 ] - initial[ 3 ] );
2153     self.log( "X rotation: " + str( rotation_x ) );
2154
2155     # Rotation in y.
2156     rotation_y = abs( final[ 4 ] - initial[ 4 ] );
2157     self.log( "Y rotation: " + str( rotation_y ) );
2158
2159     # Translation in z.
2160     translation_z = abs( final[ 2 ] - initial[ 2 ] );
2161     self.log( "Z translation: " + str( translation_z ) );
2162
2163     fitness = md + elapsed_time + rotation_x + rotation_y + translation_z;
2164
2165     return fitness;
2166
2167 def populate_physics_engine_parameters( self, genome_genes ):
2168
2169     self.log( "Genome genes." );
2170
2171     self.log( str( genome_genes ) );
2172
2173     assert len( genome_genes ) == self.ga.get_number_of_genes_per_genome( ), "Cannot populate physics
2174         engine parameters.";
2175
2176     # Blender API call examples:
2177
2178     #
2179     # bpy.data.objects["Cylinder"].game.actuators["Motion"].torque = [0,401,0];
2180     # bpy.data.objects["Cylinder"].game.sensors["Always"].use_tap = False;
2181     # bpy.ops.logic.actuator_add( type="MOTION", name="motion1", object="Cylinder");
2182     # bpy.data.objects["Cylinder"].game.controllers[ "Python" ].link( sensor=None, actuator=bpy.data.
2183         objects["Cylinder"].game.actuators["motion1"] );
2184     # bpy.data.objects["Cylinder"].game.mass = 10000.0;
2185     # bpy.data.scenes["Scene"].game.settings.physics_gravity;
2186
2187
2188     # Blender's largest number called "inf".
2189     # To find, set an appropriate field to a very large
2190     # number and then click on the field and copy the number.
2191     # When clicking the field, the "inf" will turn numeric.
2192
2193     INF = 340282346638528859811704183484516925440.0;
2194
2195     BOOLEANS = [ False, True ];
2196
2197
2198     # Assumes the correct scene, wheel object, sensor, controller, and actuator names.
2199     # These names were set by hand in the .blend file. If they are changed, Blender
2200     # will throw an exception.
2201
2202     scene      = "bbautotune";
2203     front_wheel_l = "robot_1_wheel_front_L";
2204     front_wheel_r = "robot_1_wheel_front_R";
2205     back_wheel_l = "robot_1_wheel_back_L";
2206     back_wheel_r = "robot_1_wheel_back_R";
2207     wheel_material = "wheel";
2208     actuator      = "torque_z";
2209
2210
2211     ### WORLD
2212
2213     # Gravity.
2214
2215     #gravity = get_clamped_value( ( genome_genes[ 0 ] * 10000.0 ), 0.0, 10000.0 );
2216
2217     gravity = get_clamped_value( ( genome_genes[ 0 ] * 15.0 ), 0.0, 100.0 );
2218
2219     self.log( "Setting gravity." );
2220
2221     self.log( str( gravity ) );
2222
2223     bpy.data.scenes[ scene ].game_settings.physics_gravity = gravity;
2224
2225     # Sub-steps.
2226
2227     # You can input 1 up to 50. However, if only sliding the values, the value only goes from 1 to 5.
2228     # Not sure if this is a bug in the Blender code.
2229
2230     sub_steps = get_clamped_value( math.floor( ( genome_genes[ 1 ] * ( 5 - 1 ) ) + 1 ), 1, 5 );
2231
2232     self.log( "Setting sub steps." );
2233
2234     self.log( str( sub_steps ) );
2235
2236     bpy.data.scenes[ scene ].game_settings.physics_step_sub = sub_steps;
2237
2238     # FPS.
2239
2240     # Setting FPS within [1,30) makes the game engine run extremely slow so allow the GA to find a
2241     # solution
2242     # within the range of 30 to 10000 FPS.
2243
2244     fps = get_clamped_value( math.floor( ( genome_genes[ 2 ] * ( 10000 - 30 ) ) + 30 ), 30, 10000 );

```

```

2245     self.log( "Setting FPS." );
2247     self.log( str( fps ) );
2249     bpy.data.scenes[ scene ].game_settings.fps = fps;
2251     ### Object
2253     # Scale XYZ?
2255     ...
2257     scale = get_clamped_value( genome_genes[ 3 ] * INF, 0.0, INF );
2259     bpy.data.objects[ front_wheel_l ].scale = [ scale, scale, scale ];
2261     bpy.data.objects[ front_wheel_r ].scale = [ scale, scale, scale ];
2263     bpy.data.objects[ back_wheel_l ].scale = [ scale, scale, scale ];
2265     bpy.data.objects[ back_wheel_r ].scale = [ scale, scale, scale ];
2266     ...
2267     ### MATERIAL
2269     # Use physics?
2271     # Round returns incorrect values but converting its return value to a string does return the right
2272     # value.
2273     # Convert only the numbers before the decimal '.' to an integer.
2274     ...
2275     index = int( str( round( genome_genes[ 4 ] ) ).split( "." )[ 0 ] );
2277     use_material_physics = BOOLEANS[ index ];
2279     self.log( "Setting use material physics." );
2281     self.log( str( use_material_physics ) );
2283     bpy.data.objects[ front_wheel_l ].material_slots[ 0 ].material.game_settings.physics =
2284         use_material_physics;
2285     bpy.data.objects[ front_wheel_r ].material_slots[ 0 ].material.game_settings.physics =
2286         use_material_physics;
2287     bpy.data.objects[ back_wheel_l ].material_slots[ 0 ].material.game_settings.physics =
2288         use_material_physics;
2289     bpy.data.objects[ back_wheel_r ].material_slots[ 0 ].material.game_settings.physics =
2290         use_material_physics;
2291     ...
2292     # Friction.
2293     material_friction = get_clamped_value( genome_genes[ 5 ] * 100.0, 0.0, 100.0 );
2295     self.log( "Setting material friction." );
2297     self.log( str( material_friction ) );
2299     bpy.data.objects[ front_wheel_l ].material_slots[ 0 ].material.physics.friction = material_friction;
2300     bpy.data.objects[ front_wheel_r ].material_slots[ 0 ].material.physics.friction = material_friction;
2301     bpy.data.objects[ back_wheel_l ].material_slots[ 0 ].material.physics.friction = material_friction;
2302     bpy.data.objects[ back_wheel_r ].material_slots[ 0 ].material.physics.friction = material_friction;
2303     # Elasticity.
2304     material_elasticity = get_clamped_value( genome_genes[ 6 ], 0.0, 1.0 );
2305     self.log( "Setting material elasticity." );
2307     self.log( str( material_elasticity ) );
2309     bpy.data.objects[ front_wheel_l ].material_slots[ 0 ].material.physics.elasticity =
2310         material_elasticity;
2311     bpy.data.objects[ front_wheel_r ].material_slots[ 0 ].material.physics.elasticity =
2312         material_elasticity;
2313     bpy.data.objects[ back_wheel_l ].material_slots[ 0 ].material.physics.elasticity =
2314         material_elasticity;
2315     bpy.data.objects[ back_wheel_r ].material_slots[ 0 ].material.physics.elasticity =
2316         material_elasticity;
2317     ### PHYSICS
2319     # Type?
2321     ...
2322     PHYSICS_TYPES = [ "NO_COLLISION", "STATIC", "DYNAMIC", "RIGID_BODY", "SOFT_BODY", "OCCLUDE", "SENSOR",
2323         "NAVMESH", "CHARACTER" ];
2325     physics_type = get_clamped_value( math.floor( genome_genes[ 7 ] * len( PHYSICS_TYPES ) ), 0, len(
2326         PHYSICS_TYPES ) - 1 );
2327     physics_type = PHYSICS_TYPES[ physics_type ];
2328     bpy.data.objects[ front_wheel_l ].game.physics_type = physics_type;
2329     bpy.data.objects[ front_wheel_r ].game.physics_type = physics_type;
2330     bpy.data.objects[ back_wheel_l ].game.physics_type = physics_type;
2331     bpy.data.objects[ back_wheel_r ].game.physics_type = physics_type;
2332     ...
2335

```

```

# Ghost?
2337
2338
2339 index = int( str( round( genome_genes[ 8 ] ) ).split( "." )[ 0 ] );
2340
2341 use_ghost = BOOLEANS[ index ];
2342
2343 bpy.data.objects[ front_wheel_l ].game.use_ghost = use_ghost;
2344 bpy.data.objects[ front_wheel_r ].game.use_ghost = use_ghost;
2345 bpy.data.objects[ back_wheel_l ].game.use_ghost = use_ghost;
2346 bpy.data.objects[ back_wheel_r ].game.use_ghost = use_ghost;
2347
2348
2349 # Mass.
2350
2351
2352 #mass = get_clamped_value( ( genome_genes[ 9 ] * ( 10000.0 - 0.01 ) ) + 0.010, 0.010, 10000.0 );
2353 mass = get_clamped_value( ( genome_genes[ 9 ] * ( 15.0 - 0.010 ) ) + 0.010, 0.010, 15.0 );
2354
2355 self.log( "Setting mass." );
2356
2357 self.log( str( mass ) );
2358
2359 bpy.data.objects[ front_wheel_l ].game.mass = mass;
2360 bpy.data.objects[ front_wheel_r ].game.mass = mass;
2361 bpy.data.objects[ back_wheel_l ].game.mass = mass;
2362 bpy.data.objects[ back_wheel_r ].game.mass = mass;
2363
2364 # Form factor?
2365
2366
2367 form_factor = get_clamped_value( ( genome_genes[ 10 ] ), 0.0, 1.0 );
2368
2369 self.log( "Setting form factor." );
2370
2371 self.log( str( form_factor ) );
2372
2373 bpy.data.objects[ front_wheel_l ].game.form_factor = form_factor;
2374 bpy.data.objects[ front_wheel_r ].game.form_factor = form_factor;
2375 bpy.data.objects[ back_wheel_l ].game.form_factor = form_factor;
2376 bpy.data.objects[ back_wheel_r ].game.form_factor = form_factor;
2377
2378
2379 # Velocity maximum.
2380
2381 velocity_max = get_clamped_value( ( genome_genes[ 11 ] * 1000.0 ), 0.0, 1000.0 );
2382
2383 self.log( "Setting velocity max." );
2384
2385 self.log( str( velocity_max ) );
2386
2387 bpy.data.objects[ front_wheel_l ].game.velocity_max = velocity_max;
2388 bpy.data.objects[ front_wheel_r ].game.velocity_max = velocity_max;
2389 bpy.data.objects[ back_wheel_l ].game.velocity_max = velocity_max;
2390 bpy.data.objects[ back_wheel_r ].game.velocity_max = velocity_max;
2391
2392 # Damping translation.
2393
2394 damping = get_clamped_value( ( genome_genes[ 12 ] ), 0.0, 1.0 );
2395
2396 self.log( "Setting translation damping." );
2397
2398 self.log( str( damping ) );
2399
2400 bpy.data.objects[ front_wheel_l ].game.damping = damping;
2401 bpy.data.objects[ front_wheel_r ].game.damping = damping;
2402 bpy.data.objects[ back_wheel_l ].game.damping = damping;
2403 bpy.data.objects[ back_wheel_r ].game.damping = damping;
2404
2405 # Damping rotation.
2406
2407 rotation_damping = get_clamped_value( ( genome_genes[ 13 ] ), 0.0, 1.0 );
2408
2409 self.log( "Setting rotation damping." );
2410
2411 self.log( str( rotation_damping ) );
2412
2413 bpy.data.objects[ front_wheel_l ].game.rotation_damping = rotation_damping;
2414 bpy.data.objects[ front_wheel_r ].game.rotation_damping = rotation_damping;
2415 bpy.data.objects[ back_wheel_l ].game.rotation_damping = rotation_damping;
2416 bpy.data.objects[ back_wheel_r ].game.rotation_damping = rotation_damping;
2417
2418 # Use collision bounds?
2419
2420
2421 index = int( str( round( genome_genes[ 14 ] ) ).split( "." )[ 0 ] );
2422
2423 use_collision_bounds = BOOLEANS[ index ];
2424
2425 self.log( "Setting use collision bounds." );
2426
2427 self.log( str( use_collision_bounds ) );
2428
2429 bpy.data.objects[ front_wheel_l ].game.use_collision_bounds = use_collision_bounds;
2430 bpy.data.objects[ front_wheel_r ].game.use_collision_bounds = use_collision_bounds;
2431 bpy.data.objects[ back_wheel_l ].game.use_collision_bounds = use_collision_bounds;
2432 bpy.data.objects[ back_wheel_r ].game.use_collision_bounds = use_collision_bounds;
2433
2434
2435
2436
2437

```

```

2439     ...
2441     # Collision margin?
2443     ...
2445     collision_margin = get_clamped_value( ( genome_genes[ 15 ] ), 0.0, 1.0 );
2447     self.log( "Setting collision margin." );
2449     self.log( str( collision_margin ) );
2451     bpy.data.objects[ front_wheel_l ].game.collision_margin = collision_margin;
2452     bpy.data.objects[ front_wheel_r ].game.collision_margin = collision_margin;
2453     bpy.data.objects[ back_wheel_l ].game.collision_margin = collision_margin;
2454     bpy.data.objects[ back_wheel_r ].game.collision_margin = collision_margin;
2455     ...
2457     # Collision bound type.
2459     #COLLISION_BOUNDS_TYPES = [ "TRIANGLE_MESH", "CONVEX_HULL", "CONE", "CYLINDER", "SPHERE", "BOX", "CAPSULE" ];
2461     COLLISION_BOUNDS_TYPES = [ "TRIANGLE_MESH", "CONVEX_HULL", "CYLINDER", "SPHERE" ];
2463     collision_bounds.type = get_clamped_value( math.floor( genome_genes[ 16 ] * len(
2464         COLLISION_BOUNDS_TYPES ) ), 0, len( COLLISION_BOUNDS_TYPES ) - 1 );
2465     collision_bounds.type = COLLISION_BOUNDS_TYPES[ collision_bounds.type ];
2467     self.log( "Setting collision bounds type." );
2469     self.log( str( collision_bounds.type ) );
2471     bpy.data.objects[ front_wheel_l ].game.collision_bounds_type = collision_bounds.type;
2472     bpy.data.objects[ front_wheel_r ].game.collision_bounds_type = collision_bounds.type;
2473     bpy.data.objects[ back_wheel_l ].game.collision_bounds_type = collision_bounds.type;
2474     bpy.data.objects[ back_wheel_r ].game.collision_bounds_type = collision_bounds.type;
2475
2477     ### LOGIC BRICKS
2479     # Torque.
2481     #torque_z = get_clamped_value( ( -INF + ( genome_genes[ 17 ] * ( INF + INF ) ) ), -INF, INF );
2483     torque_z = get_clamped_value( genome_genes[ 17 ] * self.max_torque, 0.0, self.max_torque );
2485     self.log( "Setting torque_z." );
2487     self.log( str( torque_z ) );
2489     # In order to make the collision_bounds_type "CYLINDER" feasible, the wheel had to be rotated in model
2490     # space by -90deg around the x-axis. This allows the cylinder shape to coincide with the wheel shape.
2491     # Otherwise, if the wheel is unrotated, the cylinder bounds' flat sides reside at the rounded sides
2492     # of the wheel. Imagine standing a tire up, putting a tube over it and rolling the wheel.
2493     # Thus, in local space, applying torque to the wheels must be done around the z-axis since in local
2494     # space,
2495     # the z-axis points out of the wheel hub.
2496     bpy.data.objects[ front_wheel_l ].game.actuators[ actuator ].use_local_torque = True;
2497     bpy.data.objects[ front_wheel_r ].game.actuators[ actuator ].use_local_torque = True;
2498     bpy.data.objects[ back_wheel_l ].game.actuators[ actuator ].use_local_torque = True;
2499     bpy.data.objects[ back_wheel_r ].game.actuators[ actuator ].use_local_torque = True;
2501     bpy.data.objects[ front_wheel_l ].game.actuators[ actuator ].torque = [ 0, 0, torque_z ];
2502     bpy.data.objects[ front_wheel_r ].game.actuators[ actuator ].torque = [ 0, 0, torque_z ];
2503     bpy.data.objects[ back_wheel_l ].game.actuators[ actuator ].torque = [ 0, 0, torque_z ];
2504     bpy.data.objects[ back_wheel_r ].game.actuators[ actuator ].torque = [ 0, 0, torque_z ];
2505
2507     # Record the genome's phenotype (the physics parameters) and its eventual fitness.
2509     physics_parameters_with_fitness_file = open( get_scripts_location( ) + "data/
2510         physics_parameters_with_fitness/" + "ppwf_" + str( self.run_id ) + ".dat", "a" );
2511     physics_parameters_with_fitness_file.write( "gravity," + str( gravity ) + "\n" );
2512     physics_parameters_with_fitness_file.write( "sub_steps," + str( sub_steps ) + "\n" );
2513     physics_parameters_with_fitness_file.write( "fps," + str( fps ) + "\n" );
2514     #physics_parameters_with_fitness_file.write( "scale," + str( scale ) + "\n" );
2515     #physics_parameters_with_fitness_file.write( "use_material_physics," + str( use_material_physics ) +
2516     "\n" );
2517     physics_parameters_with_fitness_file.write( "material_friction," + str( material_friction ) + "\n" );
2518     physics_parameters_with_fitness_file.write( "material_elasticity," + str( material_elasticity ) + "\n"
2519     );
2520     #physics_parameters_with_fitness_file.write( "physics_type," + str( physics_type ) + "\n" );
2521     #physics_parameters_with_fitness_file.write( "use_ghost," + str( use_ghost ) + "\n" );
2522     physics_parameters_with_fitness_file.write( "mass," + str( mass ) + "\n" );
2523     #physics_parameters_with_fitness_file.write( "form_factor," + str( form_factor ) + "\n" );
2524     physics_parameters_with_fitness_file.write( "velocity_max," + str( velocity_max ) + "\n" );
2525     physics_parameters_with_fitness_file.write( "damping," + str( damping ) + "\n" );
2526     physics_parameters_with_fitness_file.write( "rotation_damping," + str( rotation_damping ) + "\n" );
2527     #physics_parameters_with_fitness_file.write( "use_collision_bounds," + str( use_collision_bounds ) +
2528     "\n" );
2529     physics_parameters_with_fitness_file.write( "torque_z," + str( torque_z ) + "\n" );
2530     physics_parameters_with_fitness_file.close( );
2531 class Database_Manager( ):

```

```

2533     def __init__( self, database_name ):
2534         self.database_name = database_name;
2537         variables_location = get_scripts_location( ) + "variables/";
2539         database_file = open( variables_location + "database.var", "r" );
2541         self.user_name = database_file.readline( ).rstrip( );
2543         self.password = database_file.readline( ).rstrip( );
2545         self.connection = None;
2547         self.cursor = None;
2549     def connect_to_database( self ):
2551         connection_config = {
2553             "user": self.user_name,
2554             "password": self.password,
2555             "database": self.database_name,
2556             'raise_on_warnings': True
2557         };
2559     try:
2561         self.connection = mysql.connector.connect( **connection_config );
2563         self.cursor = self.connection.cursor();
2565     except Exception:
2567         print( "Database_Manager: could not open a connection to the database." );
2569     def close_database_connection( self ):
2571     try:
2573         self.connection.close();
2575     except Exception:
2577         print( "Database_Manager: could not close the database." );
2579     def execute( self, mysql_string = None ):
2581         if mysql_string == None:
2582             pass;
2585         elif self.cursor == None:
2587             print( "Database_Manager: you must open a connection the database first." );
2589         else:
2591             self.cursor.execute( mysql_string );
2593             self.connection.commit();
2595         # Seed the random module.
2597         random.seed();
2599     # Create the BBAutoTune object.
2601     bbautotune = BBAutoTune( Genetic_Algorithm( ), Database_Manager( "bbautotune" ) );
2603     # Add the bbautotune object instance to Blender.
2605     bpy.bbautotune = bbautotune;
2607     # Switch to the game engine in Blender.
2609     bpy.context.scene.render.engine = "BLENDER_GAME";
2611     # Switch to the render pane.
2613     bpy.data.screens[ "Default" ].areas[ 1 ].spaces[ 0 ].context = "RENDER";
2615     # Setup the robot-monitor object's python controller.
2617     position_recorder_script_text = bpy.data.texts.load( get_scripts_location( ) + "robot_monitor.py" );
2619     bpy.data.objects[ "robot_monitor" ].game.controllers[ 0 ].text = position_recorder_script_text;
2621     # Register the UI panel properties, the UI panel layout, and the start button operator with blender.
2623     bpy.utils.register_module( __name__ );

```

A.3 BlenderSim

LISTING A.5: Robot_1_Controller.py

```

    ...
2 David Lettier (C) 2014.
4 http://www.lettier.com/
6 BlenderSim Version 2.0
8 This file controls the robot's movements.
10 ,,
12 # Imports.
14 import mathutils, math, time, copy, datetime, os, pickle, random;
16 def robot_1_moving( ):
18     robot_1_base = bge.logic.getCurrentScene( ).objects[ "robot_1_base" ];
20     test = {
22         "xpos": robot_1_base.worldPosition.x * 100.0,
24         "ypos": robot_1_base.worldPosition.y * 100.0,
26         "zpos": robot_1_base.worldPosition.z * 100.0,
28         "xrot": robot_1_base.worldOrientation.to_euler( ).x,
        "yrot": robot_1_base.worldOrientation.to_euler( ).y,
        "zrot": robot_1_base.worldOrientation.to_euler( ).z
30     };
32     stopped = len( set( test.items( ) ) ^ set( bge.logic.globalDict[ "robot_1_state" ].items( ) ) );
34     if ( stopped == 0 ):
36         return False;
38     else:
40         x1 = bge.logic.globalDict[ "robot_1_simulated_poses" ][ -2 ][ 1 ];
42         y1 = bge.logic.globalDict[ "robot_1_simulated_poses" ][ -2 ][ 2 ];
42         z1 = bge.logic.globalDict[ "robot_1_simulated_poses" ][ -2 ][ 3 ];
44         x2 = test[ "xpos" ];
45         y2 = test[ "ypos" ];
46         z2 = test[ "zrot" ];
48         dx = abs( x1 - x2 );
49         dy = abs( y1 - y2 );
50         dz = abs( z1 - z2 );
52     if ( dx <= 0.02 and dy <= 0.02 and dz <= 0.00087 ):
54         return False;
55     else:
56         return True;
58 def update_robot_1_state( ):
60     robot_1_base = bge.logic.getCurrentScene( ).objects[ "robot_1_base" ];
62     bge.logic.globalDict[ "robot_1_state" ] = {
64         "xpos": robot_1_base.worldPosition.x * 100.0,
66         "ypos": robot_1_base.worldPosition.y * 100.0,
68         "zpos": robot_1_base.worldPosition.z * 100.0,
70         "xrot": robot_1_base.worldOrientation.to_euler( ).x,
        "yrot": robot_1_base.worldOrientation.to_euler( ).y,
        "zrot": robot_1_base.worldOrientation.to_euler( ).z
72     };
74 def stop_robot_1( ):
76     bge.logic.robot_1_wheel_front_L.applyTorque( [ 0.0, 0.0, 0.0 ], True );
76     bge.logic.robot_1_wheel_front_R.applyTorque( [ 0.0, 0.0, 0.0 ], True );
78     bge.logic.robot_1_wheel_back_L.applyTorque( [ 0.0, 0.0, 0.0 ], True );
78     bge.logic.robot_1_wheel_back_R.applyTorque( [ 0.0, 0.0, 0.0 ], True );
80 def drop_robot_1_campose_marker( ):
82     robot_1_campose_marker = bge.logic.getCurrentScene( ).addObject( "robot_1_campose_marker", obj );
84     robot_1_campose_marker.worldPosition = obj.worldPosition;
84     robot_1_campose_marker.worldPosition.z = 23.0 / 100.0;
86 def drop_robot_1_waypoint_marker( waypoint ):
88     robot_1_waypoint_marker = bge.logic.getCurrentScene( ).addObject( "waypoint_marker", obj );
90     robot_1_waypoint_marker.worldPosition.x = waypoint[ 1 ] / 100.0;
90     robot_1_waypoint_marker.worldPosition.y = waypoint[ 0 ] / 100.0;
92     robot_1_waypoint_marker.worldPosition.z = 25.0 / 100.0;
94 def add_sim_pose( ):
96     bge.logic.globalDict[ "robot_1_simulated_poses" ].append(
98         [
100             time.time( ) * 1000.0,
100             obj.worldPosition[ 0 ] * 100.0,
100             obj.worldPosition[ 1 ] * 100.0,
100             obj.worldOrientation.to_euler( ).z

```

```

104     ];
106   );
108 # Globals.
110 local = True;
112 world = False;
114 # Get the controller.
116 controller = bge.logic.getCurrentController( );
118 # Get the game object that the controller is attached to.
120 obj = controller.owner;
122 # Initialize variables and flags.
124 if ( obj[ "init" ] == True ):
126   obj[ "init" ] = False;
128 # Robot 1's wheels.
130 bge.logic.robot_1_wheel_front_L = bge.logic.getCurrentScene( ).objects[ "robot_1_wheel_front_L" ];
131 bge.logic.robot_1_wheel_front_R = bge.logic.getCurrentScene( ).objects[ "robot_1_wheel_front_R" ];
132 bge.logic.robot_1_wheel_back_L = bge.logic.getCurrentScene( ).objects[ "robot_1_wheel_back_L" ];
133 bge.logic.robot_1_wheel_back_R = bge.logic.getCurrentScene( ).objects[ "robot_1_wheel_back_R" ];
134
135 bge.logic.robot_1_last_move_time = time.time( ) * 1000.0;
136 update_robot_1_state( );
138 bge.logic.globalDict[ "robot_1_waypoints" ] = [
140   "stop"
142 ];
144 bge.logic.globalDict[ "robot_1_simulated_poses" ] = [ ];
146 add_sim_pose( );
148 add_sim_pose( );
150 if ( ( bge.logic.globalDict[ "robot_1_waypoints" ][ 0 ] != "stop" ) and ( not robot_1_moving( ) ) and ( (
151   time.time( ) * 1000.0 ) - bge.logic.robot_1_last_move_time >= 0.0 ) ):
152 # Get and drop the next waypoint in the arena.
154 waypoint = bge.logic.globalDict[ "robot_1_waypoints" ][ 0 ];
156 drop_robot_1_waypoint_marker( waypoint );
158 # Calculate the angle to turn in order to face the waypoint.
160 # Convert the waypoint to a 3D vector.
162 waypoint = mathutils.Vector( ( waypoint[ 1 ], waypoint[ 0 ], 1.0 ) );
164 # Translate the waypoint to the robot's local space.
166 mat_trans = mathutils.Matrix.Translation( ( -obj.worldPosition[ 0 ] * 100.0, -obj.worldPosition[ 1 ] *
167   100.0, -obj.worldPosition[ 2 ] * 100.0 ) );
168 waypoint_trans = mat_trans * waypoint;
170 waypoint_rot = mathutils.Matrix.Rotation( -obj.worldOrientation.to_euler( )[ 2 ], 4, "Z" ) *
171   waypoint_trans;
172 # Now that the waypoint world coordinate is transformed to the robot's local space,
173 # compute the angle between the robot's x-axis and the waypoint line going from the
174 # robot's origin to the waypoint.
176 rotateZ = math.atan2( waypoint_rot[ 1 ], waypoint_rot[ 0 ] );
178 # First turn and then move forward.
180 rotateZ = rotateZ * 180.0 / math.pi;
182 if ( abs( rotateZ ) < 1.0 ): # Turn error threshold.
184   a = 82.7271515601;
185   b = 23.12349975;
186   c = waypoint_rot[ 0 ];
187   f = ( a / b ) * c;
188
189   bge.logic.robot_1_wheel_front_L.applyTorque( [ 0.0, 0.0, f ], True );
190   bge.logic.robot_1_wheel_front_R.applyTorque( [ 0.0, 0.0, f ], True );
191   bge.logic.robot_1_wheel_back_L.applyTorque( [ 0.0, 0.0, f ], True );
192   bge.logic.robot_1_wheel_back_R.applyTorque( [ 0.0, 0.0, f ], True );
194
195   bge.logic.globalDict[ "robot_1_waypoints" ] = bge.logic.globalDict[ "robot_1_waypoints" ][ 1 : ];
196 elif ( rotateZ < 0.0 ):
198   a = 20.8;
199   b = 44.260811;
200   c = abs( rotateZ );
201   t = ( a / b ) * c;

```

```

202     bge.logic.robot_1_wheel_front_L.applyTorque( [ 0.0, 0.0, t ], True );
204     bge.logic.robot_1_wheel_front_R.applyTorque( [ 0.0, 0.0, -t ], True );
206     bge.logic.robot_1_wheel_back_L.applyTorque( [ 0.0, 0.0, t ], True );
208     bge.logic.robot_1_wheel_back_R.applyTorque( [ 0.0, 0.0, -t ], True );
210
211     elif ( rotateZ > 0.0 ):
212         a = 20.8;
213         b = 44.260811;
214         c = abs( rotateZ );
215         t = ( a / b ) * c;
216
217         bge.logic.robot_1_wheel_front_L.applyTorque( [ 0.0, 0.0, -t ], True );
218         bge.logic.robot_1_wheel_front_R.applyTorque( [ 0.0, 0.0, t ], True );
219         bge.logic.robot_1_wheel_back_L.applyTorque( [ 0.0, 0.0, -t ], True );
220         bge.logic.robot_1_wheel_back_R.applyTorque( [ 0.0, 0.0, t ], True );
221
222     bge.logic.robot_1_last_move_time = time.time() * 1000.0;
223
224     update_robot_1();
225
226     add_sim_pose();
227
228     drop_robot_1_compose_marker();
229
230     if ( not robot_1_moving() ):
231
232         pickle_file = open( "./pickled_data/robot_1_simulated_motion.pkl", "wb" );
233
234         pickle.dump( bge.logic.globalDict[ "robot_1_simulated_poses" ], pickle_file, protocol = 2, fix_imports =
235                     True );
236
237         pickle_file.close();

```

Bibliography

- [1] Jhih Ren. Juang, Wei Han Hung, and Shih Chung Kang. Using game engines for physicalbased simulationsa forklift. *Journal of Information Technology in Construction*, 16:3–22, 2011.
- [web] Doc:2.6/manual/game engine/physics/object type/dynamic - blenderwiki. http://wiki.blender.org/index.php/Doc:2.6/Manual/Game_Engine/Physics/Object_Type/Dynamic, . Accessed August 13, 2013.
- [web] Doc:2.6/manual/game engine/physics/object type - blenderwiki. http://wiki.blender.org/index.php/Doc:2.6/Manual/Game_Engine/Physics/Object_Type, . Accessed August 13, 2013.
- [2] Roland Hess. *The Essential Blender: Guide to 3D Creation with the Open Source Suite Blender*. No Starch Press, San Francisco, CA, USA, 2007. ISBN 1593271662, 9781593271664.
- [3] Colin Reeves. Genetic algorithms. In Fred Glover and GaryA. Kochenberger, editors, *Handbook of Metaheuristics*, volume 57 of *International Series in Operations Research & Management Science*, pages 55–82. Springer US, 2003. ISBN 978-1-4020-7263-5. doi: 10.1007/0-306-48056-5_3. URL http://dx.doi.org/10.1007/0-306-48056-5_3.
- [4] David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 1, fundamentals, 1993.
- [5] Pedro A. Diaz-Gomez and Dean F. Hougen. Initial population for genetic algorithms: A metric approach. In Hamid R. Arabnia, Jack Y. Yang, and Mary Qu Yang, editors, *GEM*, pages 43–49. CSREA Press, 2007. ISBN 1-60132-038-8.
- [6] Marek Obitko. Introduction to genetic algorithms: Encoding. <http://www.obitko.com/>

- tutorials/genetic-algorithms/encoding.php, 1998. URL <http://www.obitko.com/tutorials/genetic-algorithms/encoding.php>.
- [7] Daniel W. Dyer. Practical evolutionary computation: Elitism. <http://blog.uncommons.org/2009/02/12/practical-evolutionary-computation-elitism/>, February 2009. Accessed Apr 6, 2014.
- [8] Mat Buckland. Neural networks in plain english. <http://www.ai-junkie.com/ann/evolved/nnt1.html>, September 2013. URL <http://www.ai-junkie.com/ann/evolved/nnt1.html>.
- [9] Hartmut Pohlheim. Genetic and evolutionary algorithms: Principles, methods and algorithms. <http://www.pg.gda.pl/~mkwies/dyd/geadocu/algindex.html>, May 1997. URL <http://www.pg.gda.pl/~mkwies/dyd/geadocu/algindex.html>.
- [10] Wen-Yung Lee Wen-Yang Lin and Tzung-Pei Hong. Adapting crossover and mutation rates in genetic algorithms. *Journal of Information Science and Engineering*, 19:889–903, 2003.
- [11] Marek Obitko. Introduction to genetic algorithms: Selection. <http://www.obitko.com/tutorials/genetic-algorithms/selection.php>, 1998. URL <http://www.obitko.com/tutorials/genetic-algorithms/selection.php>.
- [12] Heinz Mühlenbein and Dirk Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm i. continuous parameter optimization. *Evol. Comput.*, 1(1):25–49, March 1993. ISSN 1063-6560. doi: 10.1162/evco.1993.1.1.25. URL <http://dx.doi.org/10.1162/evco.1993.1.1.25>.
- [web] Bullet collision detection & physics library. <http://www.continuousphysics.com/Bullet/BulletFull/index.html>, . Accessed March 26, 2014.
- [13] Brad L. Miller, Brad L. Miller, David E. Goldberg, and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.
- [14] R. De Maesschalck, D. Jouan-Rimbaud, and D.L. Massart. The mahalanobis distance. *Chemometrics and Intelligent Laboratory System*, 50(1):1–18, January 2000.
- [15] Rick Wicklin. What is mahalanobis distance? <http://blogs.sas.com/content/iml/2012/02/15/what-is-mahalanobis-distance/>, February 2012. Accessed March 27, 2014.

- [16] Peter J. Rousseeuw and Katrien Van Driesssen. A fast algorithm for the minimum covariance determinant estimator. *Technometrics*, 41(3):212–223, August 1999. ISSN 0040-1706. doi: 10.2307/1270566. URL <http://dx.doi.org/10.2307/1270566>.
- [17] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12: 2825–2830, 2011.
- [18] Mia Hubert and Michiel Debruyne. Minimum covariance determinant. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(1):36–43, 2010. ISSN 1939-0068. doi: 10.1002/wics.61. URL <http://dx.doi.org/10.1002/wics.61>.
- [19] Pankaj K. Agarwal, Rinat Ben Avraham, Haim Kaplan, and Micha Sharir. Computing the discrete fréchet distance in subquadratic time. *CoRR*, abs/1204.5333, 2012.
- [20] Xiao-Diao Chen, Weiyin Ma, Gang Xu, and Jean-Claude Paul. Computing the hausdorff distance between two b-spline curves. *Computer-Aided Design*, 42(12):1197–1206, 2010.
- [21] Elizabeth Sklar, Simon Parsons, Arif Tuna Ozgelen, Eric Schneider, Michael Costantino, and Susan L. Epstein. HRTeam: a framework to support research on human/multi-robot interaction. In Maria L. Gini, Onn Shehory, Takayuki Ito, and Catholijn M. Jonker, editors, *AAMAS*, pages 1409–1410. IFAAMAS, 2013. ISBN 978-1-4503-1993-5.
- [22] Scaling the world - physics simulation wiki. http://www.bulletphysics.org/mediawiki-1.5.8/index.php?title=Scaling_The_World, 2012. Accessed August 13, 2013.
- [23] Stage - the player project. <http://playerstage.sourceforge.net/wiki/Stage>, 2011. Accessed August 14, 2013.