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 14, 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. However, physics engines have numerous parameters to tune such as friction coefficients, rigid-body constraints, body mass, gravitational acceleration, material elasticity, and collision shapes. Tuning these either programmatically or manually is a daunting task considering the multi-dimensional search space. Simulation fidelity is compromised as some parameters are perfected while others are now maladjusted. This produces a "reality gap" between the simulated environment and the physical environment. However, by using a genetic algorithm to fine tune the physics-engine parameters, the hypothesis is that the physics engine will produce a robot motion model more closely reassembling that of the real-world robot kinematics. Approaching this hypothesis involved three incremental stages. The first stage involved the development of a genetic algorithm that tuned the weights of a neural network where the neural network controlled a paddle in a "Pong"-like game. The second stage involved tuning the parameters of a 3D physics engine using the genetic algorithm developed in stage one and motion data collected on a real robot. Lastly, stage three tested the hypothesis of the thesis in a parameter-tuned, 3D physics-based environment where the motion of a simulated robot was compared to the motion of its real-life counterpart.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The motivation for thesis was inspired by the creation of a 3D simulator titled BlenderSim. Using Blender[1], a 3D simulator was constructed for a human and robot multi-agent framework known as HRTeam [2]. By using BlenderSim, HRTeam experiments could be conducted in a real-time 3D physics-based virtual environment. The goal of BlenderSim was to provide a robust and high-fidelity simulation such that experimental results produced in simulation would be reproducible in reality.

The physics engine employed by Blender, called Bullet[3], proved to be troublesome while developing BlenderSim. The simulated robots would "jitter" erratically, fall through the floor, fly through the air, and/or fall apart even with no applied forces. Many days, turning into weeks, were lost trying to find the correct physics parameters by hand such that the simulation would produce high-fidelity results. In order to progress on the development of the BlenderSim 3D simulation environment, the idea of using the physics engine to model the motion of the robots was abandoned and a constant velocity motion model was used in its place.

[1]Blender, 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 [1].

[2]The purpose of HRTeam is to explore human and robot teamwork interactions [2][3].

[3]Bullet 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].

In the interim, the thesis problem was to tune the Blender physics engine via a genetic algorithm (GA) such that the physics engine would more closely model the motion of a real robot used in HRTeam experiments. This problem was approached in three incremental stages where each subsequent stage built off of the research and development done during the previous stage. Stage one involved the development a GA that tuned the weights of a neural network (NN) where the NN governed the movement of a paddle in a "Pong"-like game titled SimPL. Using the GA developed in stage one, stage two tuned the Blender physics engine as it iteratively compared the motion of a simulated robot to motion data collected on a real robot. Lastly, with the physics engine tuned in stage two, stage three compared the motion of a simulated robot versus a real robot by simulating a previously conducted HRTeam experiment.

1.1 Genetic Algorithms

Inspired by the theory of evolution, GAs attempt to solve problems that involve multidimensional state spaces [4][5]. GAs overcome problems experienced by other algorithms, such as hillclimbing, random search, or simulated annealing, by using a mixture of exploitation and exploration [5]. 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, and civil engineering [5]. What problems are appropriate for GAs and why GAs work when they do are highly debated topics with no dominant answers [4]. 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!" [4].

The term *genetic algorithm* has both a strict and broad definition [6]. The strict definition of a genetic algorithm is the model formally presented by John Holland in 1975 [4]. More generally, the broad definition is, "any population-based model that uses selection and recombination operators to generate new sample points in a search space"[6]. *Genetic algorithm*, as it is used in this text, should be interpreted with the broad definition. Additional terms used require their own distinctions. *Gene* refers to a single encoded value, representative of a dimension inherent to some multidimensional problem. *Genes* are a collection of

encoded values that represent a state in some multivariate state space. A *Genome* specifies a data structure that contains genes and other types of data such as the genome's fitness. The *population* of a GA is made up of genomes. The terms *genotype* and *phenotype* are interconnected. A genotype is represented by the genes of a genome. By expressing or rather decoding a genotype, a phenotype is constructed. Specifically for SimPL, the genotype was an encoding of the NN's weights and the phenotype was the paddle's movement, as dictated by the NN. For the software developed in stage two (known as BBAutoTune), the genotype was an encoding of the physics engine's parameters and the phenotype was the simulated robot's collision bounds and motion, as modeled by the physics engine.

1.2 SimPL

Instead of diving right into the difficult problem targeted by the thesis, SimPL was an intermediate step to developing a GA capable of tuning a 3D physics engine. The problem for SimPL involved a 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 GAs. With the knowledge gained by solving the problem for SimPL, the thesis project was set up 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 GA 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. 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

GAs have been applied to various problem domains since the works of John Holland, Hans-Paul Schwefel, Ingo Rechenberg, Lawrence Fogel, and John Koza [4][7]. 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. A GA may never find this global maxima or minima but may converge to some local optima depending on the GA's implementation and the fitness landscape. Coinciding with the exploration of the fitness landscape 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 [8]. 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 [4].

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 an optimum solution to. Various data structures ranging from strings to classes can be used to construct a genome [4]. 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 genome as a whole represents 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 structure (hardware and/or software) and behavior they produce is known as the phenotype. A genotype is evaluated by its phenotype's performance [5]. 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 such as a function or s-expression in the case of genetic programming [9]. Furthermore, not all the genes have to be of the same type but collectively could be some mixed tuple of various data types [4].

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 include:

1. binary—where every gene is a bit or a string of bits;
2. real-coded—where the genes could be real numbers, integers, or character strings;
3. tree—where each gene is a node in a tree that when parsed, forms some kind of expression; and
4. permutation—where the genes are some possible order of a sequence [10].

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 [4][11]. For problem domains with variables in $\mathbb{R}$, precision can be an issue when choosing a binary encoding. Another issue arises during the binary encoding of a finite set of discrete values when the set cardinality is not a power of two [11]. For example, image a variable that has five possible discrete values. To represent the five possible values, a three bit gene would be required. However, the gene configurations 101, 110, and 111 would be redundant or invalid and would need to be handled in the reproduction operators [6].

In [6], Holland's GA model is described as *the conical genetic algorithm.* Binary encoded/coded GAs (BCGA) that use recombination and selection follow Holland's model. A BCGA's search space is $\{0,1\}^l$. For one-bit genes, each gene represents the act of turning a variable on or off for some multivariate state space. Another interpretation of one-bit genes is that the bit represents the presence or absence of a phenotype trait—an analogy being a phenotype having or not having eyes. However, each gene can be made up of more than bit. In this case, each gene represents the magnitude of some variable or a set of discrete values a variable can be.

Parallel to Holland's work, Hans-Paul Schwefel and Ingo Rechenberg developed the concept of *evolution strategy* (ES) in the 1960's [4]. Originally, ES only used mutation and its population size was one [7]. However, other recombination operators and population sizes were later considered as ES researched progressed [11]. ES employs a real-coded scheme where each population member (genome) contains two vectors of floating point values. The first vector of floats is a point in the search space and the second vector of floats

is a vector of standard deviations for use in the mutation operator [11]. For some population member, let $\vec{\kappa} = \langle \kappa_1, \kappa_2, ..., \kappa_n \rangle$ be the vector of floats representing a point in the search space and let $\vec{\sigma} = \langle \sigma_1, \sigma_2, ..., \sigma_n \rangle$ be the corresponding vector of standard deviations. Here, a mutant offspring's $\vec{\kappa}' = \vec{\kappa} + N(0, \vec{\sigma})$ and its $\vec{\sigma}' = \vec{\sigma}$ after mutation where $N(0, \vec{\sigma})$ is a vector of samples from a normal distribution with a mean of zero and a standard deviation $\sigma_i \in \vec{\sigma}$ [7]. This mutant offspring only becomes a population member (by replacing its parent) if it obtains a higher fitness than its parent [7]. One could think of this strategy as being a probing of the fitness landscape, where population members never move from their locations until their random probes discover more optimal locales.

Another paradigm of evolutionary computing, differentiated mainly by its encoding scheme, is genetic programming (GP). GP was developed by John Koza [7]. In 1990, Koza presented the concept in his paper, *Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems.* Instead of searching for a solution to a problem directly, GP uses evolutionary tactics to search for a best-fit computer program capable of solving the problem at hand on its own. For example, in [9], Koza uses GP to evolve a computer program that dictates the behavior of an autonomous robot where the evolutionary goal was to get the robot to push a box from the center of a room to a wall. The main GP operator is crossover but mutation is used as well [7]. Each population member is a rooted tree, that when parsed, represents a hierarchical computer program. "Internal points of the tree correspond to functions (i.e. operations) and external points correspond to terminals (i.e. input data)" [9].

2.3 Evaluation

Evaluation of each phenotype yields the fitness of the corresponding genotype. 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 [5]. If at all possible, the fitness function should have a gradual gradient and a minimal number of local optima [5].

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 [5]. Some implementations incorporate constraint adherence within the reproduction operators, so that it is not possible to produce an invalid phenotype.

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 [4]. 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 [5]. 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 [5]. 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 [5]. Elitism attempts to preserve 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 [5].

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 [5].

2.4.2 Elitism

Elitism is one of the more simple operators. After the entire population has been evaluated, the top e performers are directly copied into the new population during the reproduction cycle. In addition to being directly copied into the next generation, the e 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 e—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 [12]. Note that e does not necessarily remain constant but could be varied per generation according to the number (or percentage) of population members that exceed a certain fitness threshold.

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 [5]. 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 [4]. 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 [5]. 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 on 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 optima and also help to keep the population diverse [4]. 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 [4]. 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 [8][4]. However, if one does seed the initial population, this may lead to premature convergence [4]. 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 straightforward answers [4].

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 [5]. 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 co-exist with offspring during any given generation [5]. 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 [4]. Human beings follow a steady-state approach with some dying and some coexisting with their children, after having reproduced.

2.5.3 Convergence

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 [5]. A converged population will be minimally diverse at the gene, genotype, and phenotype level [4].

2.6 Termination

GAs are inherently a stochastic process and will run forever unless constructed otherwise [4]. 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 [4]. Additional methods of when to terminate include:

1. the highest or average fitness reaching some predefined metric;
2. the change in fitness from one generation to another falling below some threshold; and
3. the GA reaching some generation number.

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 the screen and the ball originating from the far right of the screen. See Figure 3.1.

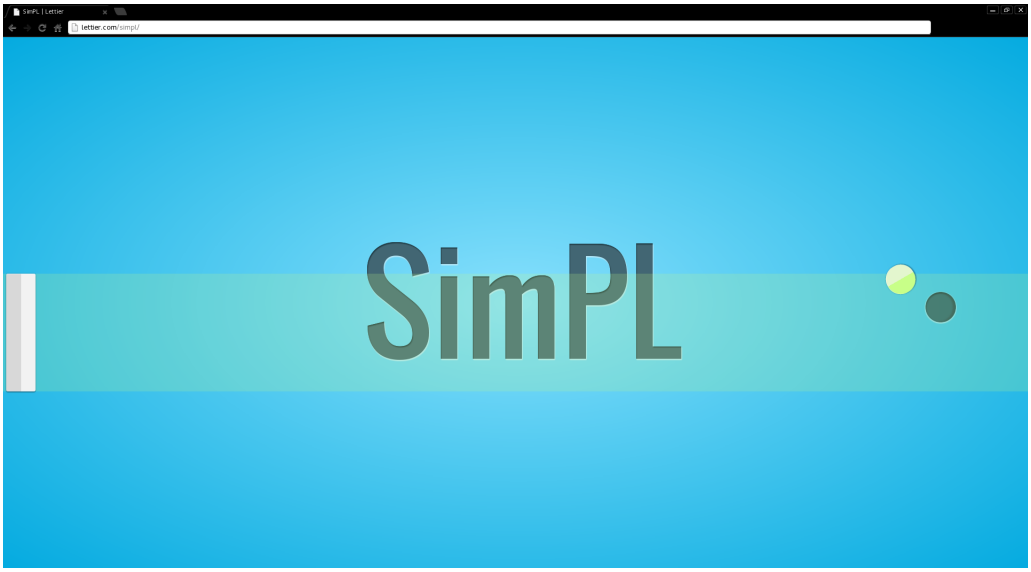

FIGURE 3.1: The SimPL arena containing the paddle and ball.

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 round[1]. Just before the beginning of a round, a random angle in the range $[135°, 225°]$ 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. Note that the ball's velocity magnitude is set to 1000 $\frac{pixels}{second}$ at the start of every round.

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

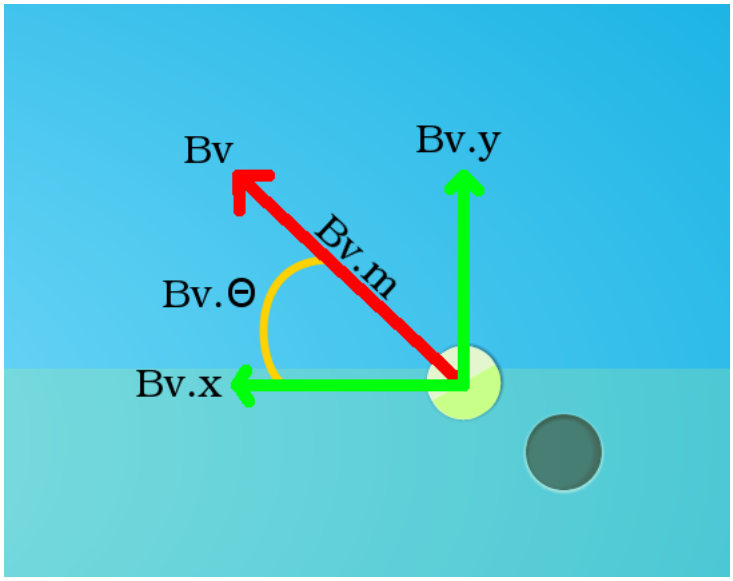

FIGURE 3.2: The ball's dynamic physics properties.

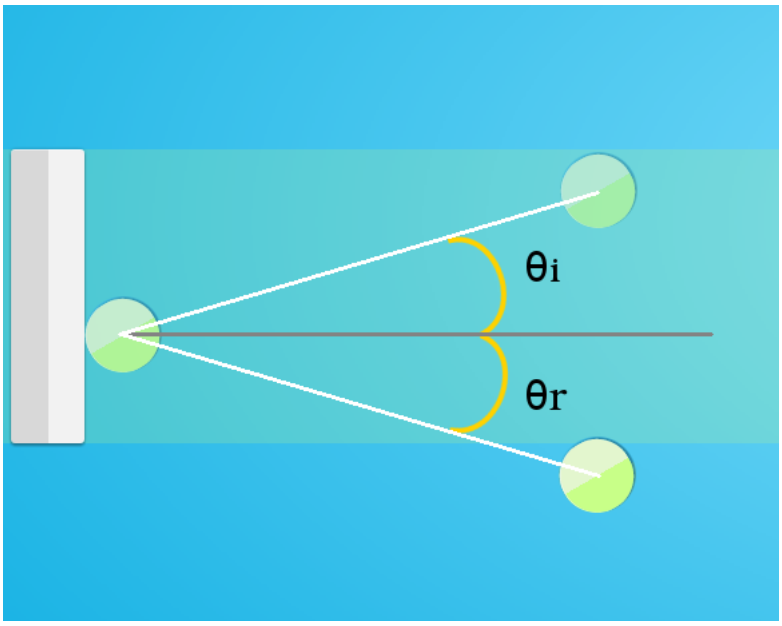

FIGURE 3.3: The ball's collision angle-of-incidence θ_i and its angle-of-reflection θ_r.

3.2.3 Paddle

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

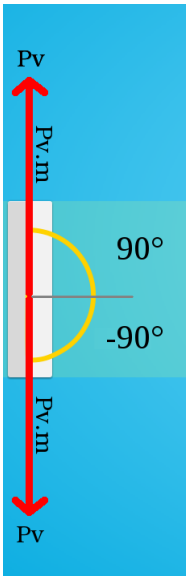

FIGURE 3.4: The paddle's dynamic physics properties.

The paddle's direction and speed are regulated by the output of a neural network. The output of the neural network is in the range $[-1, 1]$. The sign of the output determines the up or down direction of the paddle and the absolute value of the output determines the paddle's speed. 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 1000 $\frac{pixels}{second}$. A neural network output in the range of $[-1, 0)$ results in the paddle traveling down by some percentage of 1000 $\frac{pixels}{second}$. 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 $270°$ (since the sign of the neural network output was negative) and its velocity magnitude would be set to $m = |-0.68| * 1000 \frac{pixels}{second}$. Here, the paddle can always travel as fast or faster than the ball, given that the ball's velocity magnitude at the start of a round is 1000 $\frac{pixels}{second}$, the ball's velocity magnitude

decreases during a round, and the paddle's velocity magnitude is always some percentage of 1000 $\frac{pixels}{second}$ throughout the duration of a round depending on the NN output. It is never the case that the paddle does not have the possibility to reach the ball in time during the duration of any round. The paddle can always reach the ball given the neural network output is correct. Thus, at any time t during any round R_i, the paddle's velocity magnitude is $0 \leq m \leq 1000$ $\frac{pixels}{second}$ since $m = |[-1, 1]| * 1000$ $\frac{pixels}{second}$ where $|[-1, 1]|$ is the absolute value of the neural network's output range. Furthermore, at any time t during any round R_i, the paddle's velocity angle (direction) is either 90° or 270° depending on the sign of the neural network output.

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

Handles to all dynamic and static objects are passed to 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 as described above. 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 [13]. 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 $\left(tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}\right)$ resulting in an output range of $[-1, 1]$.

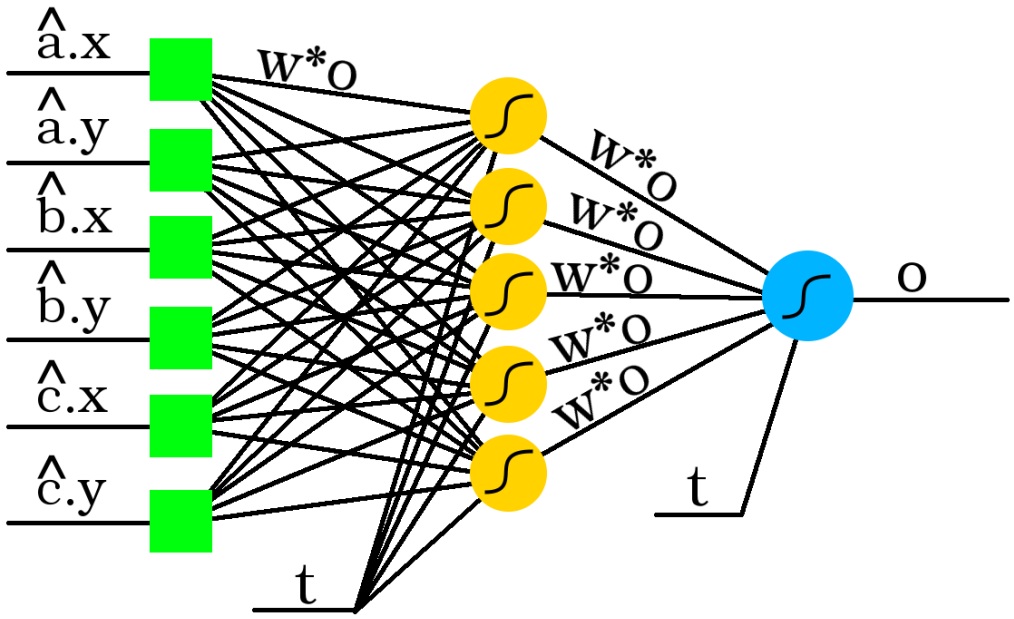

FIGURE 3.5: The neural network as constructed in SimPL.

Input to the neural network is normalized as 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.

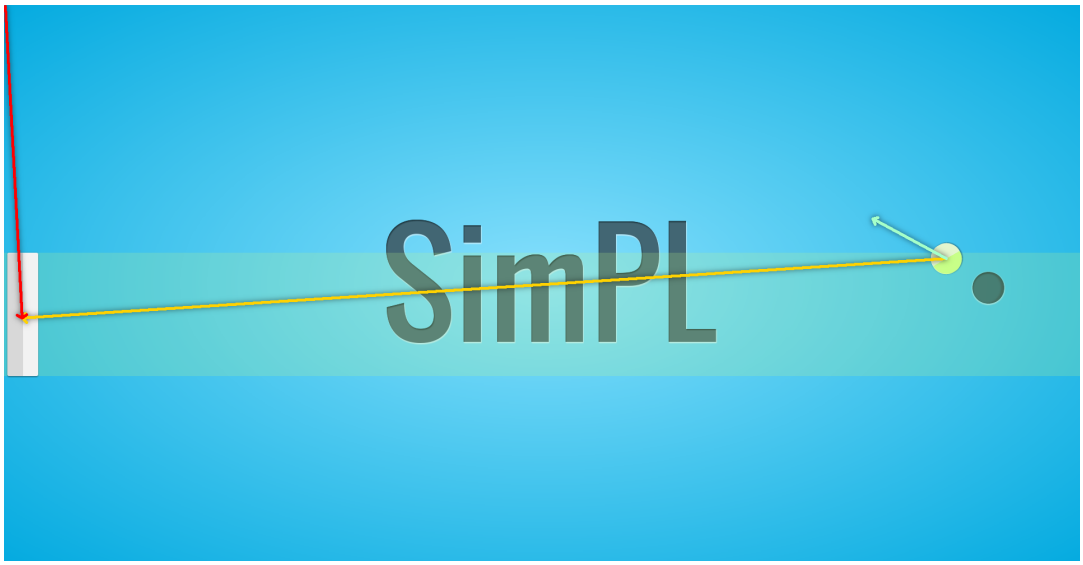

FIGURE 3.6: The three input vectors to the neural network. Note that these vectors are normalized thereby turning them into unit vectors.

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 [13]. 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

FIGURE 3.7: The normalized input to the neural network.

SimPL, the mechanism is the neural network. Each gene 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' ≤ population size do
            If perform crossover and mutation together 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: 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 [14]. 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 ($crossoverPoint = X \sim U(0, n-1)$ letting n denote 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 together (applying them both to one set of selected parents) 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. Rather, crossover and mutation occur based on some probability. If the crossover and mutation probabilities are both set to 1.0, then they always occur. Before the genetic algorithm is initialized, the crossover and mutation probabilities are set [15]. 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 each 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 described below) 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. 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*
Population Size	10	10	10	10	10	10
Fitness Function Type	Partial/Full	Full	Full	Full	Full	Full
Number of Elite Offspring	2	2	2	2	2	10
Roulette Selection - Actual Fitness	True	False	False	False	False	N/A
Roulette Selection - Rank Fitness	False	True	True	True	True	N/A
Sequential Crossover & Mutation	True	True	False	False	True	N/A
Self-adaptation	False	False	True	False	False	N/A
Crossover Type	One-point	One-point	One-point	One-point	One-point	N/A
Crossover Probability	0.7	0.8	0.5	0.7816	0.7816	N/A
Mutation Type	Uniform	Gaussian	Gaussian	Gaussian	Gaussian	N/A
Mutation Scope	Gene	Gene	Genome	Genome	Gene	N/A
Mutation Probability	0.1	$\left(\frac{1}{n}\right) = \left(\frac{1}{41}\right) \approx 0.0244$	0.5	0.2184	0.2184	N/A
Mutation Step	$X \sim U(0,1) * 0.3$	$X \sim N(\mu, \sigma^2)$	$X \sim N(\mu, \sigma^2)$	$X \sim N(\mu, \sigma^2)$	$X \sim (\mu, \sigma^2)$	N/A
Mutation Step μ	N/A	Gene Value	Gene Value	Gene Value	Gene Value	N/A
Mutation Step σ	N/A	0.5	$M_{Prob.}$	$M_{Prob.}$	$M_{Prob.}$	N/A

TABLE 3.1: 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.

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: The genetic algorithm parameters for experiment one.

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 (see below) 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.

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	$\frac{1}{n\ genes} = \frac{1}{41} = 0.024390244$
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: The genetic algorithm parameters for experiment two.

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 behavior in relation to the ball's movement throughout any round.

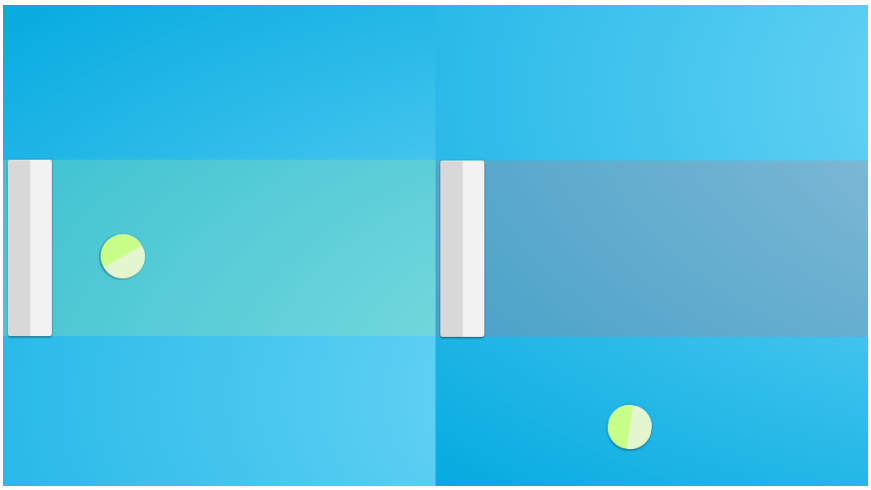

FIGURE 3.9: The simplified fitness function shown graphically, where the paddle on the left is gaining fitness (as indicated by the green-hued bar) while the paddle on the right is gaining no fitness (as indicated by the purple-hued bar).

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 [14]. 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 [16]. 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.

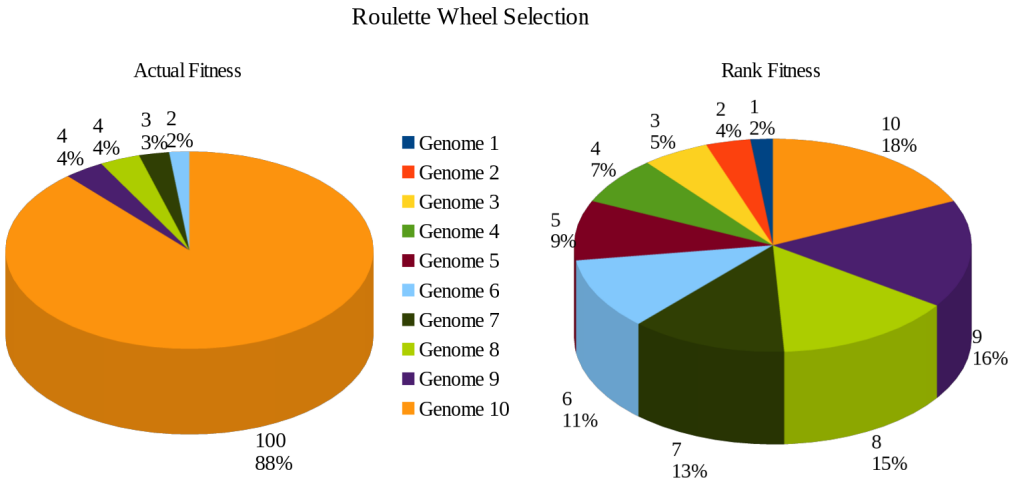

FIGURE 3.10: The roulette selection using actual fitness versus rank fitness for a population of 10 genomes. On the left, the integers are the genomes' actual fitness and the percentages are their probabilities of being selected. On the right, the integers are the genomes' rank fitness and the percentages 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).

Crossover probability was increased from 0.7 to 0.8. This 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 sufficiently random enough search to allow the genetic algorithm to escape local maxima in the fitness landscape [17]. 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 standard deviations of the mean μ. 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 lowering the number of parameters 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.

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: The genetic algorithm parameters for experiment three.

As outlined in [15], 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 used together on the same selection parents but rather were used separately, each with their own selection of parents. 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. In other words, the weighted-mean fitness 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 $\overline{PF} = \left(PF_1 * \frac{10}{41}\right) + \left(PF_2 * \frac{(41-10)}{41}\right)$ where $\overline{PF}$ is the weighted-mean parent fitness while offspring two's weighted-mean parent fitness would be calculated as $\overline{PF} = \left(PF_2 * \frac{10}{41}\right) + \left(PF_1 * \frac{(41-10)}{41}\right)$. 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. Adjustment of the crossover and mutation probabilities was handled by the adjustment parameter. Let the adjustment parameter be denoted as θ. Here θ was self-adjusted as well, as outlined in [15]. 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.

```
BEGIN
    Population P has been evaluated
    cCount = mCount = 0
    CP_sum = MP_sum = 0
    CP̄ = MP̄ = 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
    CP̄ = CP_sum / cCount
    MP̄ = MP_sum / mCount
    If P.bestFitness > P.worstFitness then
        Adjustment θ = 0.01 * (P.bestFitness − P.meanFitness) / (P.bestFitness − P.worstFitness)
    Else if P.bestFitness = P.meanFitness then
        Adjustment θ = 0.01
    End if
    If CP̄ > MP̄ then
        Crossover Probability C = C + θ
        Mutation Probability M = M − θ
    Else if CP̄ < MP̄ then
        Crossover Probability C = C − θ
        Mutation Probability M = M + θ
    End if
    Clamp C to range [0.001, 1.0]
    Clamp M to range [0.001, 1.0]
END
```

FIGURE 3.11: The self-adaptation 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 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 separately.

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.

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: The genetic algorithm parameters for experiment four.

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 together.

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

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: The genetic algorithm parameters for experiment five.

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

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 and an approximate lower bound.

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 $m = |X| * 1000 \ \frac{pixels}{second}$. 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 $m = |X| * 1000 \ \frac{pixels}{second}$. 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 (Subsections 3.5.1 through 3.5.7), followed by a comparison of all results (Subsection 3.5.8).

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) showed that the player learned 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. 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.

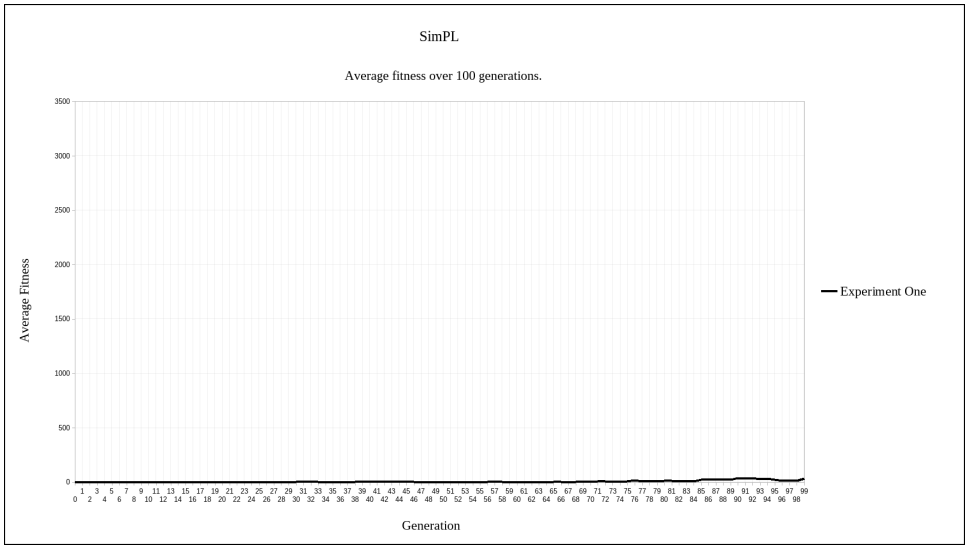

FIGURE 3.12: The average fitness over 100 generations for experiment one.

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.

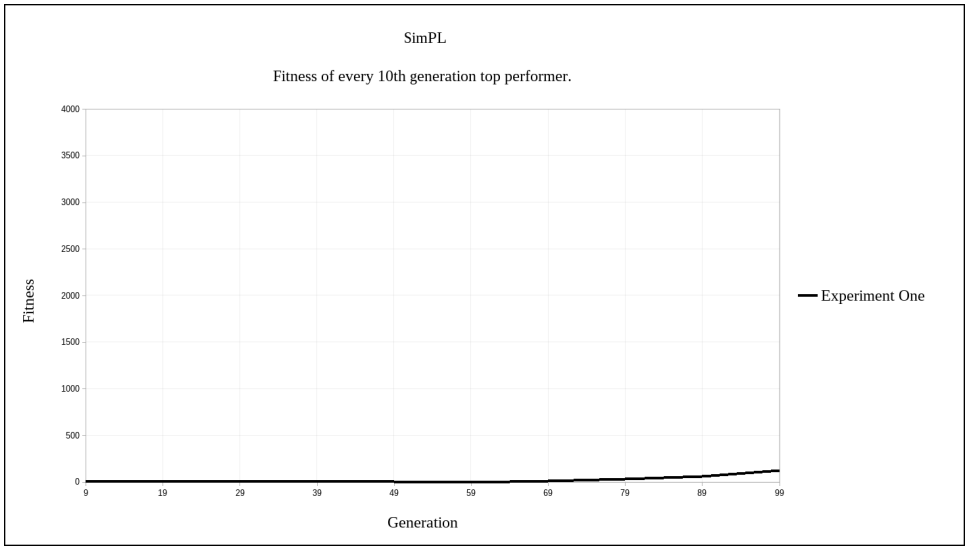

FIGURE 3.13: The fitness of every 10th generation top performer for experiment one.

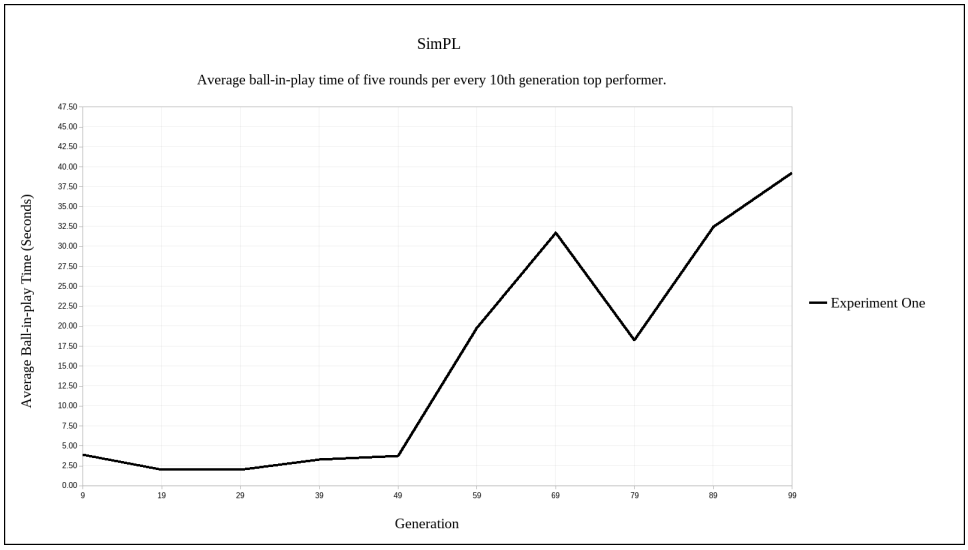

FIGURE 3.14: The average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment one.

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

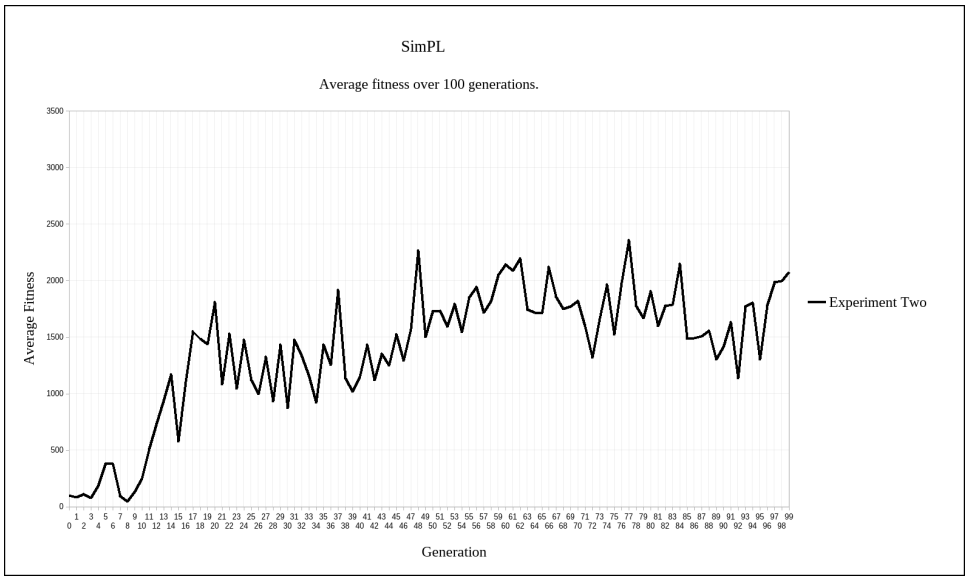

FIGURE 3.15: The average fitness over 100 generations for experiment two.

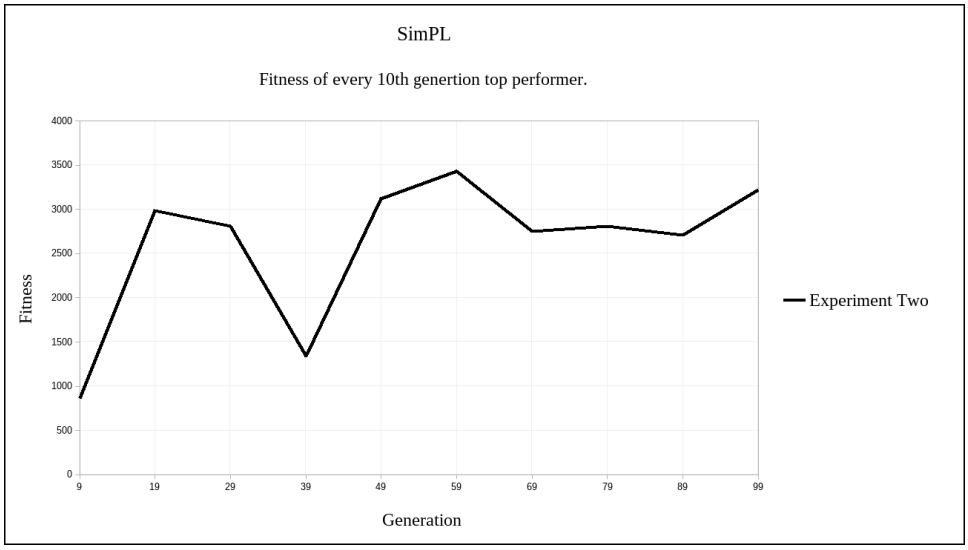

FIGURE 3.16: The fitness of every 10th generation top performer for experiment two.

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.

The crossover and mutation probabilities were both initially set to 0.5 before the start of the ex-

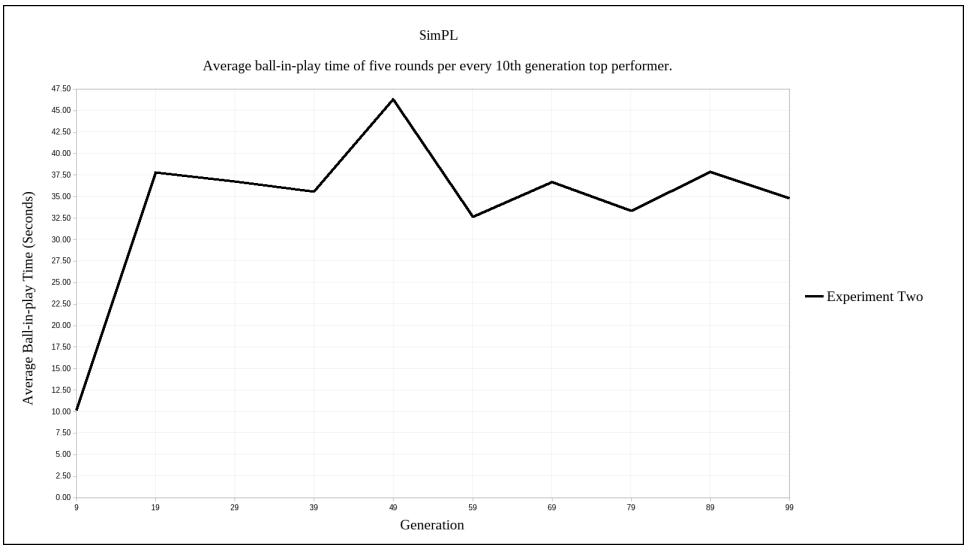

FIGURE 3.17: The average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment two.

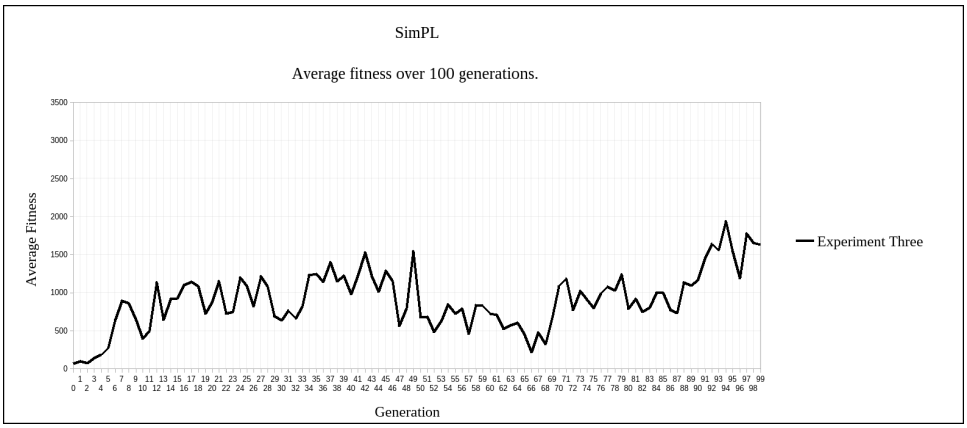

FIGURE 3.18: The average fitness over 100 generations for experiment three.

periment. 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. 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 [15].

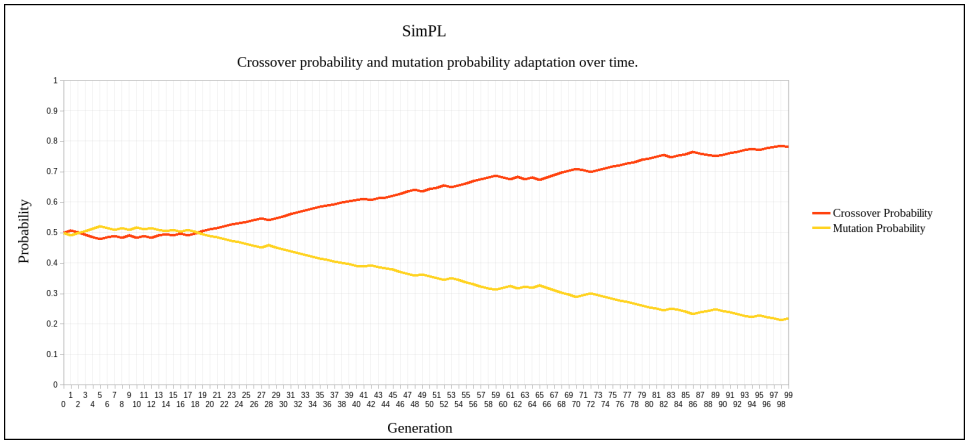

FIGURE 3.19: The self-adaptation of the crossover and mutation probabilities for experiment three.

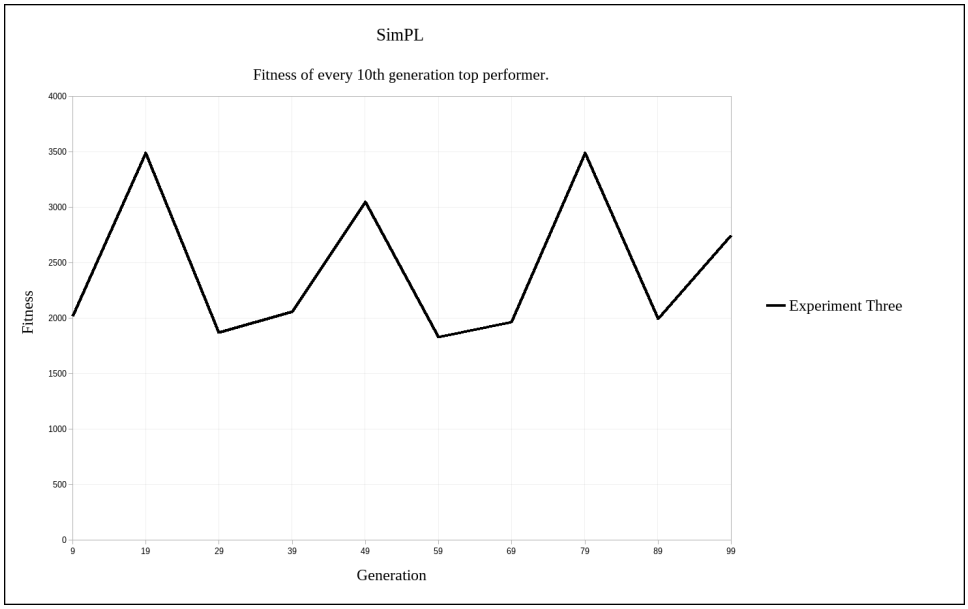

FIGURE 3.20: The fitness of every 10th generation top performer for experiment three.

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 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.

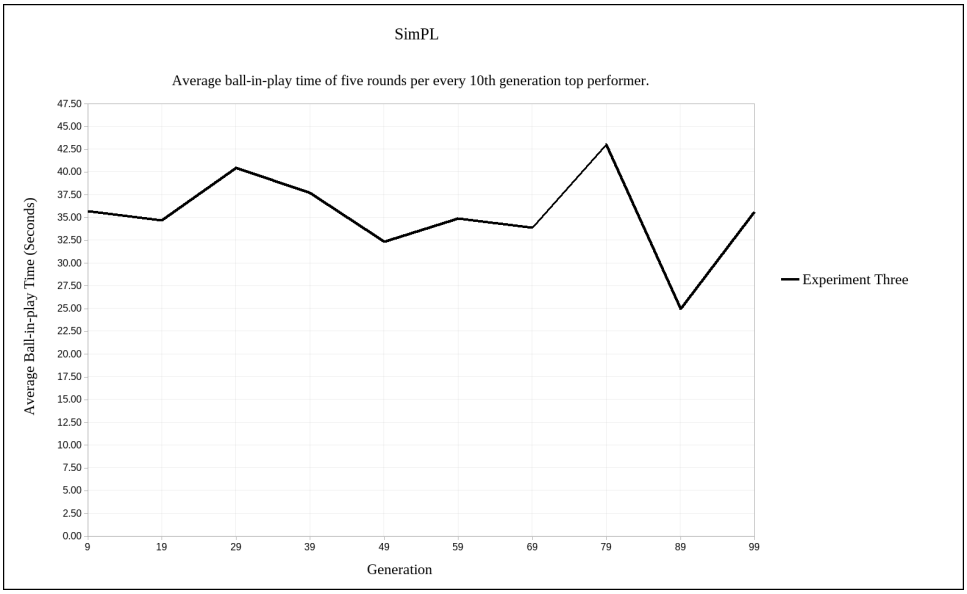

FIGURE 3.21: The average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment three.

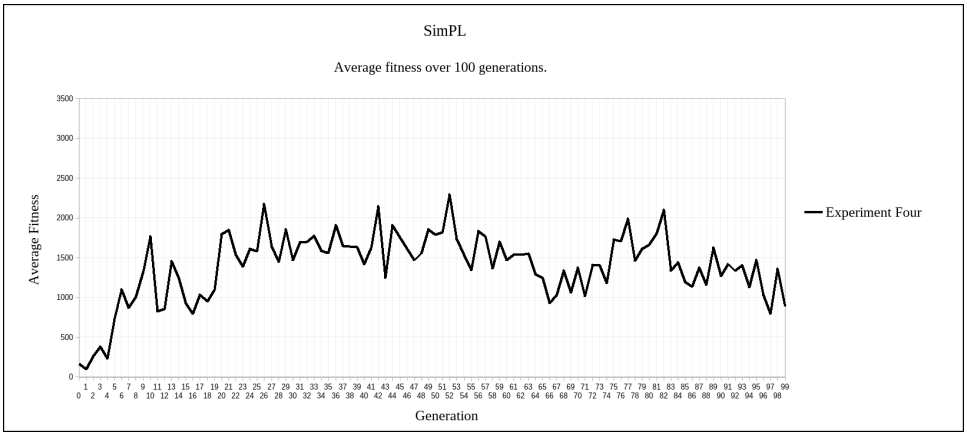

FIGURE 3.22: The average fitness over 100 generations for experiment four.

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.

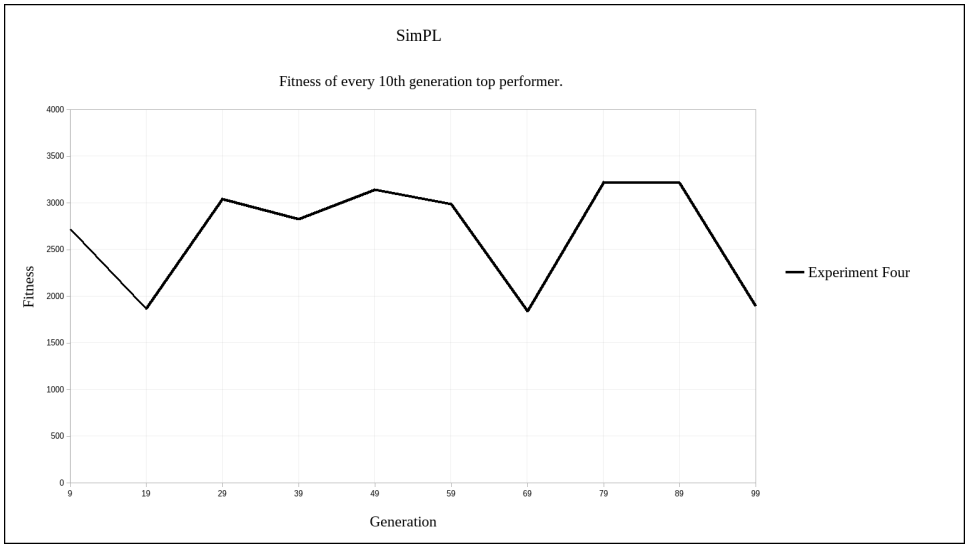

FIGURE 3.23: The fitness of every 10th generation top performer for experiment four.

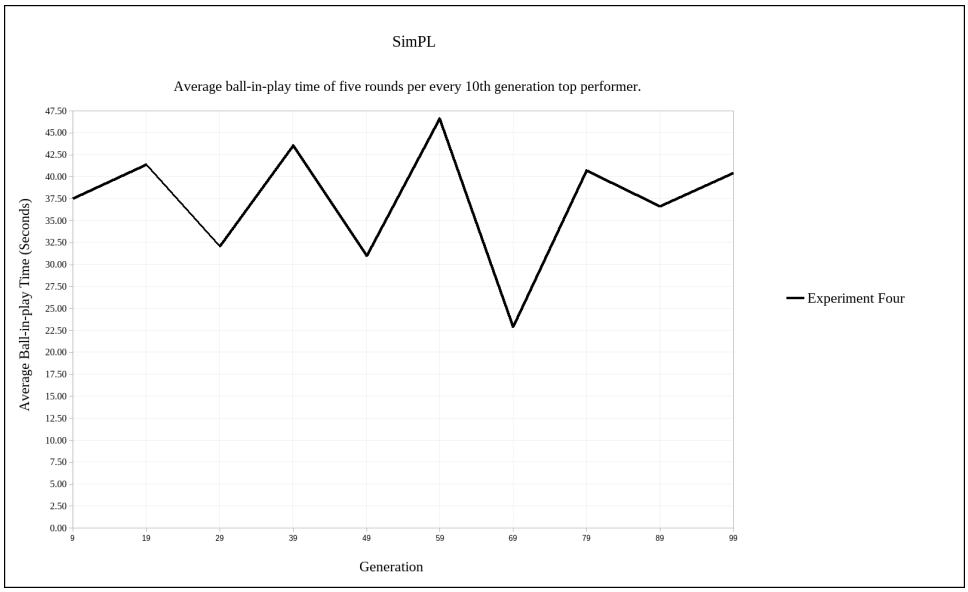

FIGURE 3.24: The average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment four.

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

Experiment six (Figures 3.28-3.30) illustrates the *perfect* results possible for 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

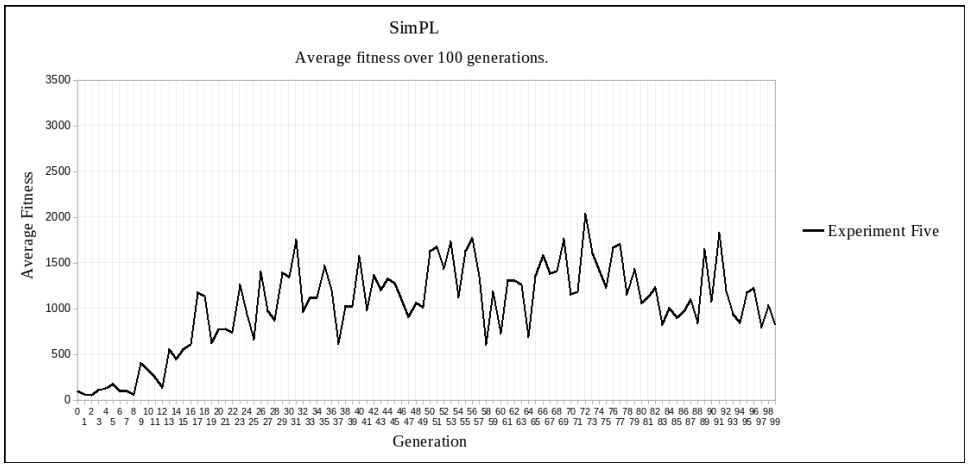

FIGURE 3.25: The average fitness over 100 generations for experiment five.

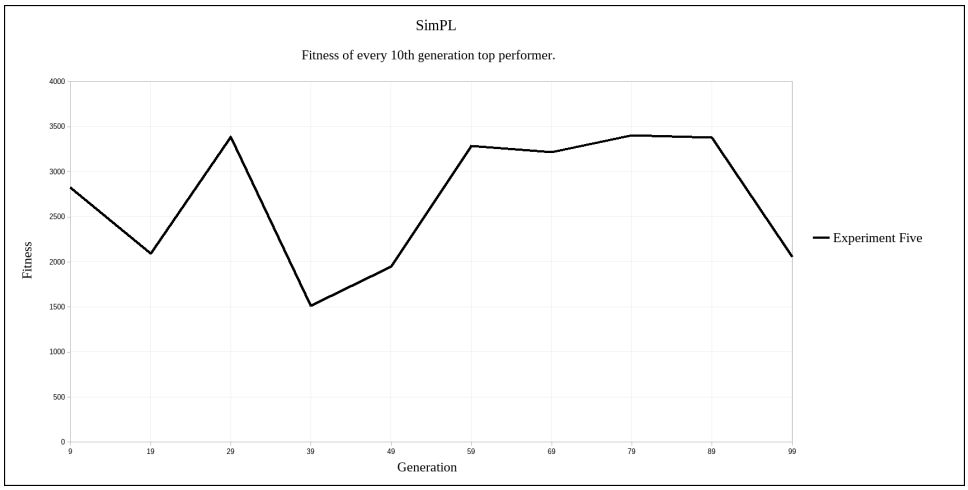

FIGURE 3.26: The fitness of every 10th generation top performer for experiment five.

five (above) comes closest to optimal performance.

3.5.7 Experiment seven: random paddles and an approximate lower bound.

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.

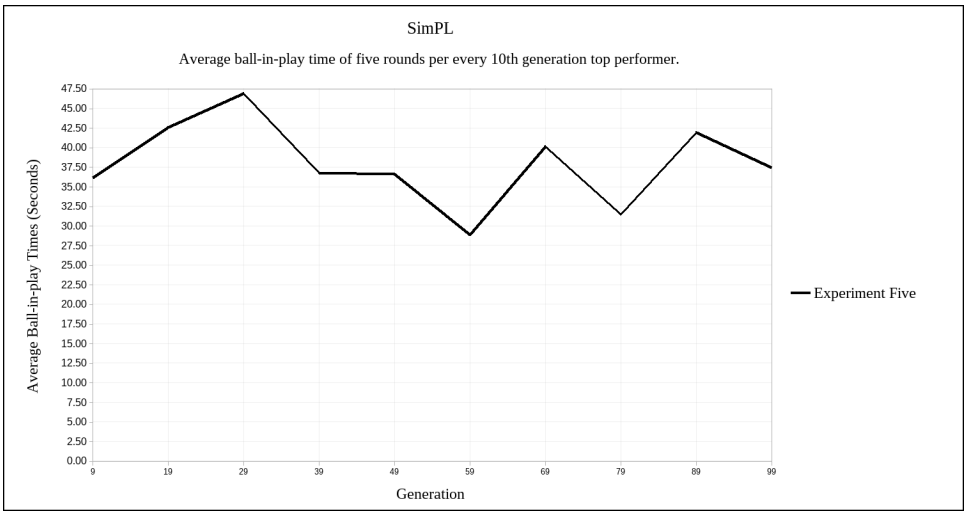

FIGURE 3.27: The average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment five.

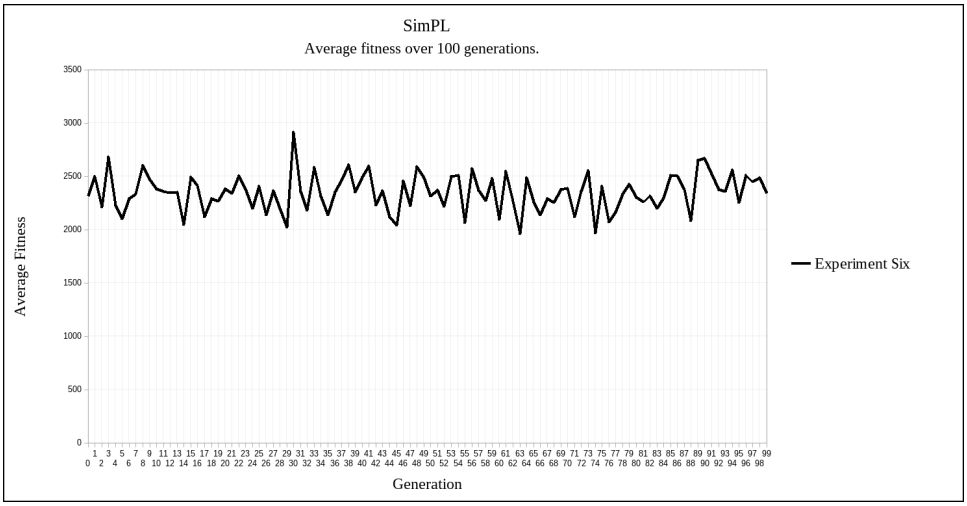

FIGURE 3.28: The average fitness over 100 generations for experiment six.

3.5.8 Comparative Results

This section illustrates comparative results by plotting all curves on the same axes (Figures 3.34 through 3.36). Figure 3.34 shows the average fitness over 100 generations for experiments one through seven. Figure 3.35 shows the fitness of every 10th generation top performer for experiments one through seven. Lastly, Figure 3.36 shows the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiments one through seven.

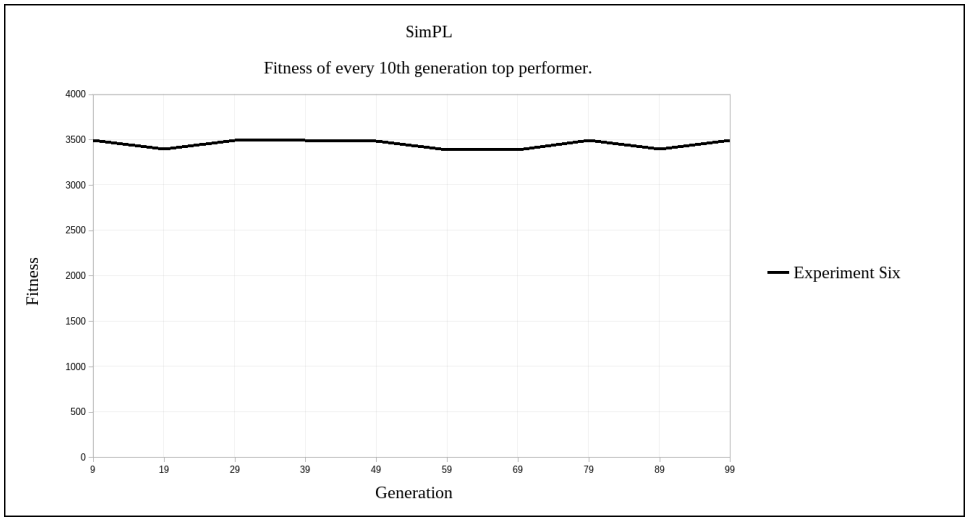

FIGURE 3.29: The fitness of every 10th generation top performer for experiment six.

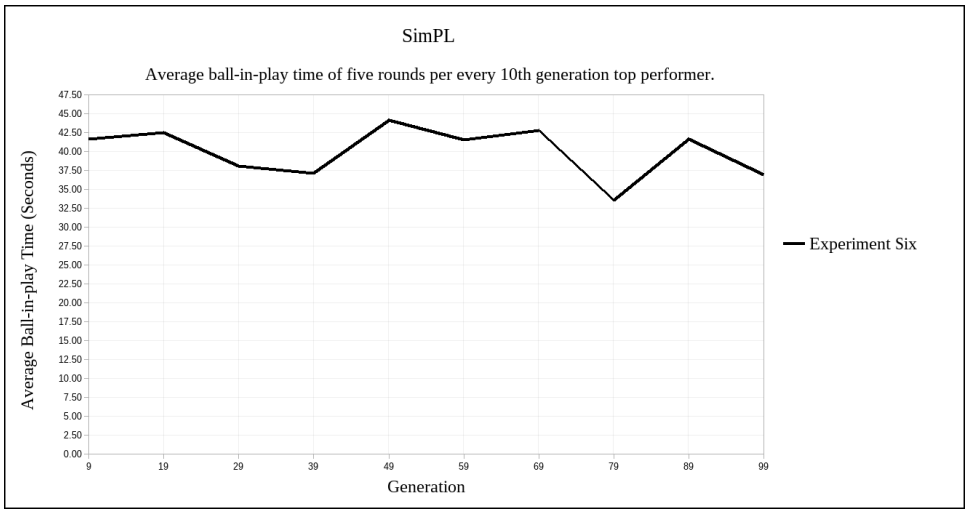

FIGURE 3.30: The average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment six.

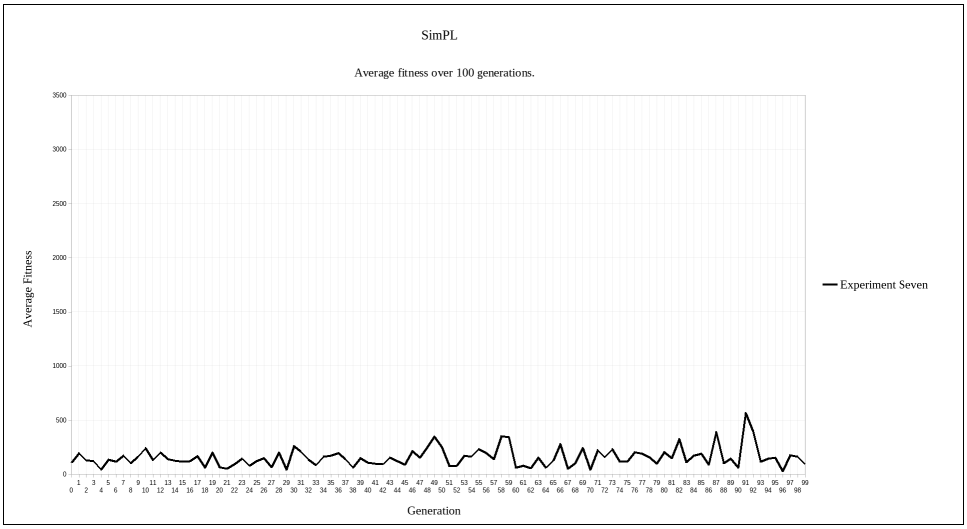

FIGURE 3.31: The average fitness over 100 generations for experiment seven.

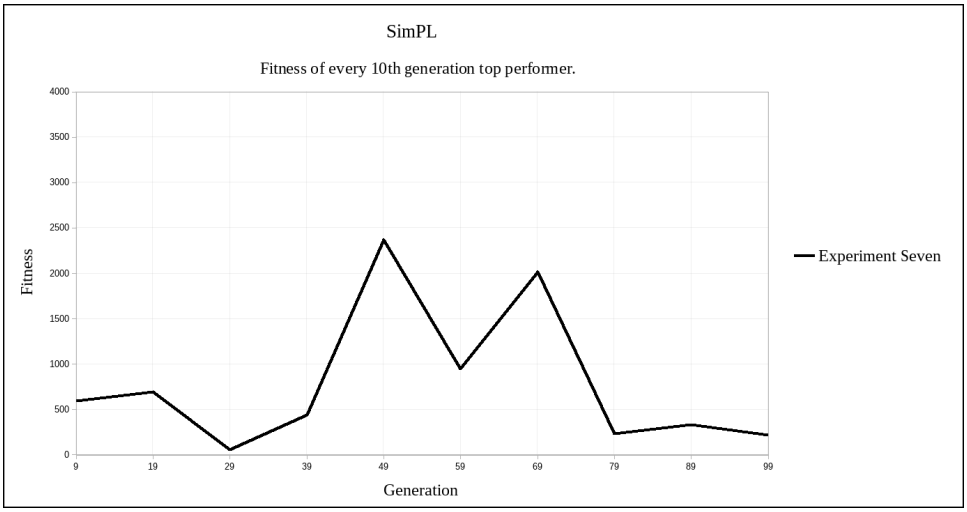

FIGURE 3.32: The fitness of every 10th generation top performer for experiment seven.

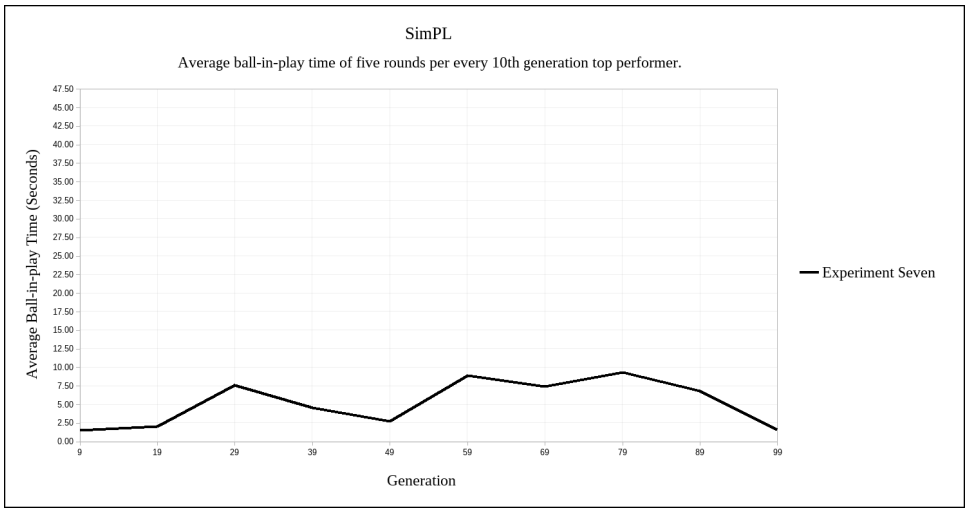

FIGURE 3.33: The average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment seven.

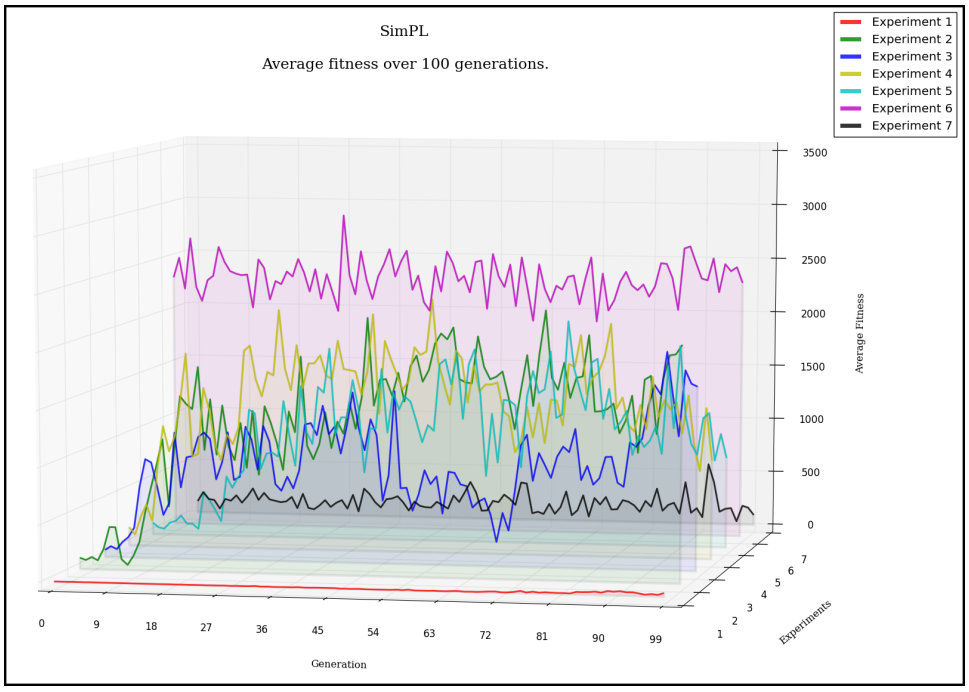

FIGURE 3.34: The average fitness over 100 generations for experiments one through seven.

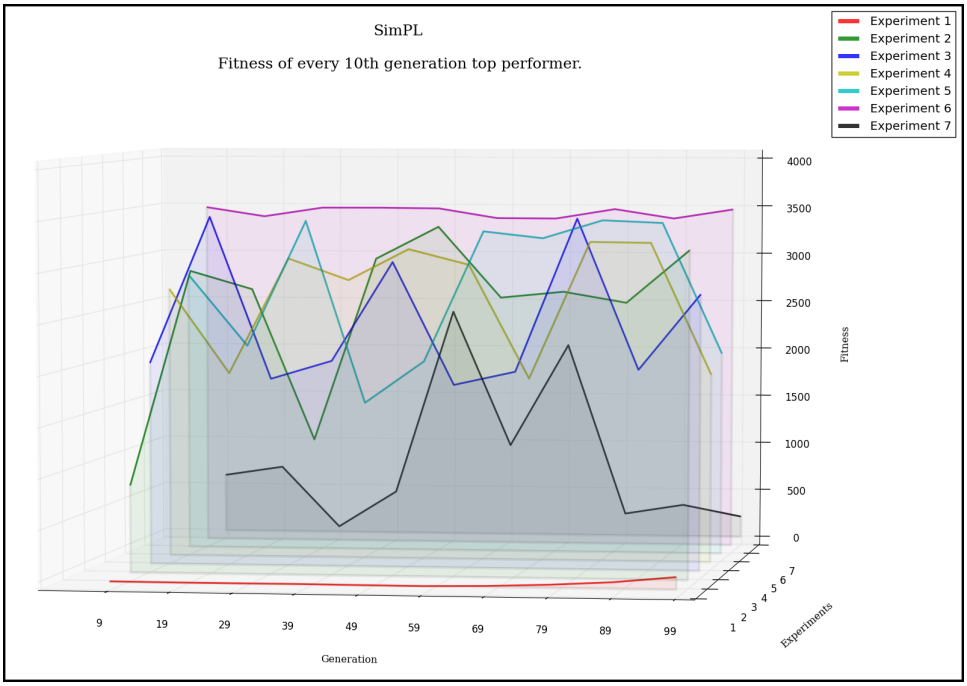

FIGURE 3.35: The fitness of every 10th generation top performer for experiments one through seven.

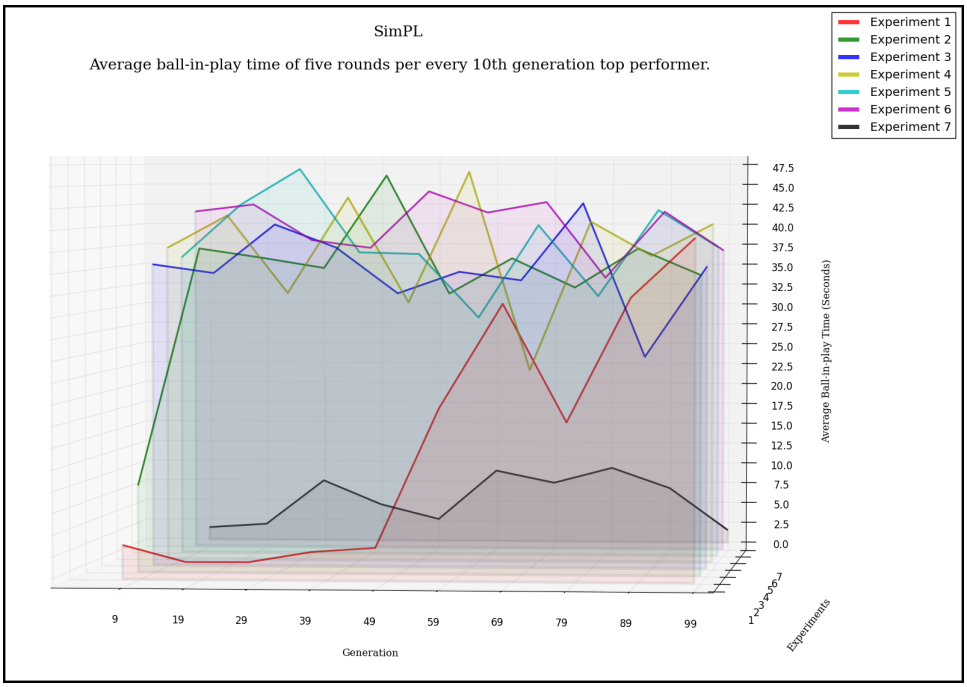

FIGURE 3.36: The average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiments one through seven.

Chapter 4

BBAutoTune

4.1 Overview

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

4.2 Implementation

BBAutoTune's implementation builds off a 3D content creation platform known as Blender. Blender's various features allow one to author static or real-time interactive 3D content. Figure 4.1 shows Blender's GUI. 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 $17.64cm \times 14.53cm \times 14.33cm$ including the Braille hat that rests above the body of the robot [3]. 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 4.2. The robot's base was centered at the world origin and the robot was faced down

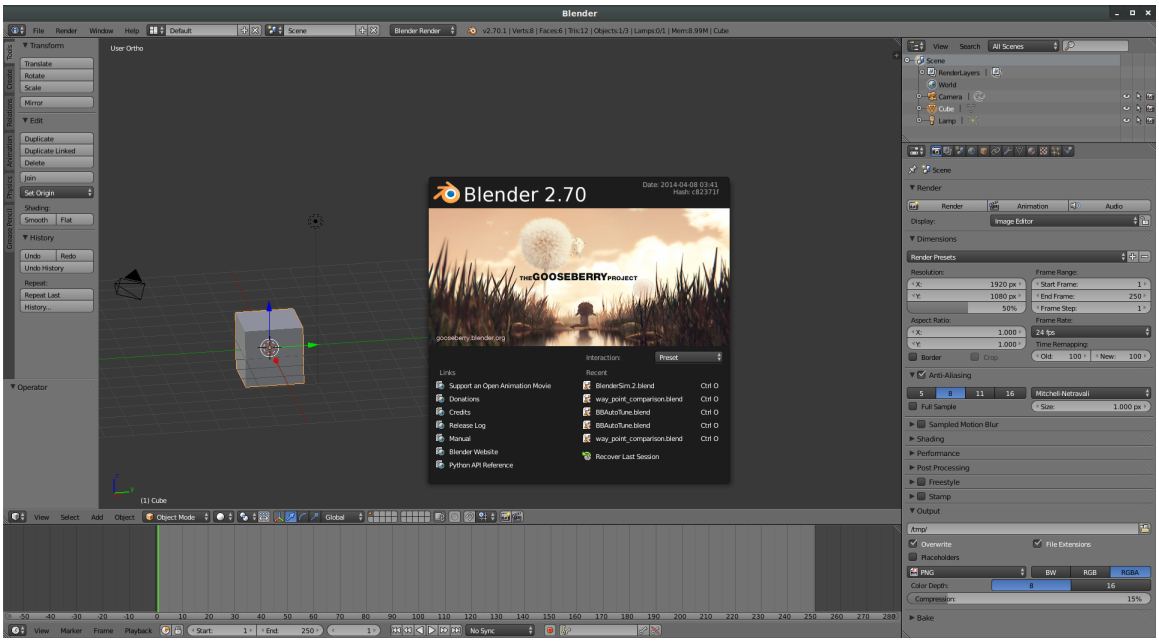

FIGURE 4.1: The Blender interface.

the global positive x-axis. All wheel geometry origins had a global z-axis position of 0.0 and were placed 1.37cm above the floor.

4.2.2 GUI

Blender's GUI 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 maximum number of generations specified. See Figure 4.3.

4.2.3 Physics Engine API

To provide the capability of authoring 3D interactive content, Blender uses a real-time physics engine known as Bullet. Blender exposes the Bullet 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. Ignoring duplicate ways to set the same parameter and duplicate parameters per dimension, there

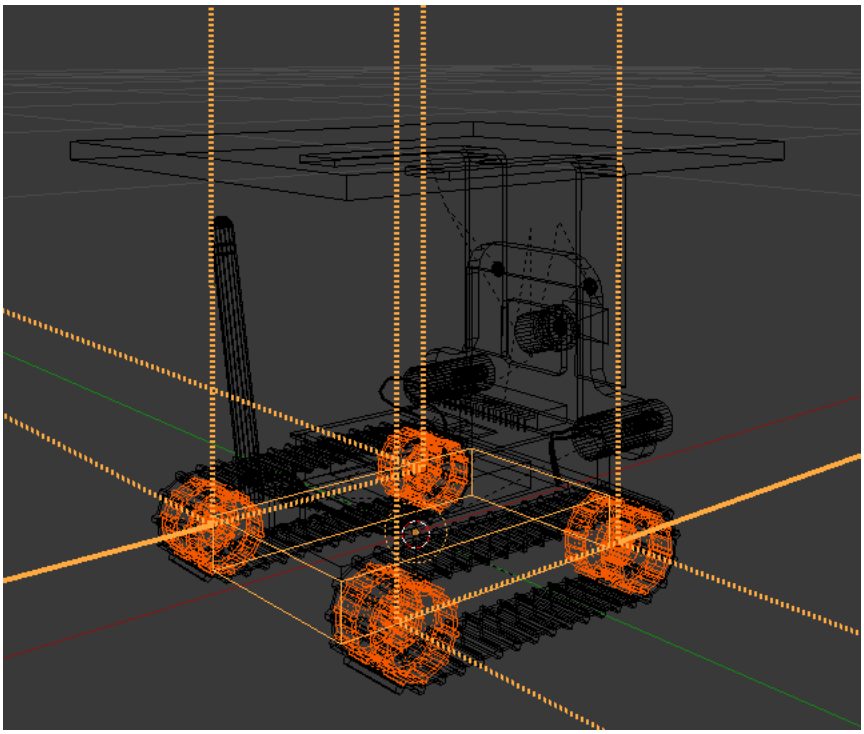

FIGURE 4.2: The 3D model of the Surveyor SRV-1 Blackfin with the wheels connected to the base via rigid body hinge joints.

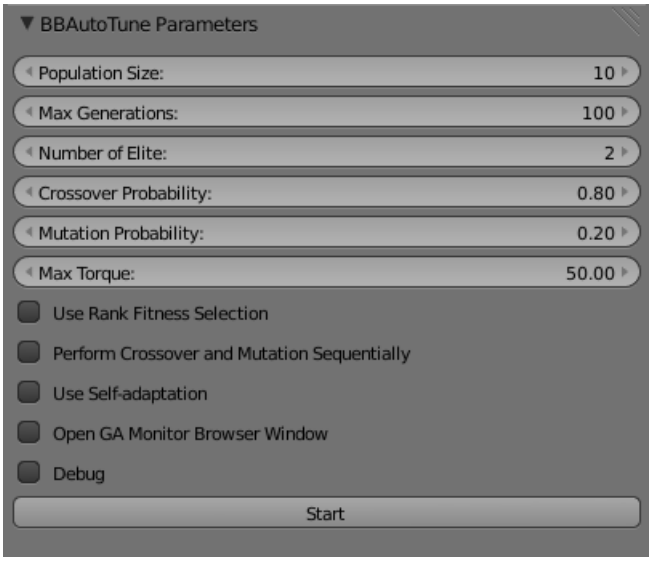

FIGURE 4.3: The BBAutoTune GUI panel added to Blender.

are 42 different physics engine parameters that can be set via the API.

4.2.3.1 Parameter Set

Blender's physics parameters can be set via its GUI or API. The API provides multiple avenues to set the same parameter depending on whether or not an API call is made from within Blender's built-in game engine or not. Each physics parameter is set to some default value at the start of Blender. All parameters have a range of either discrete or continuous values. Parameters such a *gravity* and *mass* are intended to reflect the real-world phenomena with the same names. Other parameters are more obscure such as:

1. *use material physics*—determines whether or not a material's physical properties (such as its friction coefficient and its collision elasticity) are taken into account during simulation;
2. *material force*—specifies an upward repellent spring like force;
3. *material damping*—dampens a material's spring force;
4. *material distance*—the distance at which a material's physical properties have an affect;
5. *material align to normal*—align physics objects along their surface normal;
6. *use actor*—determines if a physics object is seen by near and/or radar sensors;
7. *use ghost*—determines whether or not a physics object registers a collision when colliding with other physics objects;
8. *no sleeping*—if set, a physics object's position and orientation is still calculated by the physics engine even if the physics object is at rest;
9. *form factor*—the higher the value, the lower the likelihood a rigid body will roll when acted upon by some outside force;
10. *use collision bounds*—if set to false, a physics object's collision bounds is defaulted to a sphere shape;
11. *max physics steps*—specifies the maximum number of physics engine iterations per rendered frame;
12. *physics sub-steps*—specifying a higher value results in a more precise physical simulation model;

13. *FPS*—specifies the fixed physics time step such that the time step equals $\frac{1}{FPS}$ and is independent of the rendering frame rate; and

14. *deactivation time*—specifies the amount of time at which the physics engine will not longer calculate the position and orientation of a physics object that has a (angular or linear) velocity below some threshold [web].

See Table 4.1 for the full 41 parameter set complete with each parameter's valid range and default value.

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[1] state S of the robot's base with a time stamp T. After one second has passed, the robot monitor records the updated 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, T, S', 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

[1] Position meaning its world position in the x, y, and z dimensions and rotation meaning its world orientation around the x, y, and z axes.

Parameter	Range	Default Value
Gravity	$[0.0\frac{m}{s^2}, 10000.0\frac{m}{s^2}]$	$9.8\frac{m}{s^2}$
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
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
Physics Type	[NO_COLLISION, STATIC, DYNAMIC, RIGID_BODY, SOFT_BODY, OCCLUDE, SENSOR, NAVMESH, CHARACTER]	STATIC
Use Actor	[False,True]	True
Use Ghost	[False,True]	False
Use Material Force Field	[False,True]	False
Rotate From Normal	[False,True]	False
No Sleeping	[False,True]	False
Mass	[0.0,10000.0]	1.0
Radius	[0.01m,inf]	1m
Form Factor	[0.0,1.0]	0.4
Use Anisotropic Friction	[False,True]	False
Anisotropic Friction X,Y,Z	[0.0,1.0]	1.0
Velocity Minimum/Maximum	[0.0,1000.0]	0.0
Lock Translation X,Y,Z	[False,True]	False
Lock Rotation X,Y,Z	[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 Bounds Type	[BOX, SPHERE, CYLINDER, CONE, CONVEX_HULL, TRIANGLE_MESH, CAPSULE]	BOX
Collision Margin	[0.0m,1.0m]	6cm
Force	[-inf,inf]	0.0
Torque	[-inf,inf]	0.0
Linear Velocity	[-inf,inf]	0.0
Angular Velocity	[-inf,inf]	0.0
Use Local Force	[False,True]	True
Use Local Torque	[False,True]	True
Use Local Linear Velocity	[False,True]	True
Use Local Angular Velocity	[False,True]	True
Damping Frames	[-32768,32767]	0

TABLE 4.1: 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.

fitness, average fitness, lowest fitness, crossover probability, and the mutation probability. See Figure 4.4. 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.

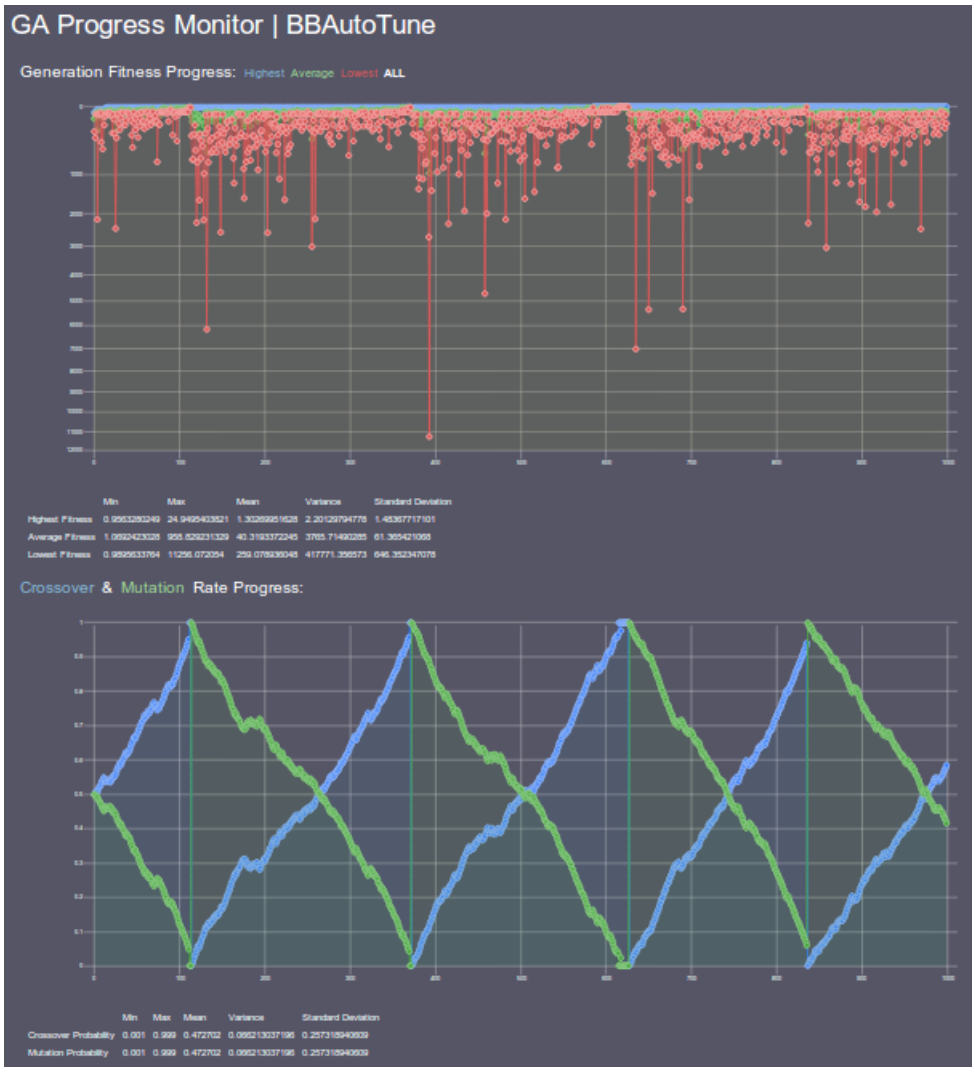

FIGURE 4.4: The external GA progress monitor.

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 considered beter 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 (see Table 4.1). 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 decoding function was $p_i = (g_i * (r_{max} - r_{min})) + r_{min}$. For an integer type, the decoding function was the same as for a float type except the whole value was floored. For an array A of strings type with size n (for example the collision bounds type parameter), the decoding function was $p_i = A[\lfloor g_i * (n-1) \rfloor]$. Finally for a boolean type, the decoding function was

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

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 [18]. 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 $\left[0, \frac{n(n-1)}{2}\right]$. 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 Figures 4.5 and 4.6.

```
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 = Σ_{k=0}^{i} (P[k].rankFitness)
    End for
    Select a random number r = unif(0, n(n−1)/2)
    Genome selected G
    For i=0 to n − 1 do
        If P[i].prefixSum ≥ r then
            G = P[i]
            Break
        End if
    End for
    Return G
END
```

FIGURE 4.5: The rank fitness selection algorithm.

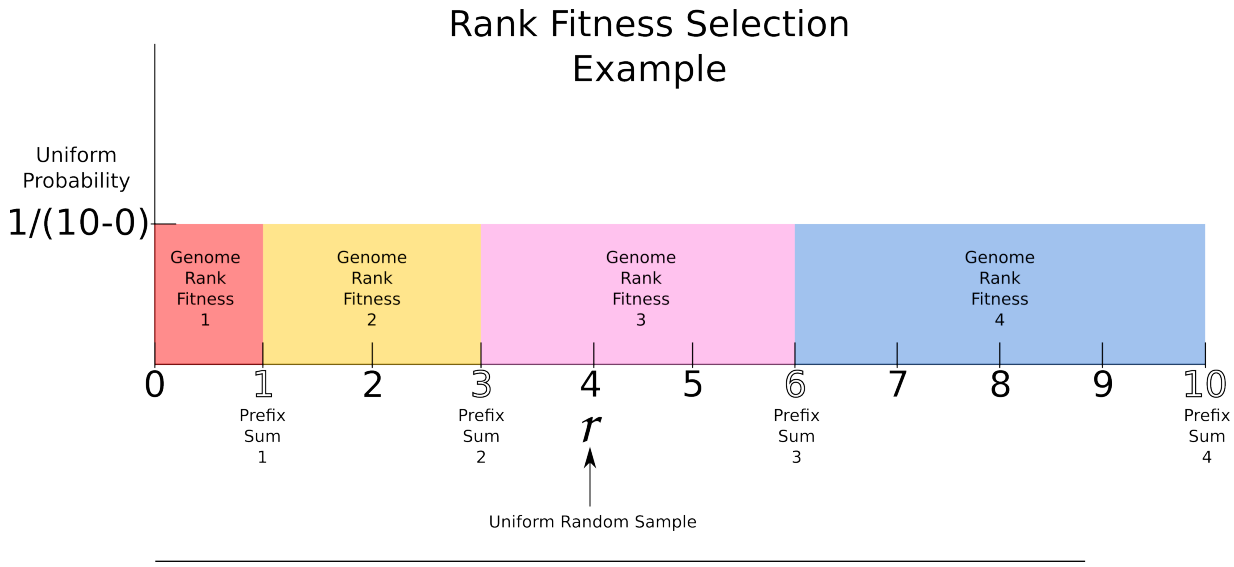

FIGURE 4.6: An illustrative rank fitness selection example. The population consists of four genomes where the rank fitnesses are one, two, three, and four. The prefix-sums are one, three, six, and 10. A random integer is sampled from a uniform distribution between zero and 10. Since the random number was four, the genome with a rank fitness of 3 would be selected. Notice that genomes with a higher rank fitness, have a greater uniform probability of being selected.

4.2.7.3 Fitness Function

To construct the fitness function, real robot motion data was collected which included 1040 sample points. Each sample point consisted of the x-position, y-position, and heading (z-orientation) of the robot before it moved and the x-position, y-position, and heading after the robot moved. 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 each forward command was recorded. The robot's position and orientation were 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 was translated and rotated to be in the same reference frame such that the robot was always at the arena origin facing down the global positive x-axis before performing the forward command. See Figures 4.7 and 4.8. 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. With all of the data transformed and rotated, each initial state had a x-position of 0.0, a y-position of 0.0, and an initial orientation of 0.0 degrees.

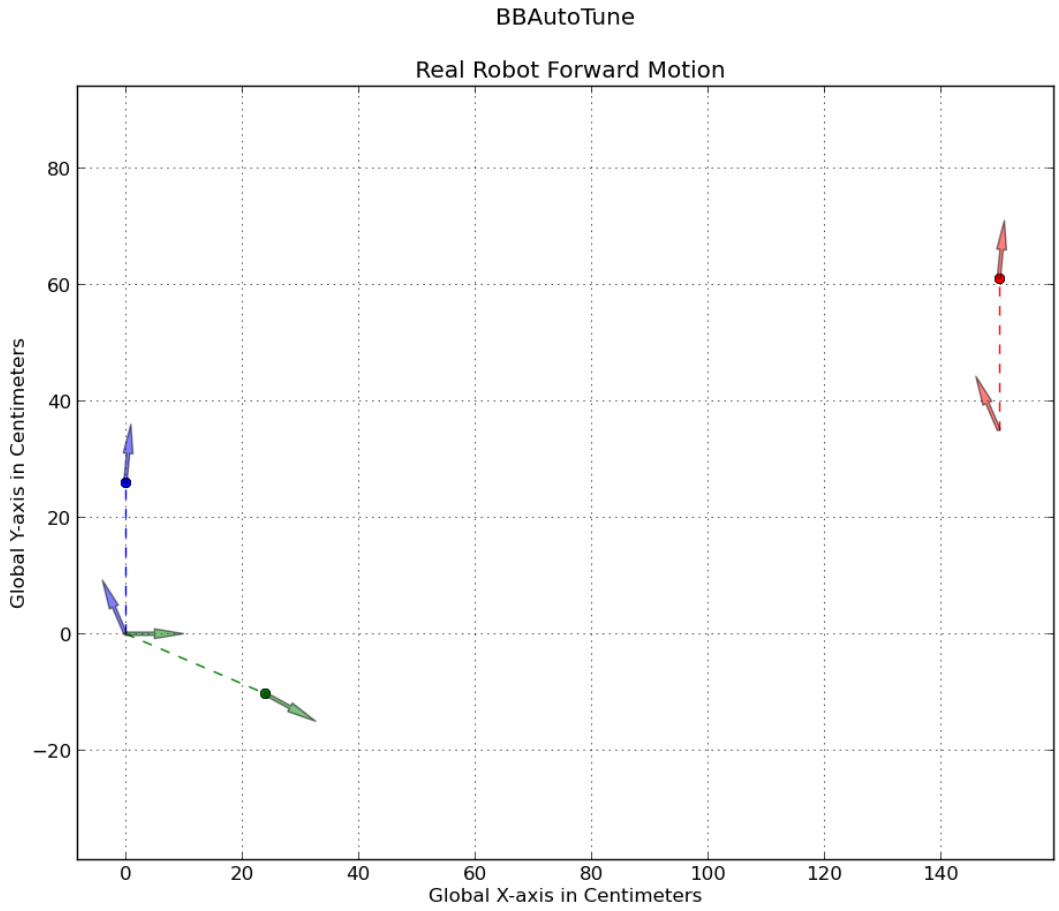

FIGURE 4.7: A specific instance of how the raw real robot forward motion data was translated and rotated such that each pair of initial and final states have the same reference frame. The red arrows, dot, and line represent the raw initial and final state of the robot recorded from the overhead camera 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 global 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.

Analyzing the distribution of final state x-positions, y-positions, and headings separately, all are unimodal and nearly symmetric with only a slight skew from their respective means. See Table 4.2 and Figure 4.9. When viewing the dimensions together, a large mass is centered around the point: 23.8644631679cm,

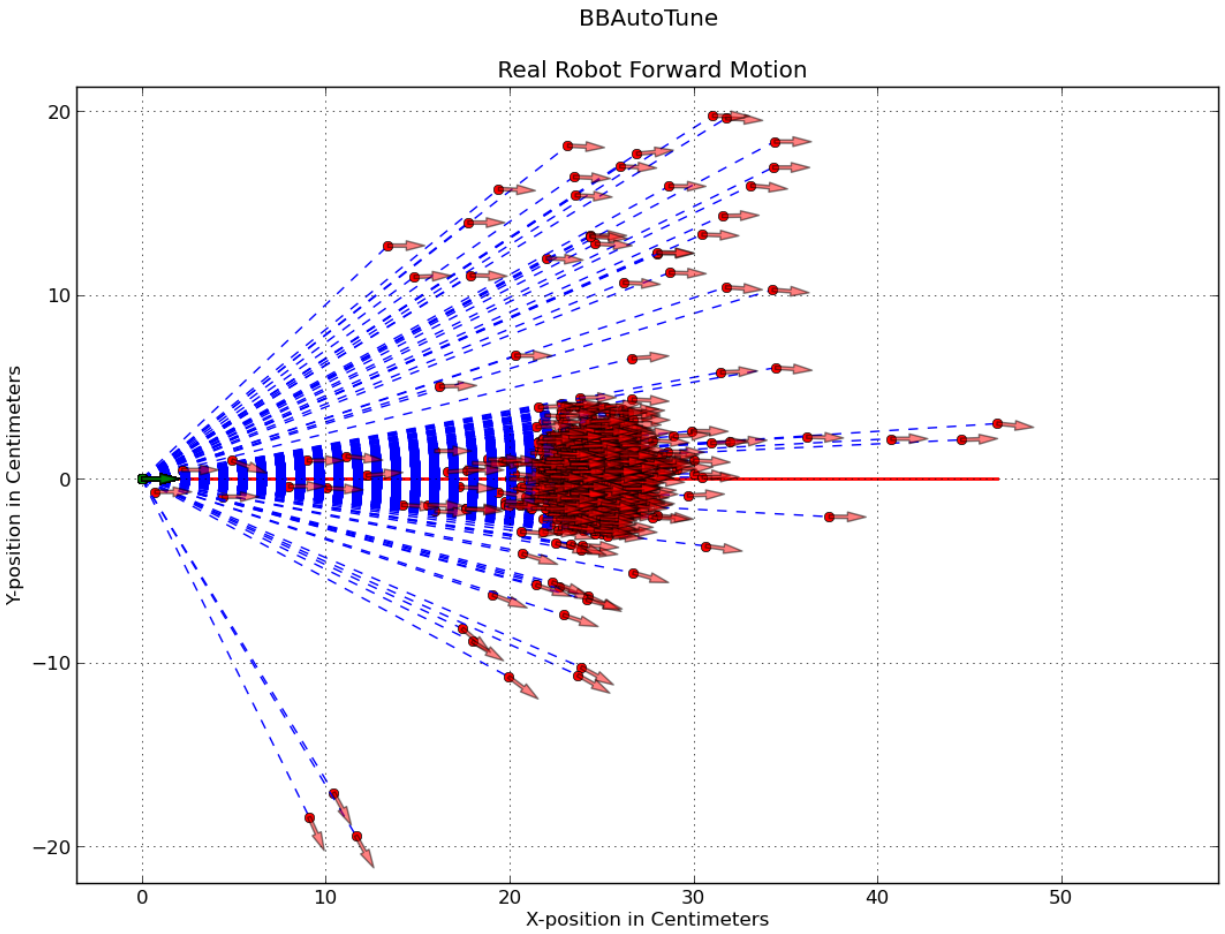

FIGURE 4.8: The collected real robot forward motion plotted from the same reference point. The green dots and arrows indicate the starting position and orientation of the real robot before the forward command was performed. The red dots and arrows indicate the ending position and orientation of the real 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 finals states.

0.338269853117cm, and -0.00417473025048rad. See Figure 4.10. Based on the variance-covariance matrix, all three dimensions vary together and are not independent of one another.

	X-position	Y-position	Heading
Maximum	46.5183415482cm	19.7486813209cm	6.23776rad
Minimum	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-position	10.4060572	2.3348963	0.0685591
Covariance with Y-position	2.3348963	9.47354175	0.08654865
Covariance with Heading	0.0685591	0.08654865	0.15261486

TABLE 4.2: The distribution metrics of final state x-positions, y-positions, and headings after the real robot moved forward.

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 moved closer to the centroid of the real robot motion, the simulated motion would become increasingly indistinguishable from the real motion. Various distance functions were considered and the Mahalanobis distance (MD) was chosen as the basis for the fitness function. The MD is a generalized form of the Euclidean distance, such that the MD accounts for the correlation in the data set since it is computed using the inverse of the variance-covariance matrix [19]. For uncorrelated data, the MD reduces to the Euclidean distance [20].

Noise introduced by 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 MD skewed. To account for the potential outliers in the real motion data set, a robust[2] 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 [22][23]. The classical mean location was 23.8644632cm, 0.338269853cm, and -0.00417473025rad for final state x-positions, y-positions, and headings respectively while the robust mean location was 23.9934044cm, 0.0351240536cm, and -0.0189964938rad for

[2] *Robust* meaning the robustness of an estimator's resistance to outliers or data point contamination [21].

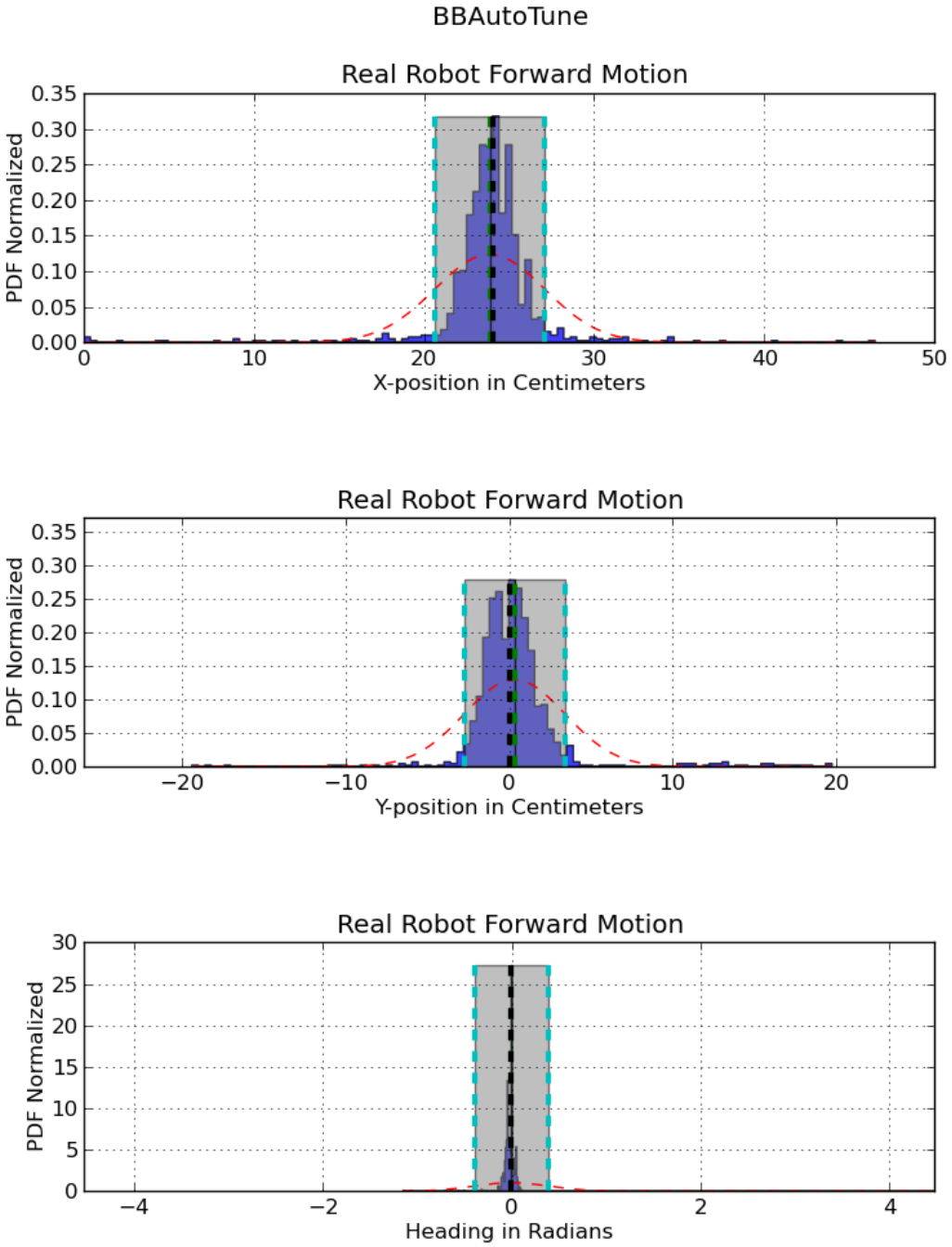

FIGURE 4.9: The distributions of final state x-positions, y-positions, and headings (after being translated and rotated to the same reference frame) recorded for the real robot forward motion. The broken black bar represents the mode, the green broken bar represents the mean, the cyan broken bars represent one standard deviation from the mean, and the red broken curve represents the best fit normal curve.

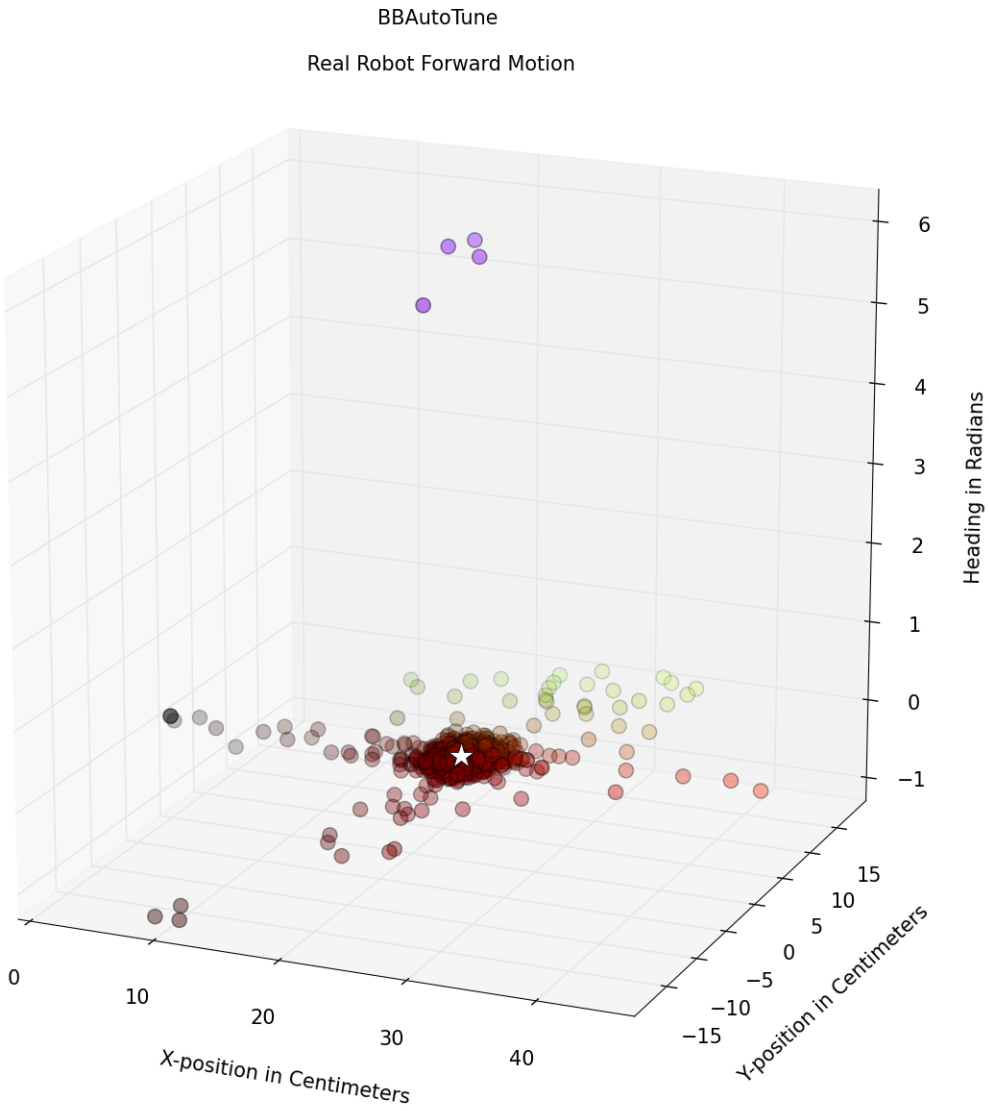

FIGURE 4.10: A 3D scatter plot of final state x-positions, y-positions, and heading values for the real robot after it performed the forward command. Positive values of x-position, y-position, and heading contribute a portion of red, green, and blue respectively to each scatter plot. The star located at the center of the large mass of points is the centroid.

final state x-positions, y-positions, and headings respectively. The left matrix below is the classical variance-covariance matrix, while on the right is 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 4.11. These samples were 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 MD calculation, the robust distance (RD) for any sample point can be computed [21]. Comparisons between the MD and the RD for each of the 1040 real robot motion data points are shown in Figure 4.12. As the data points travel away from the centroid, the RD increases more rapidly than the MD.

Only three out of the total six degrees of freedom were recorded for the real robot. However, in Blender there were an additional three degrees of freedom (pitch, roll, and heave or up and down) 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 any rotation around the global x-axis, any rotation around the global y-axis, and/or any translation up or down the global z-axis. Additionally, a time penalty was added onto the robust distance by the amount of time in seconds the simulated robot took to evaluate (or rather took to come to a complete stop) greater than one second with a maximum penalty of 15 seconds since any given evaluation period only lasted a total of 16 seconds. This time penalty was necessary since it was observed that the real robot never took more than one second to come to a complete stop after moving forward. Without a time penalty, for example, a simulated robot could move just as the real robot and it would be given an erroneously high fitness even though it took 16 seconds to come to a complete stop. This undesirable scenario was avoided by using a time penalty.

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

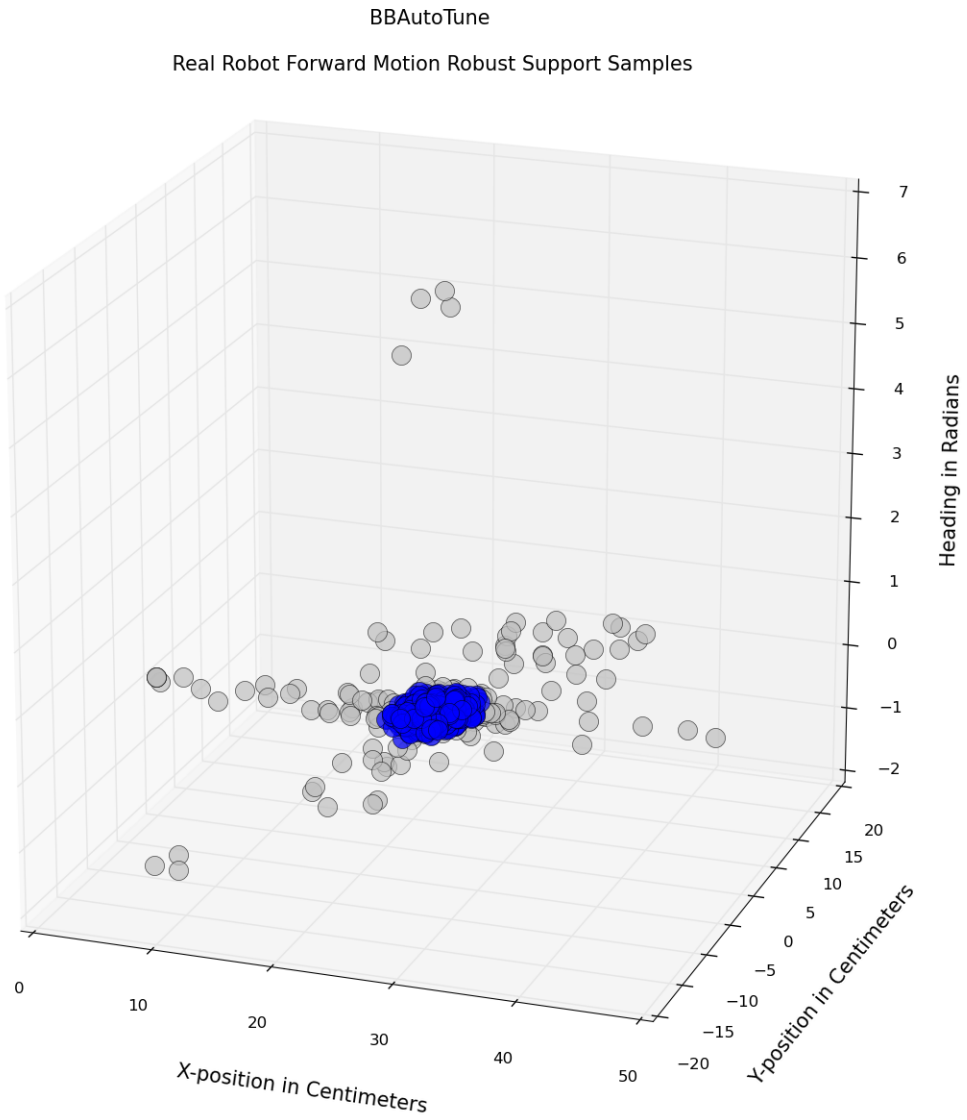

FIGURE 4.11: A 3D scatter plot of the support samples in blue used to calculate the robust mean location and the robust variance-covariance matrix as returned by the Fast-MCD algorithm.

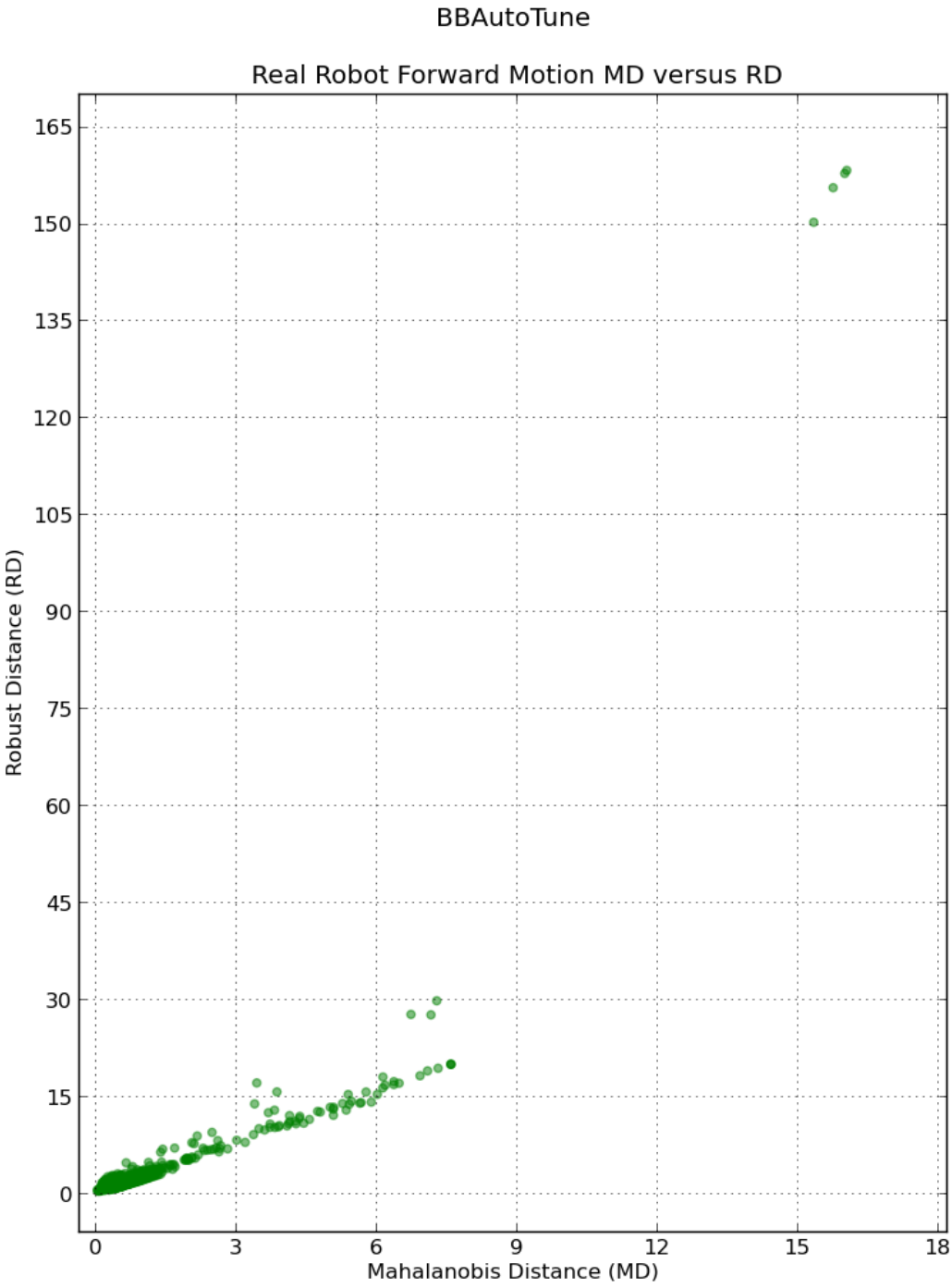

FIGURE 4.12: A scatter plot comparing the Mahalanobis distance versus the robust distance for each data point collected of the real robot motion.

collected by the robot monitor for use in the fitness function. Let $S = [x_p, y_p, z_p, x_o, y_o, z_o, T]$ be the starting state of the simulated robot at the beginning of the evaluation period at time $S[6] = T$ where the subscript $_p$ refers to position and the subscript $_o$ refers to orientation. Let $S' = [x_p, y_p, z_p, x_o, y_o, z_o, T']$ be the ending state of the simulated robot at the end of the evaluation period at time $S'[6] = T'$. Also, let $RD(x - position, y - position, z - orientation)$ be the robust distance. The fitness of G_i was defined as $Fitness(S, S') = RD(S'[0], S'[1], S'[5]) + |S'[4] - S[4]| + |S'[3] - S[3]| + |S'[2] - S[2]| + ((S'[6] - S[6]) - 1) = G_i$'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-position, y-position, and z-orientation (heading) after moving forward was 23.9934044cm, 0.0351240536cm, and -0.0189964938rad respectively.

- The simulated robot came to a complete stop 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 global x-axis, and its local origin was placed at the world origin. See Figure 4.13. 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.

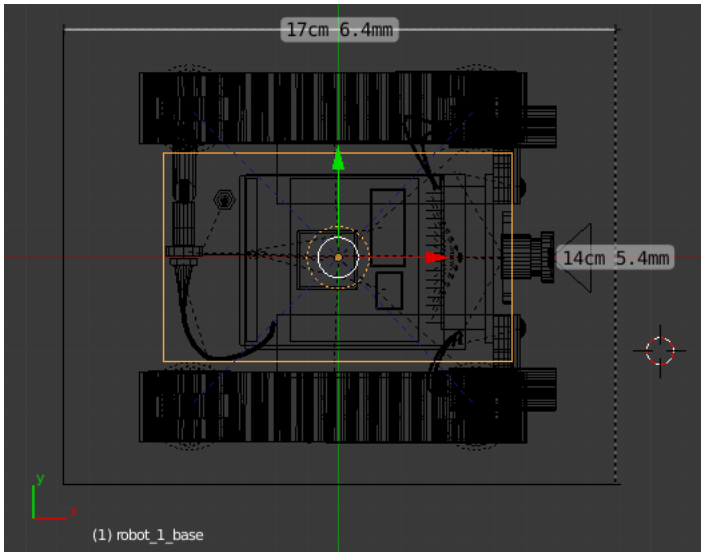

FIGURE 4.13: The simulated robot's local coordinate system axis aligned with the world coordinate system.

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 influence 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 racquet controlled via a Python script. See Figure 4.14. Note that the ball was allowed to travel in all three dimensions and was completely physics based.

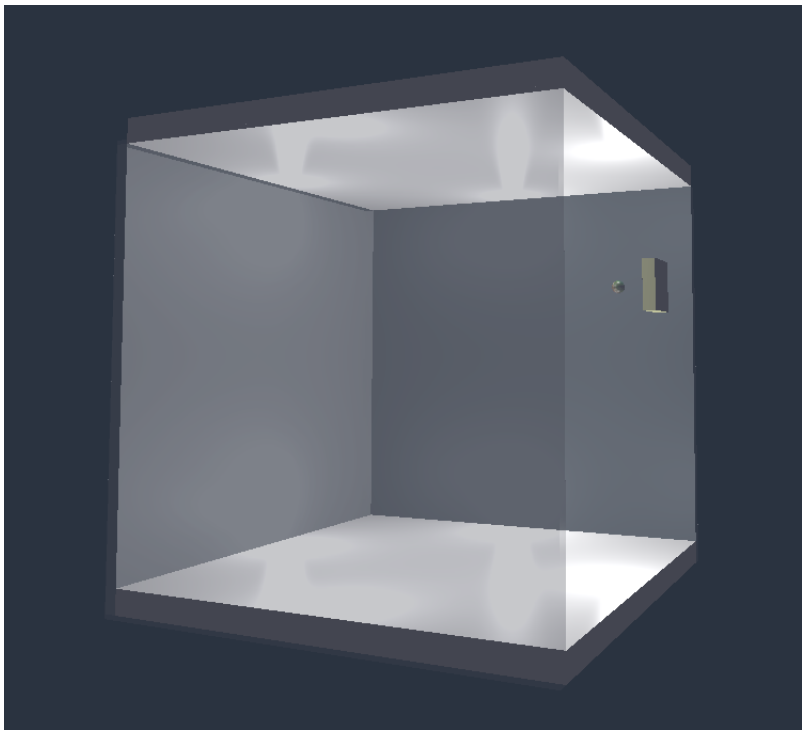

FIGURE 4.14: The racquetball environment used to test the physics engine parameter candidates' influence over the physics simulation.

Fourty-four candidate parameters were selected for testing in the racquetball environment (see Table 4.3). 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 appeared normal based on visual observation. 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 bounds 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 maximum 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 bounds 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 through five.

Experiments three through five had the exact same setup as experiment two, except for the selection method and whether or not self-adaptation was used. Experiment three used tournament selection without self-adaptation. Experiment four used rank fitness selection with self-adaptation. Lastly, experiment five used rank fitness selection without self-adaptation.

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 4.15 and Figure 4.16. The second method was an algorithm developed by the author, 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 maximum Euclidean distance between P and Q. See Figure 4.17. The third method was the discrete Fréchet

distance and the fourth method was the Hausdorff distance [24] [25].

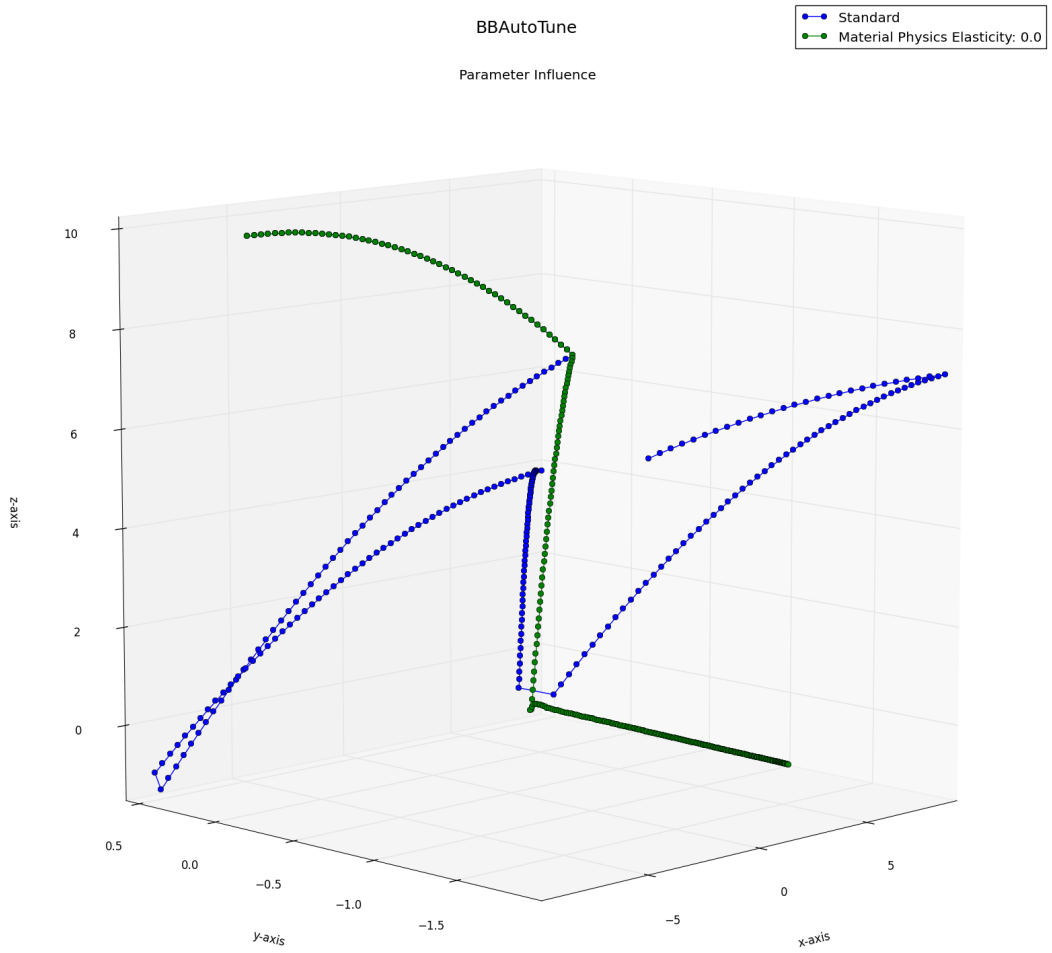

FIGURE 4.15: The dissimilarity of the ball paths between the tweaked, material-physics elasticity parameter and the standard (where no parameters were tweaked).

Twenty 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.

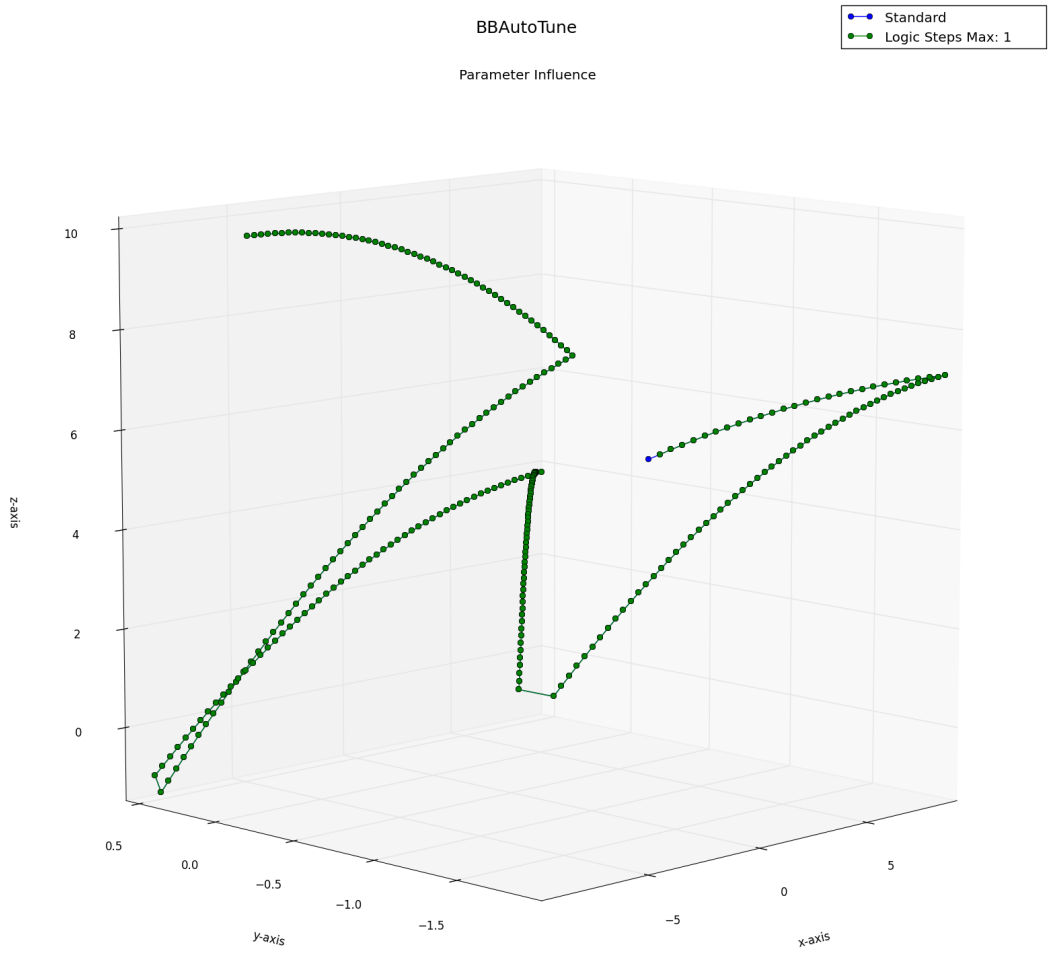

FIGURE 4.16: The similarity of the ball paths between the tweaked, logic-steps maximum parameter and the standard (where no parameters were tweaked).

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 Elasticity: 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
Use Actor: False	0.329232644023	0.329232644023	0.329232644023
Use 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
Mass: 10000.0	3.57800913743	3.30042073543	3.21534852885
Radius: 1cm	0.329232644023	0.329232644023	0.329232644023
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
Damping Frames: -32768	0.329232644023	0.329232644023	0.329232644023

TABLE 4.3: The distances between each tweaked-parameter ball path and the standard ball path for each of the three quantifiable measurement methods employed. The highlighted tweaked parameters were determined not to be influential.

```
BEGIN
    P = ⟨p_1, p_2, ..., p_n⟩
    Q = ⟨q_1, q_2, ..., q_m⟩
    max = 0.0
    For all p_i ∈ P and q_j ∈ Q do
        d = ||p_i − q_j||
        If max < d then
            max = d
        End if
    End for
    Return max
END
```

FIGURE 4.17: 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 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 (with the exception of the elites) which resulted in essentially a randomly generated population and thereby destroying the high fitness solutions previously found. Notice that after the probabilities flipped, the 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 4.18. 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.

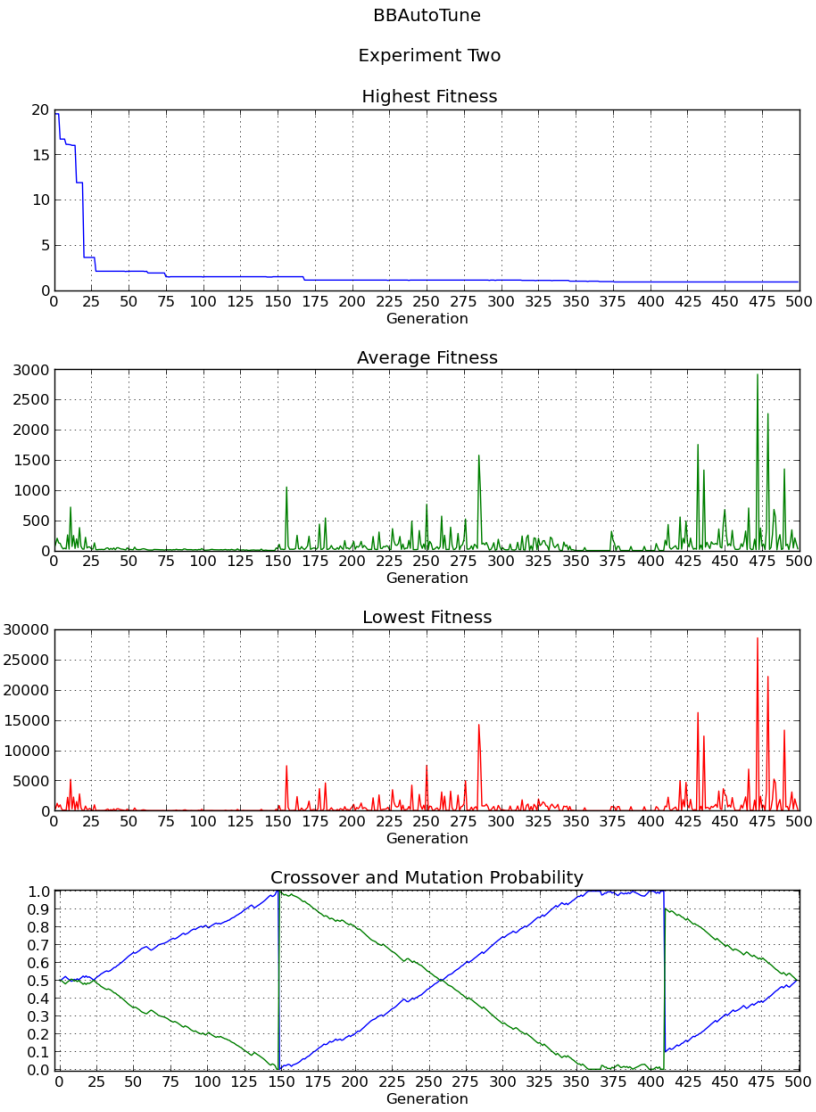

FIGURE 4.18: The highest, average, and lowest fitness in addition to the crossover and mutation probability over the course of 500 generations. For the bottom plot, the crossover probability is shown in blue while the mutation probability is shown in green.

	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 Maximum	900.11395466	647.638168514
Damping Translation	1.0	0.0
Damping Rotation	0.691143247902	0.648240602049
Collision Bounds Type	SPHERE	SPHERE
Torque	82.7271515601	100.0
Fitness	0.930619100106	28584.2244771

TABLE 4.4: 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 4.19. In contrast to experiment two, it would seem that the self-adaptation side effect of periodically randomizing the population helped experiment two obtain a better 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 Maximum	0.0	250.88084935511978
Damping Translation	1.0	0.33429608723965537
Damping Rotation	0.14340109658364997	0.0
Collision Bounds Type	TRIANGLE_MESH	SPHERE
Torque	47.85677413931377	63.68300968863001
Fitness	1.0827696957	1617.02428027

TABLE 4.5: 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.

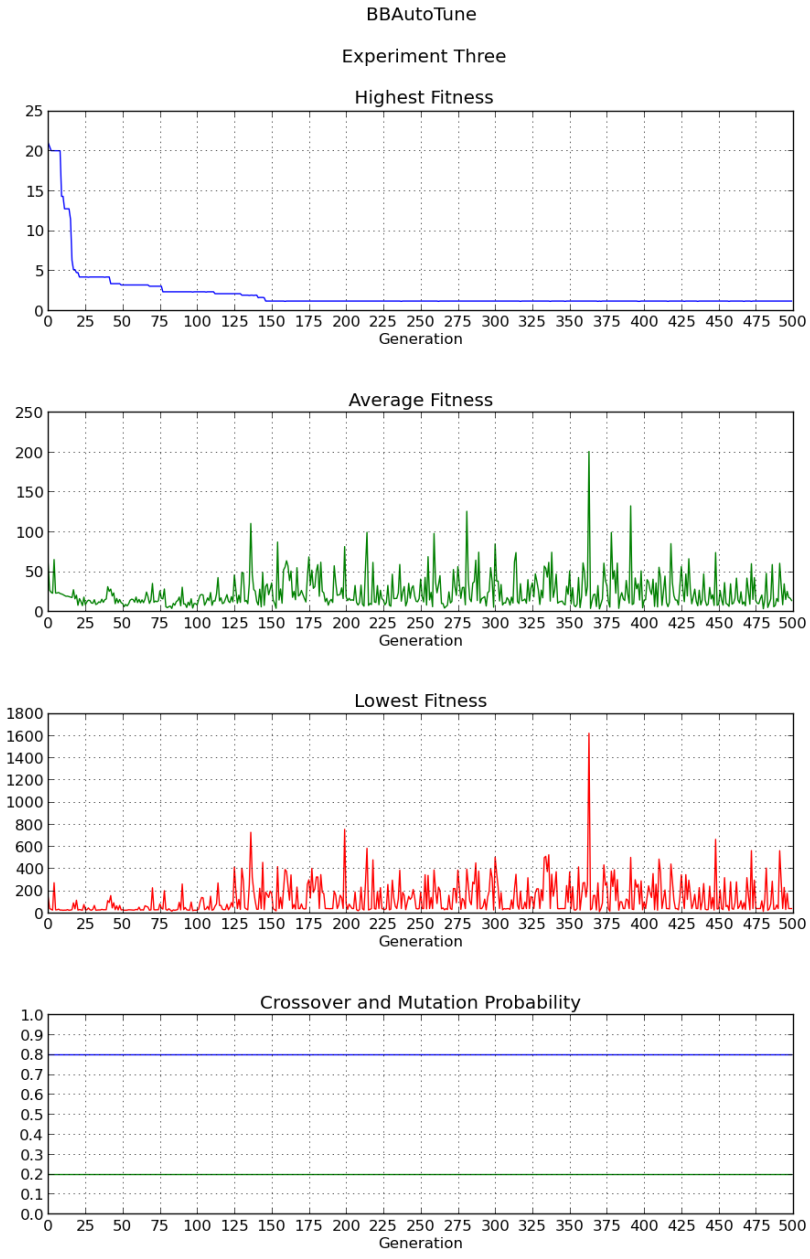

FIGURE 4.19: The highest, average, and lowest fitness in addition to the crossover and mutation probability over the course of 500 generations. For the bottom plot, the crossover probability is shown in blue while the mutation probability is shown in green.

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 4.20. 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 Maximum	1000.0	1000.0
Damping Translation	0.0	0.387362543282
Damping Rotation	0.984539067046	0.687852083808
Collision Bounds Type	SPHERE	CONVEX_HULL
Torque	6.70058480184	75.8262976514
Fitness	1.1309704845	16396.2145412

TABLE 4.6: 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.

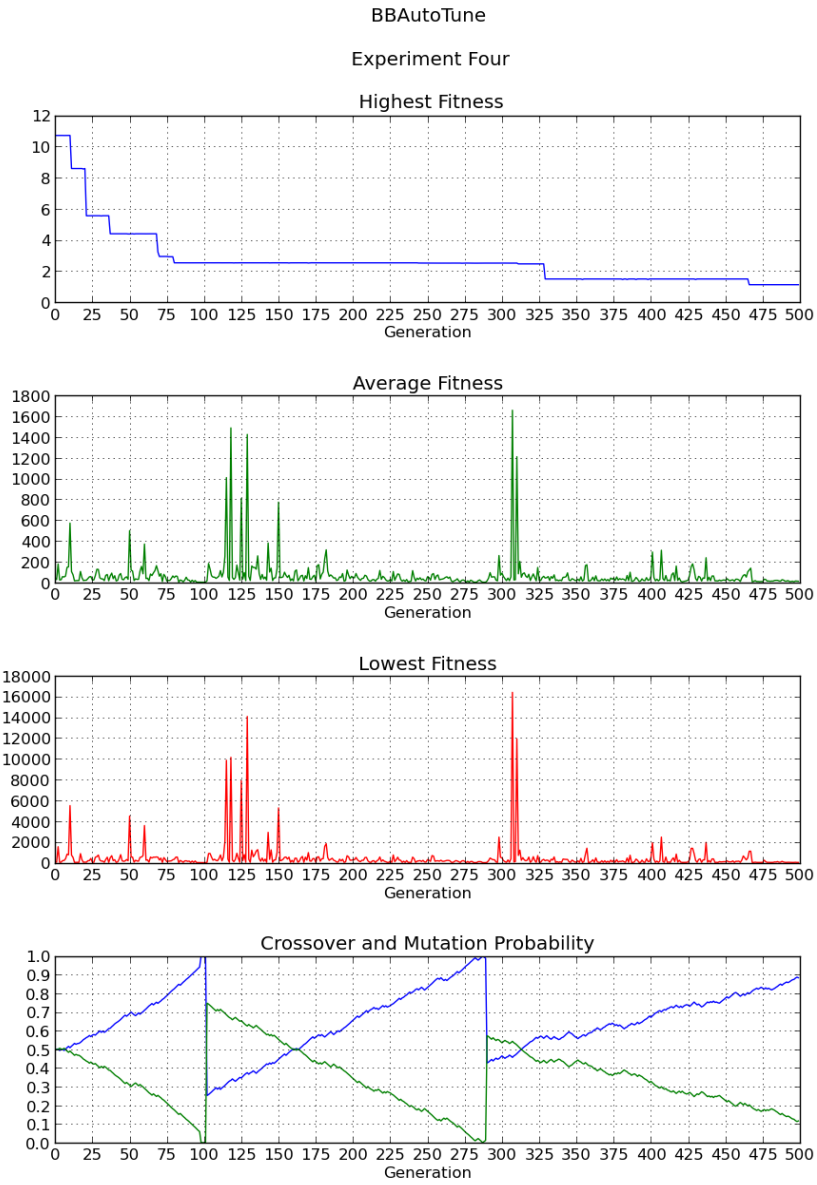

FIGURE 4.20: The highest, average, and lowest fitness in addition to the crossover and mutation probability over the course of 500 generations. For the bottom plot, the crossover probability is shown in blue while the mutation probability is shown in green.

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 4.21. 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 Maximum	660.0787581868401	787.7673658611162
Damping Translation	1.0	0.6703819812309364
Damping Rotation	0.4031179325185546	0.10076651150002103
Collision Bounds Type	SPHERE	SPHERE
Torque	46.465508081185064	49.46737364841879
Fitness	1.0638026764	3257.00654843

TABLE 4.7: 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.

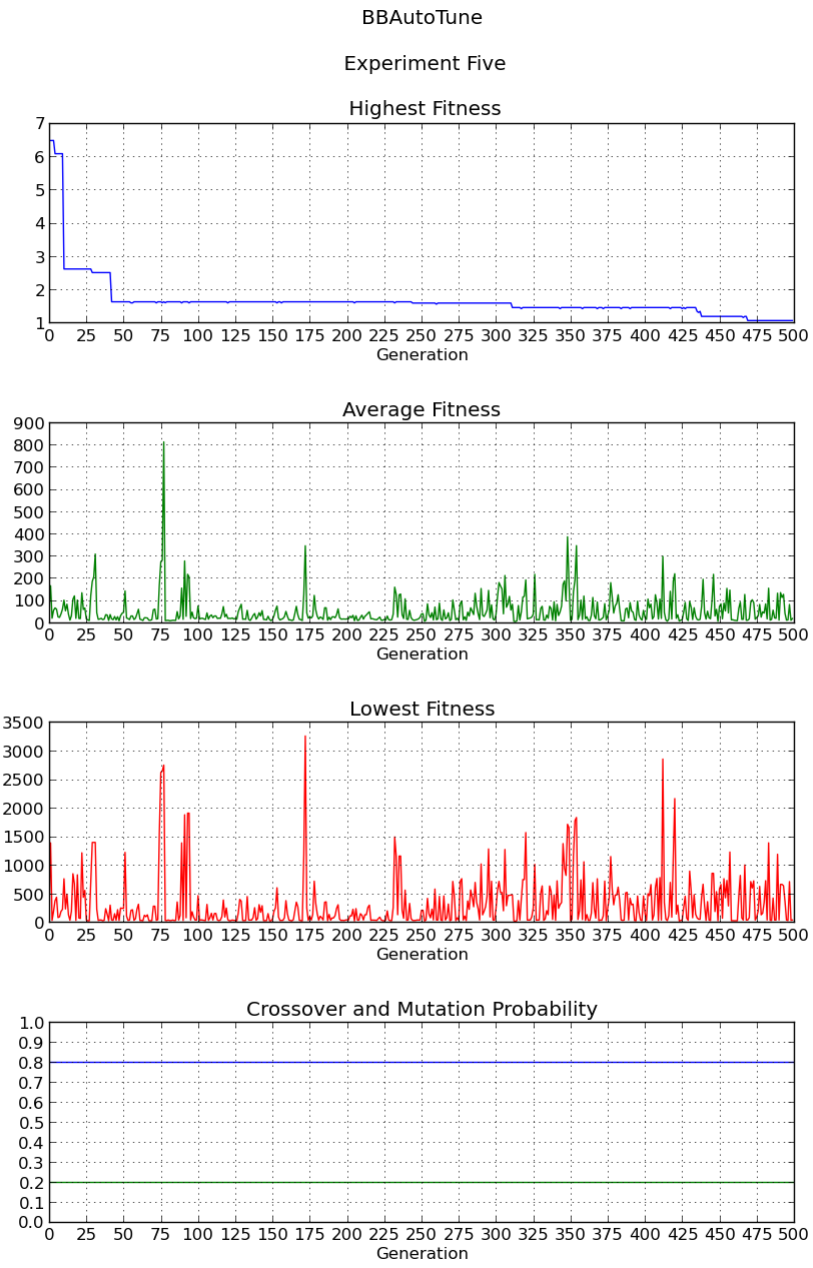

FIGURE 4.21: The highest, average, and lowest fitness in addition to the crossover and mutation probability over the course of 500 generations. For the bottom plot, the crossover probability is shown in blue while the mutation probability is shown in green.

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 4.22, and Figure 4.23.

	Final States of Highest Performing Phenotypes				
	Exp. Two	Exp. Three	Exp. Four	Exp. Five	Real Robot Robust Mean
X-position	23.12349975cm	23.95108938cm	23.49144816cm	24.05546605cm	23.9934044cm
Y-position	0.00953689cm	-0.34897618cm	-0.00286334cm	0.01641194cm	0.0351240cm
Heading (Z-orientation)	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: 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.

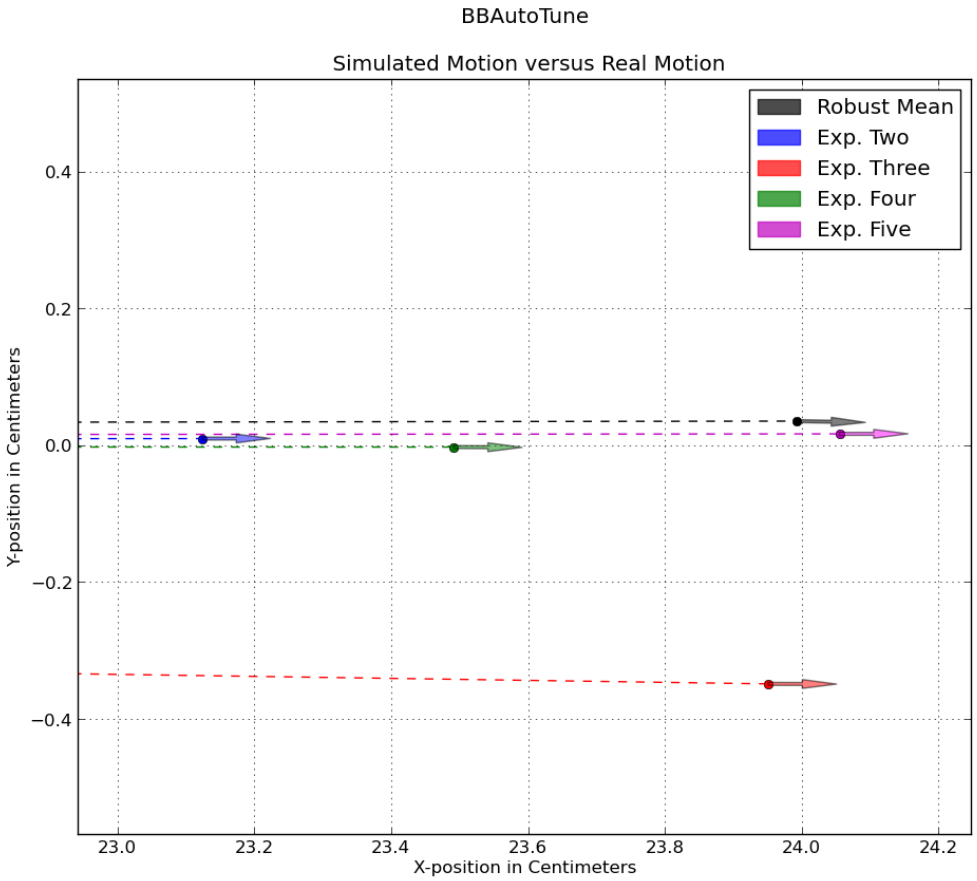

FIGURE 4.22: The final states of the top phenotypes from experiments two through five compared to the real robot robust mean.

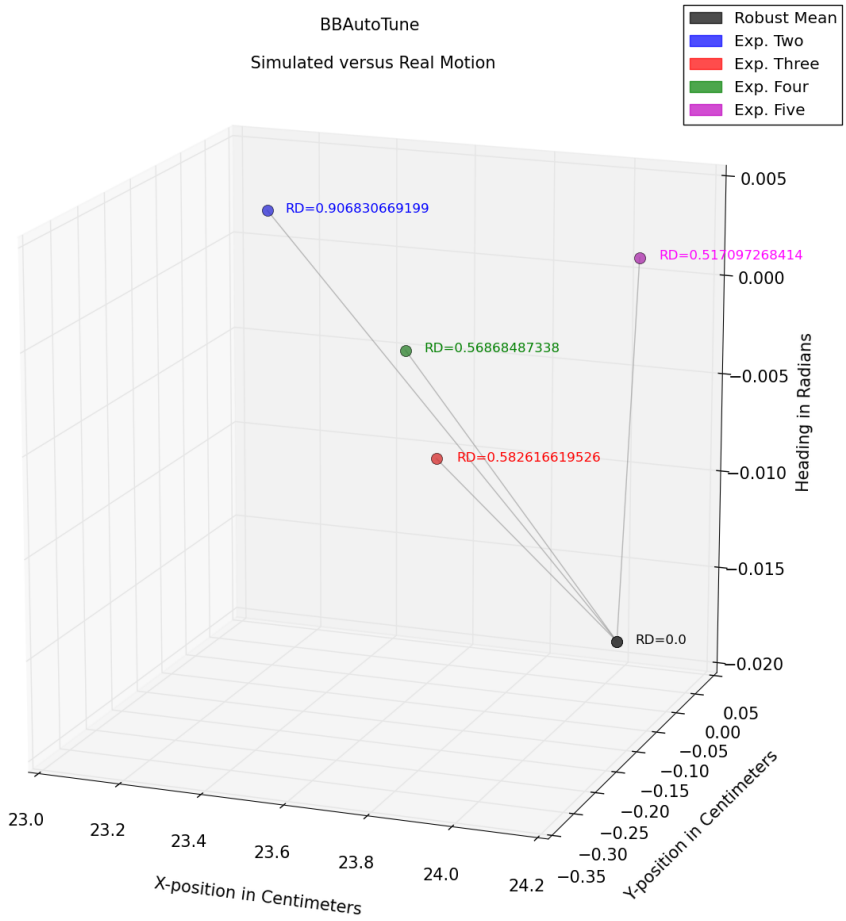

FIGURE 4.23: An alternate view of the final states of the top phenotypes from experiments two through five compared to the real robot robust mean. The robust distances are indicated near each data point.

Chapter 5

BlenderSim

5.1 Overview

Over the course of three months, a Blender based simulation was developed for HRTeam 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, in order to progress on the development of the BlenderSim 3D simulation environment.

As a proof of concept of the thesis, BlenderSim was revisited with the physics engine providing the motion model, using the best-fitness parameters learned by BBAutoTune. Using one robot, the motion of the simulated, physics-based robot was compared to the real robot by re-running a previously logged HRTeam experiment in BlenderSim.

5.2 Preliminary Work

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 [26]. Objects smaller than .05 (5cm) Blender/Bullet units, in any given dimension, erratically jitter despite having no force acting upon them.

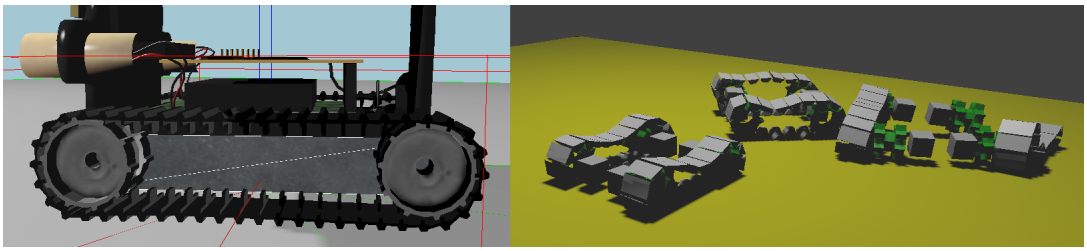

FIGURE 5.1: The to-scale treads modeled after the physical SRV-1 robot on the left and the physics-based rigid-body tracks in motion on the right.

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 default physics engine was not used to provide a 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 which only moves the 3D robot as if it were a single point body. See Figure 5.2 and Figure 5.3.

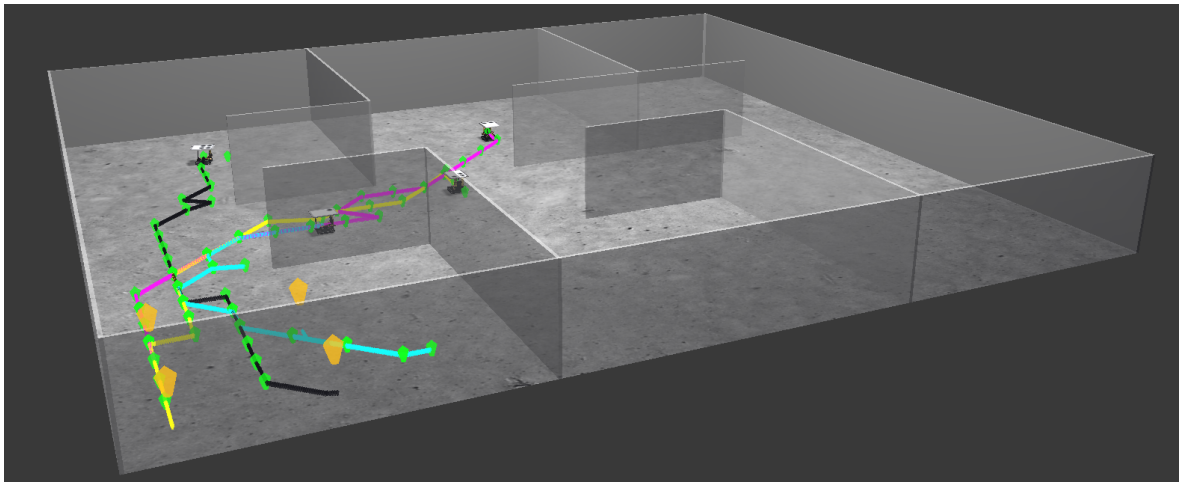

FIGURE 5.2: The constant velocity motion model in BlenderSim with four robots traversing their respective paths which consist of way-points logged in a previously recorded physical robot experiment.

5.3 Evaluation

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. The

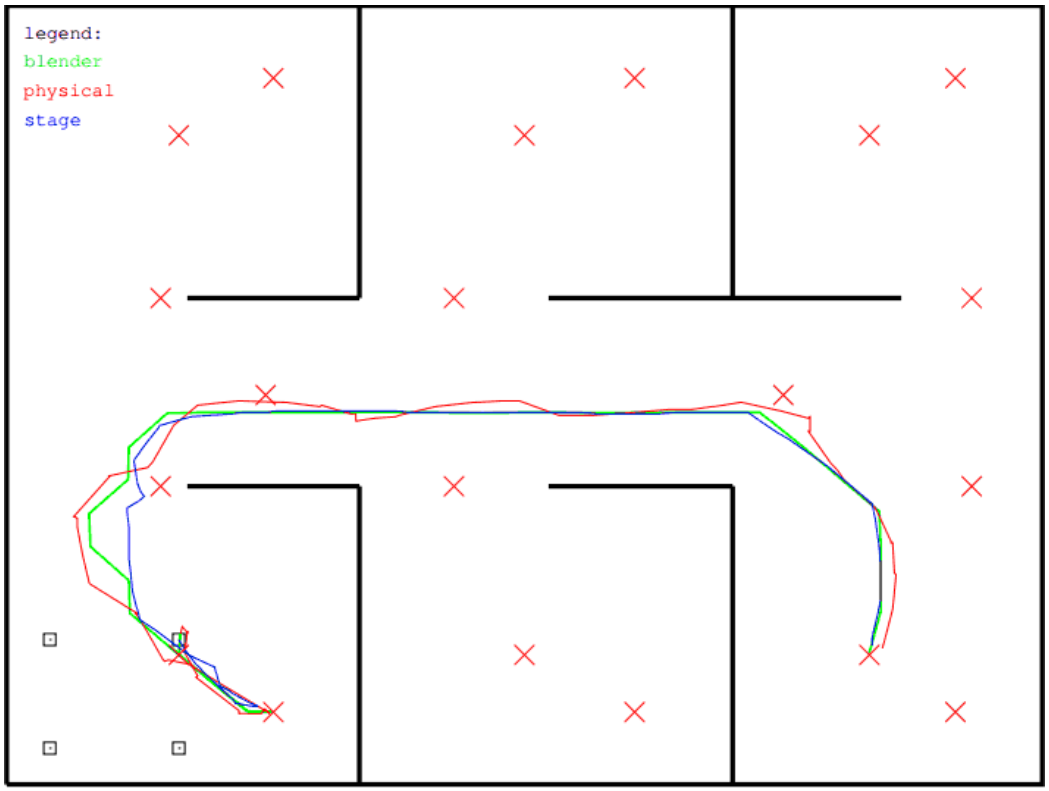

FIGURE 5.3: The path plots—as plotted by Dr. Elizabeth Sklar—of the BlenderSim robot, the physical robot, and the *Stage* (a 2D robot simulator already in use by HRTeam)[27] robot plotted in comparison.

purpose was to evaluate the efficacy of the physics parameters learned by BBAutoTune, when deployed in the simulated robot environment and measured in comparison with its physical robot counterpart.

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.

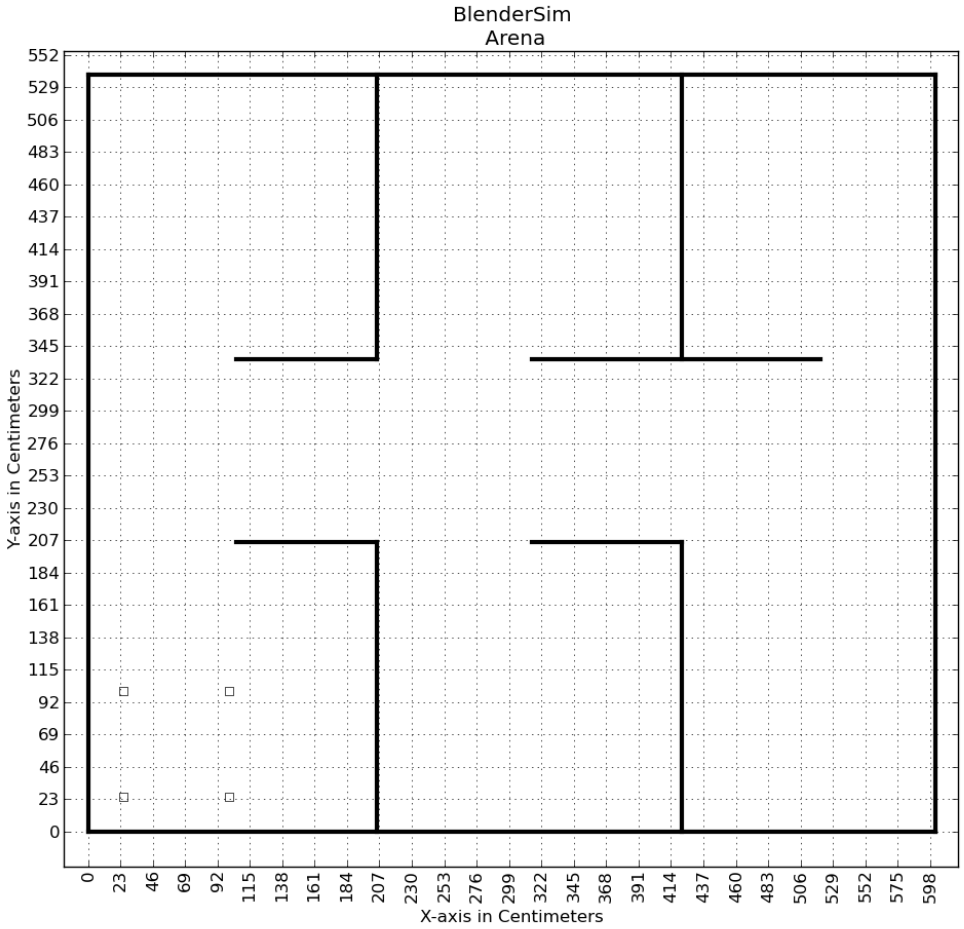

FIGURE 5.4: The layout and dimensions of the arena map both as it is in BlenderSim and in the real HRTeam lab. The smaller four squares in the lower left-hand corner represent the standard starting positions of the robots before any experiment. The grid shown in the plot is the same discrete grid used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

5.3.2 Surveyor SRV-1 Blackfin 3D Model

The physics parameters governing the wheels were the same best-fitness parameters found by BBAutoTune in subsection 4.5.2. Residing 1.37cm off of the arena floor from their local origins in the positive global z-axis, the robot's base and wheels had their z-translation fixed. This distance between the wheels and the floor is important since this was the same height used to the tune the physics engine parameters in BBAutoTune. Using any other height would result in different motion from the motion observed in section 4.6 for experiment two.

5.3.3 Robot Path Planner

Given a HRTeam experiment log and a robot number, the path planner extracts the waypoints that the real robot was instructed to go to during the HRTeam experiment. These waypoints come from an A* calculated path that runs from the robot's starting position to various points of interest or task points. With the waypoints 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 base of the SRV-1 3D model. See Figure 5.5. Contained in its logic is a path or waypoint queue that it runs through. This queue is populated by the 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. Recalling from subsection 4.5.2 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. Note that, the torque value ± 20.8 was chosen arbitrarily.

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 moving 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

HRTeam experiments have five predefined sets of "interest points" or task points that the robots visit as they travel around the arena [28]. BlenderSim imitates this using its task points manager. At the start of the simulation, the task points manager reads the five configurations from the task points configuration directory. Controls include keys ***a*** through ***e*** where each key corresponds to five possible task point configurations. See Figures 5.5 and 5.6.

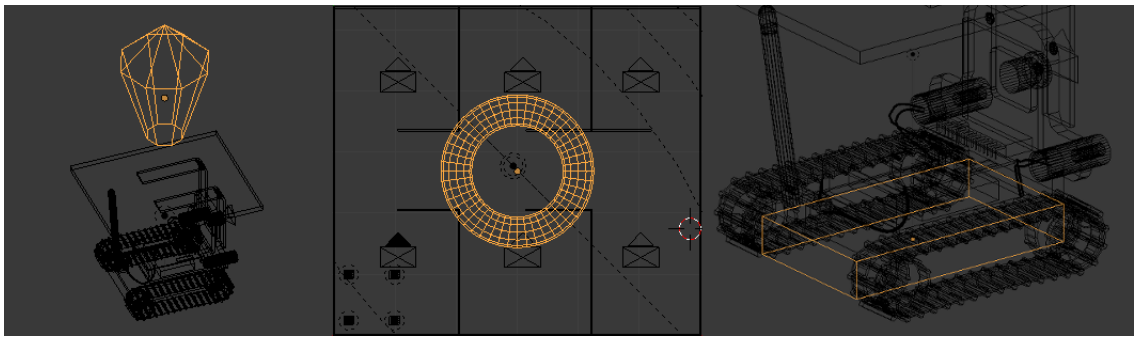

FIGURE 5.5: From left to right: the robot path planner, the task points manager, and the robot controller manifestations in the simulation.

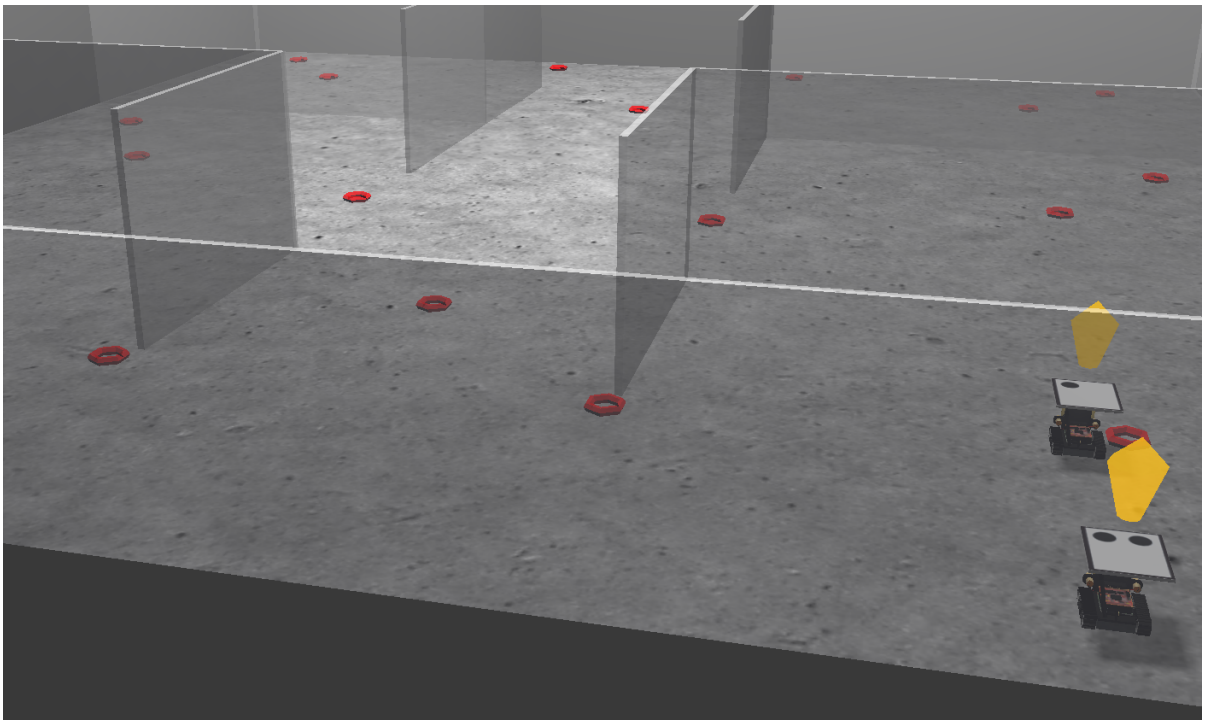

FIGURE 5.6: The red torus task points laid out in the arena.

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 best-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 5.7.

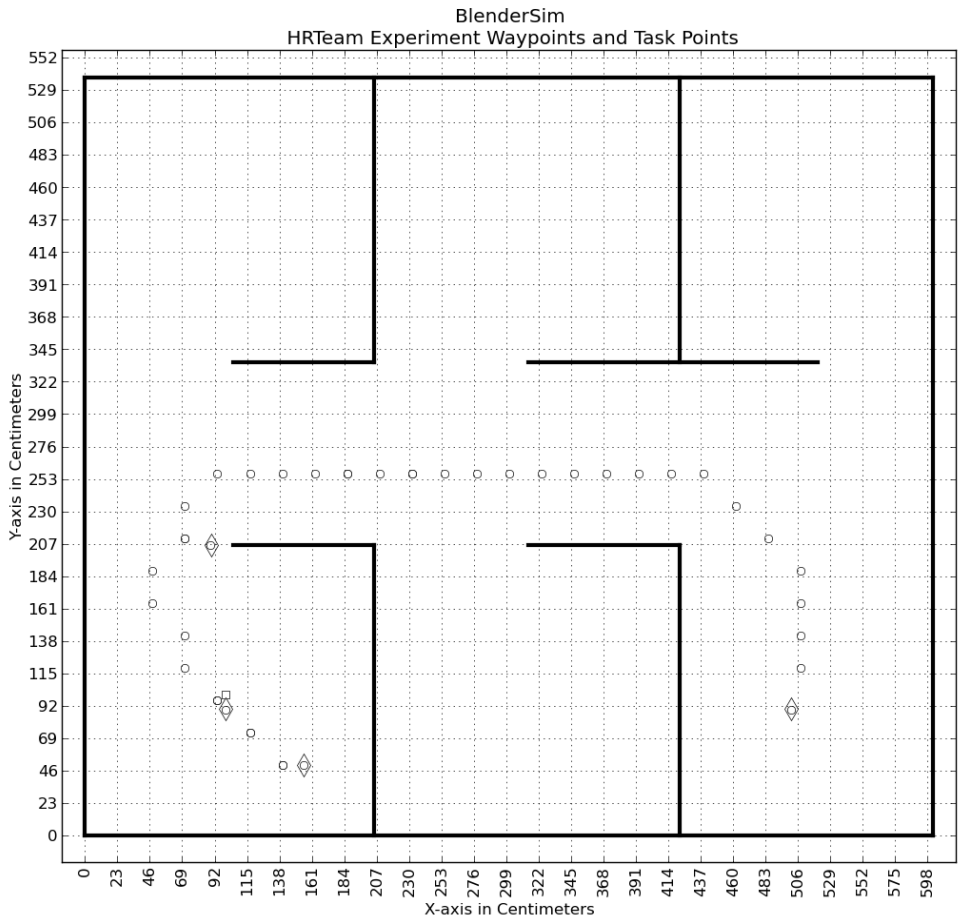

FIGURE 5.7: The arena layout with the experiment's waypoints and task points. The white square is the initial position of the robot before the experiment. The white circles are the waypoints and the white diamonds are the task points.

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[1]. See Tables 5.1, 5.2, and Figure 5.8.

	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: 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: The Hausdorff distances between the various paths.

[1]The waypoint path is the collection of waypoints that start at the robot's starting position and then travel to the four task points in the arena as seen in Figure 5.7.

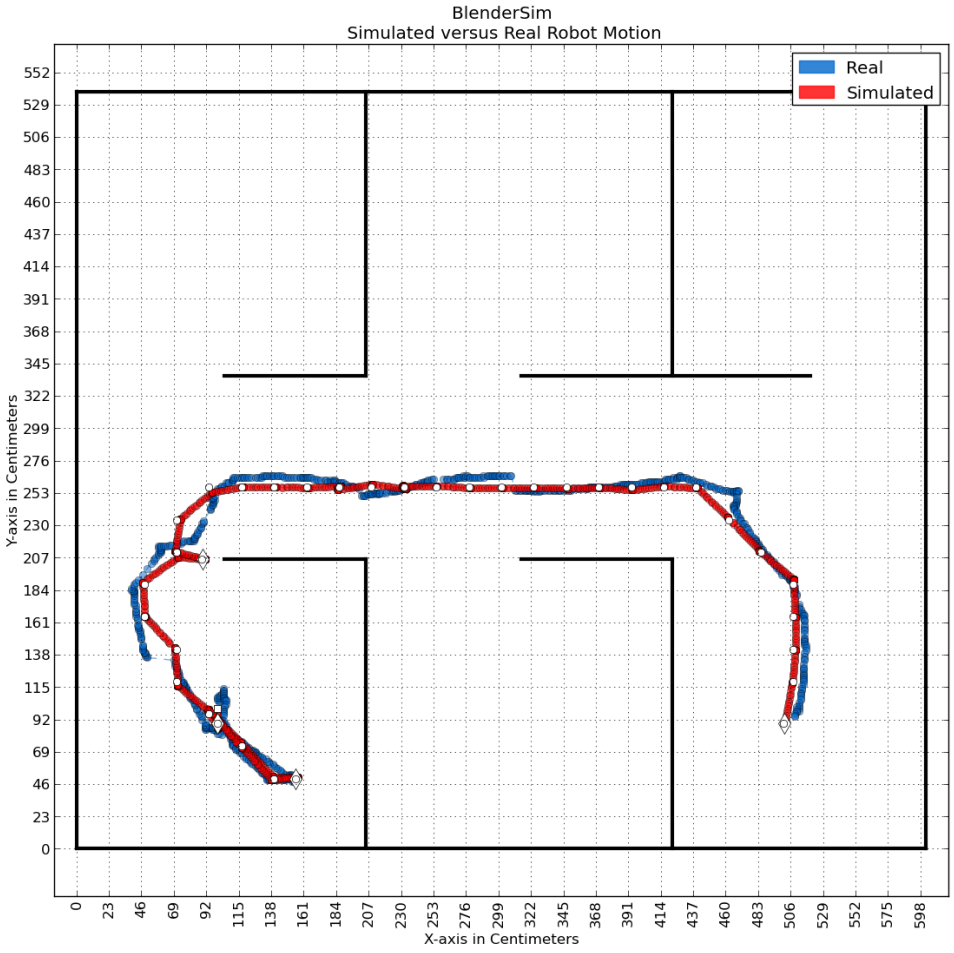

FIGURE 5.8: The simulated motion versus the real motion of the robot over the course of the experiment.

Chapter 6

Conclusion

6.1 Genetic Algorithms

The GA outlined in this thesis employed a wide variety of techniques presented by previous works. Unfortunately, there is no unanimously recognized theory of GAs [5]. Furthermore, there is no proven optimal set of GA parameters for any given problem being solved by a GA [4]. There are, however, a few schools of thought and generally accepted guidelines when developing a GA [4][5]. As to why GAs work, at least for BCGAs, a few ideas have been presented such as John Holland's schema theorem and David Goldberg's building block hypothesis [5].

6.2 SimPL

The EA developed for SimPL proved to be a robust basis for the EA 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 EA 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 an EA 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 search space size was $(2^{53})^{11} = 2^{53*11} = 2^{583} = 3.165829139 \times 10^{175}$ possible states[1]. It is even more impressive considering the many days lost trying to find reasonable parameters by hand (during preliminary work) 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 bound 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 any one time on the discrete arena grid is $\sim 32.52cm^2$ 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.

6.5 Future Work

Future work, concerning BBAutoTune and BlenderSim, could involve the following:

1. capture additional real-robot motion data on the SRV-1 by issuing it different commands (such as turning) instead of just forward;
   (a) take this additional motion data and in BBAutoTune, learn the physics parameters needed to simulate these different robot commands in BlenderSim;

[1] On a 64-bit computer architecture, there is approximately 2^{53} representable (double precision) floats between 0.0 and 1.0 [29][30]. Each genome in BBAutoTune contained an array of 11 floats which represented the 11 possible tunable physics parameters. The range of each float in the array was $[0.0, 1.0]$.

[2] Overlaid 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 $\sqrt{23^2cm + 23^2cm} = 32.526911935cm$.

2. run new (or other previously logged) experiments in tandem between HRTeam and BlenderSim, making further comparisons between simulated and real robot paths; and

3. in BBAutoTune, learn the physics parameters necessary to simulate the motion of other robots (other than the SRV-1).

Appendix A

Physics Engine Parameters Governing Forward Motion Learned

Table A.1 lists every physics engine parameter that governed over the best-fitness forward motion learned in subsection 4.5.2. Some of the parameters only related to the physics engine as a whole, thus affect everything in simulation, while the other parameters effected only the robot's base, the robot's wheels, or just the floor. The highlighted values indicate the 11 GA tuned/learned physics engine parameters.

Parameter	Physics Engine	Wheels	Base	Floor
Gravity	2.89862416489 $\frac{m}{s^2}$	N/A	N/A	N/A
Max Steps	5	N/A	N/A	N/A
Sub-steps	5	N/A	N/A	N/A
FPS	30	N/A	N/A	N/A
Linear Deactivation Threshold	5.0	N/A	N/A	N/A
Angular Deactivation Threshold	5.0	N/A	N/A	N/A
Deactivation Time	0.5sec	N/A	N/A	N/A
Use Material Physics	N/A	True	True	True
Material Friction	N/A	59.5011814113	1.0	1.0
Material Elasticity	N/A	0.0742521056997	0.0	0.0
Material Force	N/A	0.0	0.0	0.0
Material Damping	N/A	0.0	0.0	0.0
Material Distance	N/A	0.0	0.0	0.0
Material Align to Normal	N/A	False	False	False
Physics Type	N/A	RIGID_BODY	RIGID_BODY	STATIC
Use Actor	N/A	True	True	False
Use Ghost	N/A	False	False	False
Use Material Force Field	N/A	False	False	N/A
Rotate From Normal	N/A	False	False	N/A
No Sleeping	N/A	False	False	N/A
Mass	N/A	4.33392950881	9.373	N/A
Radius	N/A	1.4cm	1cm	1cm
Form Factor	N/A	1.0	1.0	N/A
Use Anisotropic Friction	N/A	True	False	False
Anisotropic Friction X,Y,Z	N/A	0.0,1.0,1.0	N/A	N/A
Velocity Minimum	N/A	0.0	0.0	N/A
Velocity Maximum	N/A	900.11395466	0.0	N/A
Damping Translation	N/A	1.0	0.0	N/A
Damping Rotation	N/A	0.691143247902	0.0	N/A
Lock Translation X,Y,Z	N/A	False,False,True	False,False,True	N/A
Lock Rotation X,Y,Z	N/A	False,False,False	True,True,False	N/A
Use Collision Bounds	N/A	True	True	True
Collision Bounds Type	N/A	SPHERE	BOX	BOX
Collision Bounds Margin	N/A	0.0m	0.0m	0.0m
Force	N/A	0.0	N/A	N/A
Torque	N/A	82.7271515601	N/A	N/A
Linear Velocity	N/A	0.0	N/A	N/A
Angular Velocity	N/A	0.0	N/A	N/A
Use Local Force	N/A	N/A	N/A	N/A
Use Local Torque	N/A	True	N/A	N/A
Use Local Linear Velocity	N/A	N/A	N/A	N/A
Use Local Angular Velocity	N/A	N/A	N/A	N/A
Damping Frames	N/A	0	N/A	N/A

TABLE A.1: The physics engine parameters governing the best-fitness forward motion learned in subsection 4.5.2. The highlighted cells indicate the GA tuned/learned values.

Appendix B

Source Code

B.1 SimPL

Listing B.1: Index.js

```
/*
 *
 * David Lettier (C) 2013.
 *
 * http://www.lettier.com/
 *
 * Simple Pong Learner or SimPL.
 *
 * Single sided pong clone with only one paddle which learns to follow and collide with the ball.
 *
 */

var request_animation_id = null;

var take_control = false;
var pause        = true;
var show_debug   = false;

var use_neural_network_paddle = true;
var use_perfect_paddle        = false;
var use_random_paddle         = false;

var run_experiment = true;

var top_wall    = new Static_Object( "top_wall"    );
var right_wall  = new Static_Object( "right_wall"  );
var bottom_wall = new Static_Object( "bottom_wall" );
var left_wall   = new Static_Object( "left_wall"   );

var ball_slot   = document.getElementById( "ball_slot" );
var paddle_slot = document.getElementById( "paddle_slot" );

var ball_reset_center   = { x: ball_slot.offsetLeft      + ( ( ball_slot.offsetWidth    || ball_slot.
      clientWidth    ) / 2 ),
                            y: ball_slot.offsetTop     + ( ( ball_slot.offsetHeight   || ball_slot.
      clientHeight   ) / 2 )  };
var paddle_reset_center = { x: paddle_slot.offsetLeft    + ( ( paddle_slot.offsetWidth  || paddle_slot.
      clientWidth  ) / 2 ),
                            y: paddle_slot.offsetTop  + ( ( paddle_slot.offsetHeight || paddle_slot.
      clientHeight ) / 2 )  };

var paddle_path = document.getElementById( "paddle_path" );
var ball_in_paddle_path_color     = "rgba( 200, 255, 136, .2 )";
var ball_not_in_paddle_path_color = "rgba( 255,  98,  98, .2 )";
var padddle_path_background_color = ball_in_paddle_path_color;

var random_angle_range = { min: 135, max: 225 };
var random_angle = random_angle_range.min + ( random_angle_range.max - random_angle_range.min ) * Math.
      random( );

var starting_magnitude        = 1000.0;
var starting_ball_magnitude   = starting_magnitude;
var starting_paddle_magnitude = starting_magnitude;
var starting_paddle_angle = 90.0;

var ball   = new Physics_Object(  new Dynamic_Object( "ball" ), random_angle, starting_ball_magnitude );

var paddle = new Physics_Object(  new Dynamic_Object( "paddle", [ paddle_path, "t" ] ),
      starting_paddle_angle, starting_paddle_magnitude );

var physics_engine = new Physics_Engine( collision_handler, paddle, ball, top_wall, right_wall,
      bottom_wall, left_wall );

var time_delta = 0.0;
```

```
var fps_monitor = new FPS_Monitor( );

var debug_manager = new Debug_Manager( "debug" );

var database_manager = new Database_Manager( );

var neural_net_parameters = {

  nInputs:                6,
  nOutputs:               1,
  nHiddenLayers:          1,
  nNeuronsPerHiddenLayer: 5,
  bias:                  -1

};

var genetic_algorithm_parameters = {

  popSize:                                10,
  nGenesPerGenome:                      null,
  useRankFitness:                       true,
  useSelfAdaptation:                    true,
  pCrMuSeq:                            false,
  iCProb:                                0.8,
  iMProb:                                0.2,
  nElite:                                  2

};

var learner_parameters = {

  neural_net:               neural_net_parameters,
  genetic_algorithm: genetic_algorithm_parameters

};

var learner = {

  parameters:                learner_parameters,
  neural_net:                              null,
  neural_net_output:                       null,
  genetic_algorithm:                       null,
  evaluation_start_time:                      0,
  evaluation_end_time:                        0,
  evaluate_current_genome:                false,
  current_genome_fitness_tracking: new Array( ),
  current_genome_paddle_hits:                 0,
  current_genome_being_evalutated:            0

};

learner.neural_net = new Neural_Net( learner.parameters.neural_net );

learner.parameters.genetic_algorithm.nGenesPerGenome = learner.neural_net.get_number_of_weights( );

learner.genetic_algorithm = new Genetic_Algorithm( learner.parameters.genetic_algorithm );

learner.neural_net_output = 0;

// Attempt to load in the last generation from the database.

var last_generation_data_string  = "population_size=" + learner.genetic_algorithm.get_population_size( );
last_generation_data_string     += "&number_of_genes_per_genome=" + learner.genetic_algorithm.
      get_number_of_genes_per_genome( );

database_manager.send_and_receive( "assets/scripts/retrieve_genomes.php", last_generation_data_string,
      database_manager, "last_generation", false );

var last_generation = database_manager.responses[ "last_generation" ];

if ( last_generation.charAt( 0 ) != ";" )
{

  if ( window.confirm( "Start from previous generation?" ) )
  {

    // String received from server is generation_number;crossover_probability;mutation_probability;
       population_genes.

    var generation_number = parseInt( last_generation.split( ";" )[ 0 ] );

    learner.genetic_algorithm.set_generation_number( generation_number );

    var crossover_probability = parseFloat( last_generation.split( ";" )[ 1 ] );

    learner.genetic_algorithm.set_crossover_probability( crossover_probability);

    var mutation_probability  = parseFloat( last_generation.split( ";" )[ 2 ] );

    learner.genetic_algorithm.set_mutation_probability( mutation_probability );

    var population_genes  = last_generation.split( ";" )[ 3 ];

    population_genes = population_genes.split( "," );

    for ( var i = 0; i < population_genes.length; ++i )
    {

      population_genes[ i ] = parseFloat( population_genes[ i ] );

    }

```

```
      learner.genetic_algorithm.replace_population_genes( population_genes );

      // Set neural network weights to the first genome's genes.

      learner.neural_net.put_weights( learner.genetic_algorithm.get_genome_genes( learner.
        current_genome_being_evalutated ) );

      pause = false;

    }
    else
    {

      learner.neural_net.put_weights( learner.genetic_algorithm.get_genome_genes( learner.
        current_genome_being_evalutated ) );

      pause = false;

    }

  }
  else
  {

    learner.neural_net.put_weights( learner.genetic_algorithm.get_genome_genes( learner.
        current_genome_being_evalutated ) );

    pause = false;

  }

  var ball_style = window.getComputedStyle( ball.dynamic_object.object, null);
  var ball_style_transform =    ball_style.getPropertyValue( "-webkit-transform" ) ||
              ball_style.getPropertyValue( "-moz-transform"    ) ||
              ball_style.getPropertyValue( "-ms-transform"     ) ||
              ball_style.getPropertyValue( "-o-transform"      ) ||
              ball_style.getPropertyValue( "transform"         );
  var ball_div_transform_angle    = 360;
  var ball_div_transform_angle_by = 20;

  function draw( time_stamp )
  {

    request_animation_id = window.requestAnimationFrame( draw );

    time_delta = fps_monitor.get_time_delta( time_stamp );

    if ( !pause )
    {

      debug_manager.add_or_update( "FPS", fps_monitor.get_fps( time_stamp ) );

      debug_manager.add_or_update( "Paddle", paddle );

      debug_manager.add_or_update( "Ball", ball );

      handle_neural_network_ouput( );

      debug_manager.add_or_update( "Current Genome", learner.current_genome_being_evalutated );

      debug_manager.add_or_update( "NN", learner.neural_net );

      debug_manager.add_or_update( "NN Weights", learner.neural_net.get_weights( ) );

      debug_manager.add_or_update( "GA", learner.genetic_algorithm );

      physics_engine.update( time_delta );

      // Spin the ball div to give the illusion of flying through the air.

      if ( ball_style_transform != undefined && ball.get_magnitude( ) != 0 )
      {

        ball.dynamic_object.object.style.webkitTransform = "rotate(" + ball_div_transform_angle + "deg)";
        ball.dynamic_object.object.style.MozTransform    = "rotate(" + ball_div_transform_angle + "deg)";
        ball.dynamic_object.object.style.msTransform     = "rotate(" + ball_div_transform_angle + "deg)";
        ball.dynamic_object.object.style.OTransform      = "rotate(" + ball_div_transform_angle + "deg)";
        ball.dynamic_object.object.style.transform       = "rotate(" + ball_div_transform_angle + "deg)";

        if ( ball.get_angle( ) >= 90 && ball.get_angle( ) <= 270 )
        {

          // Spin counter clock-wise if it is going towards the left of the screen.

          ball_div_transform_angle = ball_div_transform_angle < 0 ? 360 : ball_div_transform_angle - (
        ball_div_transform_angle_by * ( ball.get_magnitude( ) / starting_ball_magnitude ) );

        }
        else
        {

          ball_div_transform_angle = ball_div_transform_angle > 360 ? 0 : ball_div_transform_angle + (
        ball_div_transform_angle_by * ( ball.get_magnitude( ) / starting_ball_magnitude ) );

        }

      }

      // Time to evaluate the current genome?

      if ( learner.evaluate_current_genome )
      {
```

```
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
  ( function ( )
269 {
    var lastTime = 0;
271 var vendors = ['ms', 'moz', 'webkit', 'o'];
    for( var x = 0; x < vendors.length && !window.requestAnimationFrame; ++x )
273 {
275   window.requestAnimationFrame = window[ vendors[ x ] + 'RequestAnimationFrame' ];
      window.cancelAnimationFrame = window[vendors[x]+'CancelAnimationFrame'] || window[ vendors[ x ]+'
        CancelRequestAnimationFrame' ];
277
    }
279
    if ( !window.requestAnimationFrame )
281 {
283   window.requestAnimationFrame = function ( callback, element )
      {
285
        var currTime = new Date( ).getTime( );
287     var timeToCall = Math.max( 0, 16 - ( currTime - lastTime ) );
        var id = window.setTimeout( function ( ) { callback( currTime + timeToCall ); }, timeToCall );
289     lastTime = currTime + timeToCall;
        return id;
291
      };
293
    }
295
    if ( !window.cancelAnimationFrame )
297 {
299   window.cancelAnimationFrame = function ( id )
      {
301
        clearTimeout( id );
303
      };
305
    }
307 } ( ) );

309 draw( );

311 function reset( )
    {
313
      random_angle = random_angle_range.min + ( random_angle_range.max - random_angle_range.min ) * Math.
          random( );
315   ball.dynamic_object.set_center( ball_reset_center.x, ball_reset_center.y );
      ball.set_magnitude( starting_ball_magnitude );
317   ball.set_angle( random_angle );

319   paddle.dynamic_object.set_center( paddle_reset_center.x, paddle_reset_center.y );
      paddle.set_magnitude( starting_ball_magnitude );
321   paddle.set_angle( starting_paddle_angle );

323   take_control = false;
      pause        = false;
325
    }
327
  document.onkeyup = handle_key;
329
  function handle_key( kevent )
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( );
        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-
          size' ), 10 );
351
        if ( window.event.shiftKey )
353     {
```

```
        debug_div_font_size -= 1;

        debug_div.style.fontSize = debug_div_font_size + "px";

      }
      else
      {

        debug_div_font_size += 1;

        debug_div.style.fontSize = debug_div_font_size + "px";

      }

      break;

    case 77: // [m] key.

      ball.set_magnitude( ball.get_magnitude( ) + 100 );

      break;

    case 80: // [p] key.

      pause = !pause;

      break;

    case 82: // [r] key.

      reset( );

      break;

    case 84: // [t] key.

      take_control = !take_control;

      break;

  }
}

document.onmousemove = handle_mouse_move;

function handle_mouse_move( mevent )
{

  if ( !take_control ) return null;

  mevent = ( window.event ) ? window.event : mevent;

  if ( ( mevent.clientY - ( paddle.dynamic_object.get_height( ) / 2 ) ) >= ( window.innerHeight - paddle.
      dynamic_object.get_height( ) ) )
  {

    paddle.dynamic_object.set_left_top( paddle.dynamic_object.get_left( ), window.innerHeight - paddle.
      dynamic_object.get_height( ) );

    return null;

  }
  else if ( ( mevent.clientY - ( paddle.dynamic_object.get_height( ) / 2 ) ) <= 0 )
  {

    paddle.dynamic_object.set_left_top( paddle.dynamic_object.get_left( ), 0 );

    return null;

  }

  paddle.dynamic_object.set_left_top( paddle.dynamic_object.get_left( ), mevent.clientY - ( paddle.
      dynamic_object.get_height( ) / 2 ) );

}

function handle_neural_network_ouput( )
{

  if ( take_control )
  {

    paddle.set_magnitude( 0 );

    paddle.set_angle( starting_paddle_angle );

    return null;

  }

  // To help with fitness testing, keep track of if the ball
  // is in the path of the paddle.
  // 1 for in the path of the ball.
  // Nothing for not in the path of the ball.
  //
  // In the path:
  //  ___________
  // |          0
  // |
  // |___________
  //
```

```
453   // Not in the path:
      //
455   //     0
      //  -----------
457   // |
      // |
459   // |-----------
      //
461   //
      //  -----------
463   // |
      // |
465   // |-----------
      //   0
467
469   if ( !( ( ball.dynamic_object.get_top( ) > paddle.dynamic_object.get_bottom( ) ) || ( ball.
          dynamic_object.get_bottom( ) < paddle.dynamic_object.get_top( ) ) ) )
      {
471
        learner.current_genome_fitness_tracking.push( 1 );
473
        // Adjust paddle path visual.
475
        paddle_path.style.background = ball_in_paddle_path_color;
477
479   }
      else
481   {
483     // Add nothing.
485     // Adjust paddle path visual.
487     paddle_path.style.background = ball_not_in_paddle_path_color;
489   }
491   /*
493   // Experiment one tracking.
495   learner.current_genome_fitness_tracking.push( Math.abs( paddle.dynamic_object.get_center( ).y - ball.
          dynamic_object.get_center( ).y ) );
497   */
499   // Get input values for the neural network.
501   var x = ball.dynamic_object.get_center( ).x - paddle.dynamic_object.get_center( ).x;
      var y = ball.dynamic_object.get_center( ).y - paddle.dynamic_object.get_center( ).y;
503
      var paddle_offset_from_ball = { x: x,
505                 y: y,
                   h: Math.sqrt( ( x * x ) + ( y * y ) )
507   };
509   x = paddle.dynamic_object.get_center( ).x - 0;
      y = paddle.dynamic_object.get_center( ).y - 0;
511
      var paddle_offset_from_screen_origin = { x: x,
513                    y: y,
                                               h: Math.sqrt( ( x * x ) + ( y * y ) )
515   };
517   var ball_velocity = deep_copy( ball.get_velocity( ) );
519   // Normalize input in [-1,1].
521   paddle_offset_from_ball.x = paddle_offset_from_ball.x / paddle_offset_from_ball.h;
      paddle_offset_from_ball.y = paddle_offset_from_ball.y / paddle_offset_from_ball.h;
523
      paddle_offset_from_screen_origin.x = paddle_offset_from_screen_origin.x /
          paddle_offset_from_screen_origin.h;
525   paddle_offset_from_screen_origin.y = paddle_offset_from_screen_origin.y /
          paddle_offset_from_screen_origin.h;
527   // Turn the velocity into unit vectors.
529   ball_velocity.x = ball_velocity.x / ball.get_magnitude( );
      ball_velocity.y = ball_velocity.y / ball.get_magnitude( );
531
      // Send in the input and get out the output which is an array.
533
      learner.neural_net_output = learner.neural_net.update( [ paddle_offset_from_ball.x,
          paddle_offset_from_ball.y, ball_velocity.x, ball_velocity.y, paddle_offset_from_screen_origin.x,
          paddle_offset_from_screen_origin.y ] );
535
      debug_manager.add_or_update( "NN Input", paddle_offset_from_ball.x.toFixed( 3 ) + " " +
          paddle_offset_from_ball.y.toFixed( 3 ) + " " + ball_velocity.x.toFixed( 3 ) + " " + ball_velocity.y.
          toFixed( 3 ) + " " + paddle_offset_from_screen_origin.x.toFixed( 3 ) + " " +
          paddle_offset_from_screen_origin.y.toFixed( 3 ) );
537
      debug_manager.add_or_update( "NN Output", learner.neural_net_output );
539
      if ( use_neural_network_paddle )
541   {
543     if ( learner.neural_net_output[ 0 ] < 0.0 )
        {
545
```

```
      // Output indicates that we must go down the screen
      // by some magnitude in the range (0,MAG_MAX].

      // Don't set a negative magnitude so multiply the output by -1 to
      // make it positive and multiply this result by the starting paddle
      // magnitude.

      paddle.set_magnitude( starting_paddle_magnitude * ( -1 * learner.neural_net_output[ 0 ] ) );

      paddle.set_angle( 270 );

    }
    else if ( learner.neural_net_output[ 0 ] == 0.0 )
    {

      // Don't move.

      paddle.set_magnitude( 0.0 );

      paddle.set_angle( starting_paddle_angle );

    }
    else if ( learner.neural_net_output[ 0 ] > 0.0 )
    {

      // Go up by some portion of the starting paddle magnitude.

      paddle.set_magnitude( starting_paddle_magnitude * learner.neural_net_output[ 0 ] );

      paddle.set_angle( 90.0 );

    }

  }
  else if ( use_perfect_paddle )
  {

    /*

    var difference = paddle.dynamic_object.get_center( ).y - ball.dynamic_object.get_center( ).y;

    debug_manager.add_or_update( "Perfect Output", difference );

    if ( difference < 0 )
    {

      paddle.set_magnitude( 10.0 * ( -1 * difference ) );

      paddle.set_angle( 270 );

    }
    else if ( difference == 0.0 )
    {

      paddle.set_magnitude( 0.0 );

      paddle.set_angle( starting_paddle_angle );

    }
    else if ( difference > 0 )
    {

      paddle.set_magnitude( 10.0 * difference );

      paddle.set_angle( 90.0 );

    }

    */

    paddle.set_magnitude( 0.0 );

    paddle.set_angle( starting_paddle_angle );

    var ball_center   = ball.dynamic_object.get_center( );

    var paddle_center = paddle.dynamic_object.get_center( );

    paddle.dynamic_object.set_center( paddle_center.x, ball_center.y );

  }
  else if ( use_random_paddle )
  {

    var random_float = get_random_float( -1, 1 );

    debug_manager.add_or_update( "Random Output", random_float );

    if ( random_float < 0.0 )
    {

      paddle.set_magnitude( starting_paddle_magnitude * ( -1 * random_float ) );

      paddle.set_angle( 270.0 );

    }
    else if ( random_float == 0.0 )
    {

      paddle.set_magnitude( 0.0 );

      paddle.set_angle( starting_paddle_angle );
```

```
649       }
          else if ( random_float > 0.0 )
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   {
669     pause = true;
671     var fitness = 0.0;
673     for( var i = 0; i < learner.current_genome_fitness_tracking.length - 1; ++i )
675     {
677       if ( learner.current_genome_fitness_tracking[ i ] < learner.current_genome_fitness_tracking[ i + 1 ] )
679       {
681         fitness = fitness - 0.1;
683       }
          else if ( learner.current_genome_fitness_tracking[ i ] == learner.current_genome_fitness_tracking[ i +
              1 ] )
685       {
687         if ( learner.current_genome_fitness_tracking[ i ] == 0 && learner.current_genome_fitness_tracking[ i
              + 1 ] == 0 )
689         {
691           fitness = fitness + 0.1;
693         }
            else
695         {
697           fitness = fitness - 0.1;
699         }
701       }
          else if ( learner.current_genome_fitness_tracking[ i ] > learner.current_genome_fitness_tracking[ i +
              1 ] )
703       {
705         fitness = fitness + 0.1;
707       }
709     }
711     learner.current_genome_fitness_tracking = [ ];
713     fitness = fitness + ( 1 * learner.current_genome_paddle_hits );
715     learner.current_genome_paddle_hits = 0;
717     if ( fitness < 0.0 ) fitness = 0.0;
719     debug_manager.add_or_update( "Last Genome Fitness", fitness );
721     learner.genetic_algorithm.set_genome_fitness( learner.current_genome_being_evalutated, fitness );
723     if ( ( learner.current_genome_being_evalutated + 1 ) == learner.genetic_algorithm.get_population_size( )
            )
725     {
727       // Evaluated all the genomes in the population.
729       // Calculate population metrics.
731       // Adjust crossover and mutation probabilities if using self-adaptation.
733       learner.genetic_algorithm.sort_population( );
735       learner.genetic_algorithm.evaluate_population( );
737       if ( learner.parameters.genetic_algorithm.useSelfAdaptation )
739       {
741         learner.genetic_algorithm.adjust_crossover_and_mutation_probabilities( );
743       }
745       // Package up this population and send it to the database if its generation number is higher than
          // the highest generation number already in the database.
747       var data_string  = "generation_number="           + learner.genetic_algorithm.get_generation_number( )
              ;
749       data_string     += "&best_fitness="               + learner.genetic_algorithm.get_best_fitness( );
751       data_string     += "&average_fitness="            + learner.genetic_algorithm.get_average_fitness( );
753       data_string     += "&worst_fitness="              + learner.genetic_algorithm.get_worst_fitness( );
755       data_string     += "&population_size="            + learner.genetic_algorithm.get_population_size( );
```

```
    data_string     += "&number_of_genes_per_genome=" + learner.genetic_algorithm.
      get_number_of_genes_per_genome( );
    data_string     += "&crossover_probability="      + learner.genetic_algorithm.
      get_crossover_probability( );
    data_string     += "&mutation_probability="       + learner.genetic_algorithm.get_mutation_probability
      ( );
    data_string     += "&population_genes="           + learner.genetic_algorithm.
      get_population_genes_flattened( );

    database_manager.send_and_receive( "assets/scripts/store_genomes.php", data_string, database_manager,
      "adding_genome_population" );

    if ( run_experiment )
    {

      // Experiment of first 100 generations.

      if ( learner.genetic_algorithm.get_generation_number( ) < 100 )
      {

        var experiment_record_data_string  = "action=record";
        experiment_record_data_string     += "&generation_number="        + learner.genetic_algorithm.
      get_generation_number( );
        experiment_record_data_string     += "&best_fitness="             + learner.genetic_algorithm.
      get_best_fitness( );
        experiment_record_data_string     += "&average_fitness="          + learner.genetic_algorithm.
      get_average_fitness( );
        experiment_record_data_string     += "&worst_fitness="            + learner.genetic_algorithm.
      get_worst_fitness( );
        experiment_record_data_string     += "&crossover_probability="    + learner.genetic_algorithm.
      get_crossover_probability( );
        experiment_record_data_string     += "&mutation_probability="     + learner.genetic_algorithm.
      get_mutation_probability( );

        database_manager.send_and_receive( "assets/scripts/experiment.php", experiment_record_data_string,
       database_manager, "experiment_record" );

        if ( ( learner.genetic_algorithm.get_generation_number( ) + 1 ) % 10 == 0 )
        {

          var experiment_store_data_string  = "action=store";
          experiment_store_data_string     += "&generation_number="        + learner.genetic_algorithm.
      get_generation_number( );
          experiment_store_data_string     += "&fitness="                  + learner.genetic_algorithm.
      get_genome_fitness( learner.genetic_algorithm.get_fittest_genome_index( ) );
          experiment_store_data_string     += "&crossover_probability="    + learner.genetic_algorithm.
      get_crossover_probability( );
          experiment_store_data_string     += "&mutation_probability="     + learner.genetic_algorithm.
      get_mutation_probability( );
          experiment_store_data_string     += "&genes="                    + learner.genetic_algorithm.
      get_genome_genes_flattened( learner.genetic_algorithm.get_fittest_genome_index( ) );

          database_manager.send_and_receive( "assets/scripts/experiment.php", experiment_store_data_string
      , database_manager, "experiment_store" );

        }

      }

    }

    // Generate a new generation.

    learner.genetic_algorithm.generate_new_generation( );

    // Get the makeup of the population.

    learner.genetic_algorithm.compute_population_makeup( );

    // Load in the first genome's genes.

    learner.current_genome_being_evalutated = 0;

    learner.neural_net.put_weights( learner.genetic_algorithm.get_genome_genes( learner.
      current_genome_being_evalutated ) );

  }
  else
  {

    // Test the next genome in the population.

    learner.current_genome_being_evalutated += 1;

    learner.neural_net.put_weights( learner.genetic_algorithm.get_genome_genes( learner.
      current_genome_being_evalutated ) );

  }

  learner.evaluate_current_genome = false;

  reset( );

}

*/

function handle_genome_evaluation( )
{

  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_evalutated, fitness );
843
      if ( ( learner.current_genome_being_evalutated + 1 ) == learner.genetic_algorithm.get_population_size( )
           )
845   {
847     // Evaluated all the genomes in the population.
        // 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( )
          ;
        data_string     += "&best_fitness="               + learner.genetic_algorithm.get_best_fitness( );
867     data_string     += "&average_fitness="            + learner.genetic_algorithm.get_average_fitness( );
        data_string     += "&worst_fitness="              + learner.genetic_algorithm.get_worst_fitness( );
869     data_string     += "&population_size="            + learner.genetic_algorithm.get_population_size( );
        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( );
        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 )
883       {
885         var experiment_record_data_string  = "action=record";
            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( );
            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( );
            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";
899           experiment_store_data_string     += "&generation_number="        + learner.genetic_algorithm.
          get_generation_number( );
              experiment_store_data_string     += "&fitness="                  + learner.genetic_algorithm.
          get_genome_fitness( learner.genetic_algorithm.get_fittest_genome_index( ) );
901           experiment_store_data_string     += "&crossover_probability="    + learner.genetic_algorithm.
          get_crossover_probability( );
              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_evalutated = 0;
925     learner.neural_net.put_weights( learner.genetic_algorithm.get_genome_genes( learner.
          current_genome_being_evalutated ) );
927   }
      else
929   {
931     // Test the next genome in the population.
933     learner.current_genome_being_evalutated += 1;
935     learner.neural_net.put_weights( learner.genetic_algorithm.get_genome_genes( learner.
          current_genome_being_evalutated ) );
937   }
939   learner.evaluate_current_genome = false;
941   reset( );
943 }
945 var ball_magnitude_reduction_threshold = 100;
    var ball_magnitude_reduce_by_percentage = .5;
947
    function collision_handler( colliding_objects )
949 {
951   learner.evaluate_current_genome = false;
953   for ( var i = 0; i < colliding_objects.length; ++i )
      {
955
        // Ball collisions
957
        if ( colliding_objects[ i ][ 0 ].id == "ball" && colliding_objects[ i ][ 1 ].id == "paddle" || (
          colliding_objects[ i ][ 0 ].id == "paddle" && colliding_objects[ i ][ 1 ].id == "ball" ) )
959     {
961       ball.set_magnitude( ( ball.get_magnitude( ) - ( ball.get_magnitude( ) *
          ball_magnitude_reduce_by_percentage ) ) > ball_magnitude_reduction_threshold ? ( ball.get_magnitude(
          ) - ( ball.get_magnitude( ) * .1 ) ) : 0 );
963       learner.current_genome_paddle_hits = learner.current_genome_paddle_hits + 1;
965       if ( ball.get_magnitude( ) == 0 )
          {
967
            learner.evaluate_current_genome = true;
969
          }
971
          // Wind the ball's position back, going along the direction opposite of the angle it was traveling
          along when it collided with
973       // the paddle until it no longer collides with the paddle.
          // This is easier to calculate than finding the intersection of a line along the direction of the
          ball's vector with that of
975       // the paddle's top, right, bottom, and left boundary lines.
977       var reverse_angle = ball.mod_degrees( ( ball.get_angle( ) + 180 ) );
          var dx =  5 * Math.cos( ball.degrees_to_radians( reverse_angle ) );
979       var dy = -5 * Math.sin( ball.degrees_to_radians( reverse_angle ) ); // Negative one because of the
          mirrored coordinate system.
981       /*
983       while ( physics_engine.rectangle_intersection( paddle.dynamic_object, ball.dynamic_object ) )
          {
985
            console.log( "x" );
987
            ball.dynamic_object.move_left_top( dx, dy );
989
          }
991
          */
993
          // There are two cases:
995       //  1. The ball hits the paddle from the top or bottom.
          //  2. The ball hits the paddle from either the left or right.
997
          if ( ( ball.dynamic_object.get_top( ) > paddle.dynamic_object.get_bottom( ) ) || ( ball.
          dynamic_object.get_bottom( ) < paddle.dynamic_object.get_top( ) ) )
999       {
1001        // Top or bottom side case.
```

```
      // Physics_Object.set_angle expects an angle as if y+ was going UP the screen.
      // So say the ball hits the top wall at 35 degrees. The angle of incidence would be 35 degrees and
     so the ball should be reflected back
      // at -35 degrees or -35 % 360 = 325  which we set.
      // -1 * 35 = -35 mod 360 = 325
      // |_||_||_||_||_||_||_||_||_||_||_||_||_||_||_|
      //                        .
      //                          )
      // .   )                                      .
      // ---------------------------------------------

      var reflection_angle = ball.mod_degrees( -1 * ball.get_angle( ) );
      ball.set_angle( reflection_angle );

    }
    else
    {

      // Left or right side case.

      // Physics_Object.set_angle expects an angle as if y+ was going UP the screen.
      // So say the ball hits the paddle at 170 degrees. The angle of incidence would be 10 degrees and
    so the ball should be reflected back
      // at 10 degrees which we set.
      // -1 * ( 170 - 180 ) = 10
      // |/|
      // |/|
      // |/|
      // |/|                 .
      // |/|.________________)___
      // |/|                (.
      // |/|
      // |/|


      var reflection_angle = -1 * ( ball.get_angle( ) - 180.0);
      ball.set_angle( reflection_angle );

    }

  }
  else if ( colliding_objects[ i ][ 0 ].id == "ball" && colliding_objects[ i ][ 1 ].id == "top_wall" ||
    ( colliding_objects[ i ][ 0 ].id == "top_wall" && colliding_objects[ i ][ 1 ].id == "ball" ) )
  {

    ball.set_magnitude( ( ball.get_magnitude( ) - ( ball.get_magnitude( ) *
    ball_magnitude_reduce_by_percentage ) ) > ball_magnitude_reduction_threshold ? ( ball.get_magnitude(
     ) - ( ball.get_magnitude( ) * .1 ) ) : 0 );

    if ( ball.get_magnitude( ) == 0 )
    {

      learner.evaluate_current_genome = true;

    }


    ball.dynamic_object.move_left_top( 0, ( -1 * ball.dynamic_object.get_top( ) ) + 1 );

    // Physics_Object.set_angle expects an angle as if y+ was going UP the screen.
    // So say the ball hits the top wall at 35 degrees. The angle of incidence would be 35 degrees and
    so the ball should be reflected back
    // at -35 degrees or -35 % 360 = 325  which we set.
    // -1 * 35 = -35 mod 360 = 325
    // |_||_||_||_||_||_||_||_||_||_||_||_||_||_||_|
    //                        .
    //                          )
    // .   )                                      .
    // ---------------------------------------------


    var reflection_angle = ball.mod_degrees( -1 * ball.get_angle( ) );
    ball.set_angle( reflection_angle );

  }
  else if ( colliding_objects[ i ][ 0 ].id == "ball" && colliding_objects[ i ][ 1 ].id == "right_wall"
    || ( colliding_objects[ i ][ 0 ].id == "right_wall" && colliding_objects[ i ][ 1 ].id == "ball" ) )
  {

    ball.set_magnitude( ( ball.get_magnitude( ) - ( ball.get_magnitude( ) *
    ball_magnitude_reduce_by_percentage ) ) > ball_magnitude_reduction_threshold ? ( ball.get_magnitude(
     ) - ( ball.get_magnitude( ) * .1 ) ) : 0 );

    if ( ball.get_magnitude( ) == 0 )
    {

      learner.evaluate_current_genome = true;

    }

    ball.dynamic_object.move_left_top( ( -1 * ( ball.dynamic_object.get_right( ) - window.innerWidth ) )
     -1, 0 );

    var reflection_angle = -1 * ( ball.get_angle( ) - 180.0);
    ball.set_angle( reflection_angle );

  }
  else if ( colliding_objects[ i ][ 0 ].id == "ball" && colliding_objects[ i ][ 1 ].id == "bottom_wall"
    || ( colliding_objects[ i ][ 0 ].id == "bottom_wall" && colliding_objects[ i ][ 1 ].id == "ball" ) )
  {
```

```
      ball.set_magnitude( ( ball.get_magnitude( ) - ( ball.get_magnitude( ) *
       ball_magnitude_reduce_by_percentage) ) > ball_magnitude_reduction_threshold ? ( ball.get_magnitude(
       ) - ( ball.get_magnitude( ) * .1 ) ) : 0 );

      if ( ball.get_magnitude( ) == 0 )
      {

        learner.evaluate_current_genome = true;

      }

      ball.dynamic_object.move_left_top( 0, ( -1 * ( ball.dynamic_object.get_bottom( ) - window.
       innerHeight ) ) -1 );

      var reflection_angle = ball.mod_degrees( -1 * ball.get_angle( ) );
      ball.set_angle( reflection_angle );

    }
    else if ( colliding_objects[ i ][ 0 ].id == "ball" && colliding_objects[ i ][ 1 ].id == "left_wall" ||
       ( colliding_objects[ i ][ 0 ].id == "left_wall" && colliding_objects[ i ][ 1 ].id == "ball" ) )
    {

      ball.set_magnitude( ( ball.get_magnitude( ) - ( ball.get_magnitude( ) *
       ball_magnitude_reduce_by_percentage ) ) > ball_magnitude_reduction_threshold ? ( ball.get_magnitude(
        ) - ( ball.get_magnitude( ) * .1 ) ) : 0 );

      learner.evaluate_current_genome = true;

      ball.dynamic_object.move_left_top( ( -1 * ( ball.dynamic_object.get_left( ) ) ) + 1, 0 );

      var reflection_angle = -1 * ( ball.get_angle( ) - 180.0);
      ball.set_angle( reflection_angle );

    }

    // Paddle collisions.

    if ( colliding_objects[ i ][ 0 ].id == "paddle" && colliding_objects[ i ][ 1 ].id == "top_wall" || (
       colliding_objects[ i ][ 0 ].id == "top_wall" && colliding_objects[ i ][ 1 ].id == "paddle" ) )
    {

      paddle.dynamic_object.move_left_top( 0, ( -1 * paddle.dynamic_object.get_top( ) ) + 1 );

    }
    else if ( colliding_objects[ i ][ 0 ].id == "paddle" && colliding_objects[ i ][ 1 ].id == "bottom_wall
       " || ( colliding_objects[ i ][ 0 ].id == "bottom_wall" && colliding_objects[ i ][ 1 ].id == "paddle"
        ) )
    {

      paddle.dynamic_object.move_left_top( 0, ( -1 * ( paddle.dynamic_object.get_bottom( ) - window.
       innerHeight ) ) -1 );

    }

  }

}
```

Listing B.2: Genetic_Algorithm.js

```
/*
 *
 * David Lettier (C) 2013.
 *
 * http://www.lettier.com/
 *
 * Code ported to JS and HEAVILY modified from original C++ source
 * found at http://www.ai-junkie.com/ann/evolved/nnt1.html and
 * written by Mat Buckland.
 *
 * Implements a genetic algorithm for NN weight tuning.
 *
 */

function Genome( genes, fitness )
{

  this.genes = null;
  this.fitness = null;

  if ( genes == undefined ) this.genes = new Array( );
  else this.genes = genes;

  if ( fitness == undefined ) this.fitness = 0.0;
  else this.fitness = fitness;

  // Used to calculate either the crossover progress or mutation progress.
  // If this genome is created via crossover, use the weighted average
  // based on the cross over point.
  // So if the crossover point is say 9 and the genome length is 10,
  // then the weighted average pf = (p1.f*.9) + (p2.f*.1).
  // In other words the offspring received 90% of its genes from parent one
  // and it received 10% of its genes from parent two so its parent fitness is
  // 90% of parent one's fitness and 10% of parent two's fitness.

  this.parent_fitness = 0.0;

  // Created by means if this genome was generated either by randomness, crossover,
  // mutation, both crossover and mutation, or elitism.
  // Initially it is created from nothing so set it to -1.
```

```
  // 0 = randomness, 1 = crossover, 2 = mutation, 3 = crossover & mutation, 4 = elitism

  // This encoding is to facilitate crossover's and mutation's progress at producing fitter offspring than
      the offspring's parents.

  this.created_by = 0;

}

function Genetic_Algorithm( params )
{

  // Size of population.

  this.population_size = params.popSize;

  // Amount of genes per genome.

  this.number_of_genes_per_genome = params.nGenesPerGenome;

  // Use rank in selection?

  this.use_rank_fitness = params.useRankFitness;

  // Perform crossover and mutation sequentially or separately?

  this.perform_crossover_and_mutation_sequentially = params.pCrMuSeq;

  // Probability of genome's crossing over bits.
  // 0.7 is pretty good.

  this.crossover_probability               = params.iCProb;
  this.crossover_probability_minimum       = 0.001;
  this.crossover_probability_adjustment    = 0.01;
  this.crossover_operator_progress_average = 0.0;
  this.observed_crossover_rate             = 0.0;
  this.total_number_of_crossovers          = 0;
  this.total_number_of_crossover_attempts  = 0;

  // Probability that a genomes bits will mutate.
  // Try figures around 0.05 to 0.3-ish.

  this.mutation_probability               = params.iMProb;
  this.mutation_probability_minimum       = 0.001;
  this.mutation_probability_adjustment    = 0.01;
  this.mutation_operator_progress_average = 0.0;
  this.observed_mutation_rate             = 0.0;
  this.total_number_of_mutations          = 0;
  this.total_number_of_mutation_attempts  = 0;

  // Set the number of elite that go on to the next generation.

  this.number_of_elite = params.nElite;

  // This holds the entire population of genomes.

  this.population = new Array( );

  // Total fitness of population.

  this.total_fitness = 0;

  // Average fitness.

  this.average_fitness = 0;

  // Best fitness this population.

  this.best_fitness = 0;

  // Worst fitness.

  this.worst_fitness = 0;

  // Keeps track of the best genome.

  this.fittest_genome_index = -1;

  // Keep track of the worst genome.

  this.weakest_genome_index = -1;

  // Generation number.

  this.generation_number = 0;

  // Current population makeup of randoms, crossovers, mutants, crossover mutants, elites.

  this.population_makeup = "";

  // Initialize population with genomes consisting of random
  // genes and all fitness's set to zero.

  for ( var i = 0; i < this.population_size; ++i )
  {

    this.population.push( new Genome( ) );

    for ( var j = 0; j < this.number_of_genes_per_genome; ++j )
    {

      this.population[ i ].genes.push( get_random_float( -1.0, 1.0 ) );
```

```
142
    }
144
  }
146
  this.replace_population_genes = function ( replacement_population_genes )
148 {
150   if ( replacement_population_genes === undefined ||
        replacement_population_genes.length === 0  ||
152     replacement_population_genes.length != ( this.population_size * this.number_of_genes_per_genome ) )
    {
154
      console.error( "[Genetic_Algorithm:replace_population_genes] Replacement population genes invalid."
        );
156
      return null;
158
    }
160
    // Assumes replace_population_genes is one big array.
162 // Splices the big array based on the number of genes
    // per genome.
164
166 // Big array: [ 1,1,1,1,1,1,1,1,1,1 ] >>
    // Population: [ [ 1, 1 ] G0
168 //               [ 1, 1 ] G1
    //               [ 1, 1 ] ...
170 //               [ 1, 1 ] ...
    //               [ 1, 1 ] GN-1
172 //             ]

174 var k = 0;

176 for ( var i = 0; i < this.population_size; ++i )
    {
178
      this.population[ i ].genes = [ ];
180
      for ( var j = 0; j < this.number_of_genes_per_genome; ++j )
182   {

184     this.population[ i ].genes.push( replacement_population_genes[ k ] );

186     k += 1;

188   }

190 }

192 }

194 this.selection_operator = function ( number_of_indexes )
    {
196
      // Assumes population has been evaluated.
198
      // Assumes population is sorted in ascending order according to fitness.
200
      // Roulette selection of n genome indexes in the population.
202
      if ( !this.use_rank_fitness )
204   {

206     // Say we have a population of 4 with these fitness values:
        // G-1: 1
208     // G-2: 2
        // G-3: 3
210     // G-4: 4
        // Total fitness: 10
212     // Probabilities:
        // G-1: .1
214     // G-2: .2
        // G-3: .3
216     // G-4: .4
        //
218     // Now shift them over by the running sum.
        // This give them a portion on the number line [0.0,1.0] proportional to their
220     // fitness.
        //
222     // G-1: .1
        // G-2: G-1 + .2 = .3
224     // G-3: G-2 + .3 = .6
        // G-4: G-3 + .4 = 1.0
226     //
        // 0.0----.10----.20----.30----.40----.50----.60----.70----.80----.90----1.0
228     //        G-1           G-2                 G-3                         G-4
        //
230     // Now selected a random float in [0.0,1.0]:
        // RF: .51
232     //
        // 0.0----.10----.20----.30----.40----.50----.60----.70----.80----.90----1.0
234     //        G-1           G-2              RF  G-3                         G-4
        //
236     // G-3 gets selected for mating.

238     var probabilities = new Array( );

240     var genome_indexes_selected = new Array( );

242     if ( this.total_fitness == 0 )
```

```
    {
      // So that we don't divide by zero.
      // This means genomes all have zero fitness
      // so just select random genome indexes.

      for ( var i = 0; i < number_of_indexes; ++i )
      {
        genome_indexes_selected.push( get_random_integer( 0, this.population_size - 1 ) );
      }

      return genome_indexes_selected;
    }

    probabilities.push( this.population[ 0 ].fitness / this.total_fitness );

    for ( var i = 1; i < this.population_size; ++i )
    {
      probabilities.push( probabilities[ i - 1 ] + ( this.population[ i ].fitness / this.total_fitness )
     );
    }

    while ( genome_indexes_selected.length < number_of_indexes )
    {
      var random_number = get_random_float( 0.0, 1.0 );

      for ( var i = 0; i < this.population_size; ++i )
      {
        if ( random_number <= probabilities[ i ] )
        {
          genome_indexes_selected.push( i );
        }
      }
    }

    return genome_indexes_selected;
  }
  else
  {
    // Give the worst genome a rank fitness of 1.
    // Give the second worst genome a rank fitness of 2.
    // ...
    // Give the best genome a rank fitness of the population size.

    // Now, based on rank fitness, do a roulette selection where the
    // probabilities are based on the rank fitness.

    var probabilities = new Array( );

    var genome_indexes_selected = new Array( );

    // Rank fitness of the first is 1.
    // Probability is 1/(n(n+1)/2).
    // Where n is population size.
    // (n(n+1)/2) = the total rank fitness.
    // Summing the numbers from 1 to population size.
    // Say population size is 10.
    // Rank fitness: G-1 = 1, G-2 = 2, ..., G-10 = 10.
    // Total rank fitness is 1+2+3+...+10 = n(n+1)/2 = (10*11)/2 = 55
    // Probabilities:
    // G-1: 1/55
    // G-2: G-1 + 2/55
    // ...
    // G-10: G-9 + 10/55

    var total_rank_fitness = ( this.population_size * ( this.population_size + 1 ) ) / 2;

    probabilities.push( 1 / total_rank_fitness ); // First rank fitness probability.

    // Rest of the rank fitness probabilities.

    for ( var i = 1; i < this.population_size; ++i )
    {
      probabilities.push( probabilities[ i - 1 ] + ( ( i + 1 ) / total_rank_fitness ) );
    }

    while ( genome_indexes_selected.length < number_of_indexes )
    {
      var random_number = get_random_float( 0.0, 1.0 );

      for ( var i = 0; i < this.population_size; ++i )
      {
        if ( random_number <= probabilities[ i ] )
        {
```

```
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
        // i = 2 check.
368     // i = 1 decrement.
        // A[ ( ( 10 - 1 = 9 ) - i ) = 8 ]
370     // i = 1 check.
        // i = 0 decrement.
372     // A[ ( ( 10 - 1 = 9 ) - i ) = 9 ]
        // i = 0 check.
374     // Stop.

376     var i = this.number_of_elite;

378     while( i-- )
        {
380
          var genome_temp = deep_copy( this.population[ ( this.population_size - 1 ) - i ] );
382
          genome_temp.fitness        = 0;
384       genome_temp.parent_fitness = 0;
          genome_temp.created_by     = 4;
386
          new_population.push( genome_temp );
388
          if ( new_population.length == this.population_size ) return;
390
        }
392
      }
394
      this.crossover_operator = function ( parent_one_index, parent_two_index )
396   {

398     // One point crossover operator.

400     // Do we crossover?

402     if ( get_random_float( 0.0, 1.0 ) <= this.crossover_probability )
        {
404
          // If the parents are the same genome then this is not a true crossover.
406
          if ( parent_one_index === parent_two_index ) return 0;
408
          // Only returns one crossed offspring.
410
          var offspring = new Genome( );
412
          // Determine a crossover point.
414
          // Let the uniform sample be in the range of [1,n-1].
416       // If the crossover point was zero than no true crossover takes place
          // as all of one parent's genes get copied into the offspring.
418       // If the cp = n-1 then at least you get n-1 from one parent and 1
          // from another parent.
420
          var crossover_point = get_random_integer( 1, ( this.number_of_genes_per_genome - 1 ) );
422
          // Cross the parent's genes in the offspring.
424
          offspring.genes = [ ];
426
          offspring.fitness = 0;
428
          offspring.parent_fitness = 0;
430
          for ( var i = 0; i < crossover_point; ++i )
432       {

434         offspring.genes.push( deep_copy( this.population[ parent_one_index ].genes[ i ] ) );

436       }

438       for ( var i = crossover_point; i < this.number_of_genes_per_genome; ++i )
          {
440
            offspring.genes.push( deep_copy( this.population[ parent_two_index ].genes[ i ] ) );
442
          }
444
          // Determine if a crossover actually took place.
```

```
446       // The offspring should not match the parent one's genes and
          // it should not match parent two's genes as the offspring
448       // should be a combination of the two.

450       if ( ( offspring.genes.toString( ) != this.population[ parent_one_index ].genes.toString( ) ) &&
            ( offspring.genes.toString( ) != this.population[ parent_two_index ].genes.toString( ) ) )
452
          {
454
            // Weighted average fitness of the parents based on crossover point
456         // determining percentage of genes received from parent one and parent two.
            // Let the number of genes per genome be 41 and let the crossover point be 1.
458         // Offspring gets [0,1) = 1 gene from parent one and [1,41) = 40 genes from parent two.
            // PF = PF1 * (1/41) + PF2 * ((41-1)/41).
460
            var parent_one_contribution = ( this.population[ parent_one_index ].fitness * ( (
                           crossover_point ) / ( this.number_of_genes_per_genome ) ) );
462         var parent_two_contribution = ( this.population[ parent_two_index ].fitness * ( ( this.
          number_of_genes_per_genome - crossover_point ) / ( this.number_of_genes_per_genome ) ) );

464         offspring.parent_fitness = parent_one_contribution + parent_two_contribution;

466         offspring.created_by = 1;

468         return offspring;

470       }
          else
472       {

474         return 0;

476       }

478     }
        else
480     {

482       return 0;

484     }

486   }

488   this.mutation_operator = function ( parent_index )
      {
490
        // Mutates parent genome's genes on a whole genome basis based on the mutation probability.
492
        // Gaussian distribution mutation.
494
        // Do we mutate?
496
        if ( get_random_float( 0.0, 1.0 ) <= this.mutation_probability )
498     {

500       // Reference: http://www.nashcoding.com/2010/07/07/evolutionary-algorithms-the-little-things-youd-
          never-guess-part-1/

502       function gaussian_distribution( mean, standard_deviation )
          {
504
            // Two uniformally distributed random variable samples.
506
            var x1 = Math.random( );
508         var x2 = Math.random( );

510         // The method requires sampling from a uniform random of (0,1]
            // but Math.random( ) returns a sample of [0,1).
512
            if ( x1 == 0.0 ) x1 = 1.0;
514         if ( x2 == 0.0 ) x2 = 1.0;

516         // Box-Muller transformation for Z_0.

518         var y1 = Math.sqrt( -2.0 * Math.log( x1 ) ) * Math.cos( 2.0 * Math.PI * x2 );

520         return ( y1 * standard_deviation ) + mean;

522       }

524       // Create an offspring blank.

526       var offspring = new Genome( );

528       offspring.genes = [ ];

530       offspring.genes = deep_copy( this.population[ parent_index ].genes );

532       offspring.fitness = 0;

534       offspring.parent_fitness = 0;

536       // Begin to mutate.

538       var mutated = false;

540       for ( var i = 0; i < this.number_of_genes_per_genome; ++i )
          {
542
            // Mutate this gene by sampling a value from a normal distribution
544         // where the mean is the current gene value and the standard deviation
```

```
        // the is mutation step = mutation probability in the range [0,1].
        // A low mutation probability will give a mutated gene value close to the original gene
        // value (the mean) (most of the time) as the standard deviation is small and therefore the
      mutation step is small.
        // A high mutation probability will give (or it can easily) a mutated gene value farther from the
      original gene
        // value (the mean) as the standard deviation is large and therefore the mutation step is large.

        // Note that gv = gv +   *N(0,1) is the same as gv = N(gv,  ).

        // Clamp the gene to range [-1,1].

        var temp_gene_value = deep_copy( offspring.genes[ i ] );

        offspring.genes[ i ] = gaussian_distribution( offspring.genes[ i ], this.mutation_probability );
        offspring.genes[ i ] = get_clamped_value( offspring.genes[ i ], -1.0, 1.0 );

        // Test if it was truly mutated.

        if ( temp_gene_value != offspring.genes[ i ] )
        {

          mutated = true;

        }

      }

      if ( mutated ) // If truly mutated.
      {

        offspring.parent_fitness = deep_copy( this.population[ parent_index ].fitness );

        offspring.created_by = 2;

        return offspring;

      }
      else
      {

        return 0;

      }

    }
    else
    {

      return 0;

    }

  }

  this.crossover_then_mutate_operator = function ( parent_one_index, parent_two_index )
  {

    // Crossover and mutation done sequentially as in more traditional genetic algorithms.

    // First attempts crossover and then attempts mutation.

    var offspring_one = deep_copy( this.population[ parent_one_index ] );
    var offspring_two = deep_copy( this.population[ parent_one_index ] );

    offspring_one.fitness = 0.0;
    offspring_two.fitness = 0.0;

    offspring_one.parent_fitness = null;
    offspring_two.parent_fitness = null;

    offspring_one.created_by = 0;
    offspring_two.created_by = 0;

    // Attempt crossover.

    var crossover_point = get_random_integer( 0, ( this.number_of_genes_per_genome - 1 ) );

    if ( ( get_random_float( 0.0, 1.0 ) <= this.crossover_probability ) && ( parent_one_index !=
       parent_two_index ) && ( crossover_point != 0 ) )
    {

      // Cross the parent's genes in the offspring.

      offspring_one.genes = [ ];
      offspring_two.genes = [ ];

      for ( var i = 0; i < crossover_point; ++i )
      {

        offspring_one.genes.push( deep_copy( this.population[ parent_one_index ].genes[ i ] ) );
        offspring_two.genes.push( deep_copy( this.population[ parent_two_index ].genes[ i ] ) );

      }

      for ( var i = crossover_point; i < this.number_of_genes_per_genome; ++i )
      {

        offspring_one.genes.push( deep_copy( this.population[ parent_two_index ].genes[ i ] ) );
        offspring_two.genes.push( deep_copy( this.population[ parent_one_index ].genes[ i ] ) );

      }
```

```
644
      var parent_one_contribution = ( this.population[ parent_one_index ].fitness * ( (
                      crossover_point ) / ( this.number_of_genes_per_genome ) ) );
646   var parent_two_contribution = ( this.population[ parent_two_index ].fitness * ( ( this.
      number_of_genes_per_genome - crossover_point ) / ( this.number_of_genes_per_genome ) ) );

648   offspring_one.parent_fitness = parent_one_contribution + parent_two_contribution;
      offspring_one.created_by = offspring_one.created_by + 1;
650
      parent_two_contribution = ( this.population[ parent_two_index ].fitness * ( (
                crossover_point ) / ( this.number_of_genes_per_genome ) ) );
652   parent_one_contribution = ( this.population[ parent_one_index ].fitness * ( ( this.
      number_of_genes_per_genome - crossover_point ) / ( this.number_of_genes_per_genome ) ) );

654   offspring_two.parent_fitness = parent_two_contribution + parent_one_contribution;
      offspring_two.created_by = offspring_two.created_by + 1;
656
    }
658
    // Crossover may or may not have happened but now try to mutation.
660
    // Normal distribution sample function.
662
    function gaussian_distribution( mean, standard_deviation )
664 {
666   // Two uniformally distributed random variable samples.

668   var x1 = Math.random( );
      var x2 = Math.random( );
670
      // The method requires sampling from a uniform random of (0,1]
672   // but Math.random( ) returns a sample of [0,1).

674   if ( x1 == 0.0 ) x1 = 1.0;
      if ( x2 == 0.0 ) x2 = 1.0;
676
      // Box-Muller transformation for Z_0.
678
      var y1 = Math.sqrt( -2.0 * Math.log( x1 ) ) * Math.cos( 2.0 * Math.PI * x2 );
680
      return ( y1 * standard_deviation ) + mean;
682
    }
684
    // Attempt to mutate offspring one.
686
    var mutated_one = false;
688
    for ( var i = 0; i < this.number_of_genes_per_genome; ++i )
690 {
692   // Mutate this gene by sampling a value from a normal distribution
      // where the mean is the current gene value and the standard deviation
694   // the is mutation step = mutation probability in the range [0,1].
      // A low mutation probability will give a mutated gene value close to the original gene
696   // value (the mean) (most of the time) as the standard deviation is small and therefore the mutation
       step is small.
      // A high mutation probability will give (or it can easily) a mutated gene value farther from the
      original gene
698   // value (the mean) as the standard deviation is large and therefore the mutation step is large.

700   // Note that gv = gv +    * N( 0, 1 ) is the same as gv = N( gv,    ).

702   // Clamp the gene to range [-1,1].

704   if ( get_random_float( 0.0, 1.0 ) <= this.mutation_probability ) // Mutate this gene?
      {
706
        var temp_gene_value_one = deep_copy( offspring_one.genes[ i ] );
708
        offspring_one.genes[ i ] = gaussian_distribution( offspring_one.genes[ i ], this.
      mutation_probability );
710     // offspring_one.genes[ i ] = gaussian_distribution( offspring_one.genes[ i ], 0.5 );
        // offspring_one.genes[ i ] = offspring_one.genes[ i ] + ( get_random_float( -1.0, 1.0 ) * .3 );
712     offspring_one.genes[ i ] = get_clamped_value( offspring_one.genes[ i ], -1.0, 1.0 );

714     // Test if it was truly mutated.

716     if ( temp_gene_value_one != offspring_one.genes[ i ] )
        {
718
          mutated_one = true;
720
        }
722
      }
724
    }
726
    // Attempt to mutate offspring two.
728
    var mutated_two = false;
730
    for ( var i = 0; i < this.number_of_genes_per_genome; ++i )
732 {
734   // Mutate this gene by sampling a value from a normal distribution
      // where the mean is the current gene value and the standard deviation
736   // the is mutation step = mutation probability in the range [0,1].
      // 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
           step is small.
          // A high mutation probability will give (or it can easily) a mutated gene value farther from the
          original gene
740       // value (the mean) as the standard deviation is large and therefore the mutation step is large.

742       // Note that gv = gv +    * N( 0, 1 ) is the same as gv = N( gv,    ).

744       // Clamp the gene to range [-1,1].

746       if ( get_random_float( 0.0, 1.0 ) <= this.mutation_probability ) // Mutate this gene?
          {
748
            var temp_gene_value_two = deep_copy( offspring_two.genes[ i ] );
750
            offspring_two.genes[ i ] = gaussian_distribution( offspring_two.genes[ i ], this.
          mutation_probability );
752         // offspring_two.genes[ i ] = gaussian_distribution( offspring_two.genes[ i ], 0.5 );
            // offspring_two.genes[ i ] = offspring_two.genes[ i ] + ( get_random_float( -1.0, 1.0 ) * .3 );
754         offspring_two.genes[ i ] = get_clamped_value( offspring_two.genes[ i ], -1.0, 1.0 );

756         // Test if it was truly mutated.

758         if ( temp_gene_value_two != offspring_two.genes[ i ] )
            {
760
              mutated_two = true;
762
            }
764
          }
766
        }
768
        if ( mutated_one ) // If truly mutated.
770     {

772       // Mutation = 2, crossover = 1, crossover + mutation = 3.

774       offspring_one.created_by = offspring_one.created_by + 2;

776       // If this offspring was only mutated, that is, it was not crossed then get its parent fitness.
          // It it was crossed before being mutated then offspring_one.created would equal 3.
778
          if ( offspring_one.created_by === 2 )
780       {

782         offspring_one.parent_fitness = deep_copy( this.population[ parent_one_index ].fitness );

784       }

786     }

788     if ( mutated_two ) // If truly mutated.
        {
790
          offspring_two.created_by = offspring_two.created_by + 2;
792
          // If this offspring was only mutated, that is, it was not crossed then get its parent fitness.
794       // It it was crossed before being mutated then offspring_two.created would equal 3.

796       if ( offspring_two.created_by === 2 )
          {
798
            offspring_two.parent_fitness = deep_copy( this.population[ parent_two_index ].fitness );
800
          }
802
        }
804
        // No parents->offspring not crossed and/or not mutated enter into the new population.
806     // Each offspring going into the new population must either crossed, mutated, or both.

808     if ( ( offspring_one.created_by === 0 ) || ( offspring_two.created_by === 0 ) )
        {
810
          return 0;
812
        }
814     else if ( ( offspring_one.genes.toString( ) === this.population[ parent_one_index ].genes.toString( )
            ) || ( offspring_two.genes.toString( ) === this.population[ parent_two_index ].genes.toString( ) ) )
        {
816
          return 0;
818
        }
820     else
        {
822
          return { one: offspring_one, two: offspring_two };
824
        }
826
      }
828
      this.evaluate_population = function ( )
830   {

832     this.reset_population_evaluation( );

834     var highest_so_far = this.population[ 0 ].fitness;
        var lowest_so_far  = this.population[ 0 ].fitness;
```

```
836
838     this.fittest_genome_index = 0;
        this.weakest_genome_index = 0;
840
        this.total_fitness  = this.population[ 0 ].fitness;
842     this.best_fitness   = this.population[ 0 ].fitness;
        this.worst_fitness  = this.population[ 0 ].fitness;
844
        for ( var i = 1; i < this.population_size; ++i )
846     {
          // Update fittest if necessary.
848
          if ( highest_so_far < this.population[ i ].fitness )
850       {
852         highest_so_far = this.population[ i ].fitness;
854         this.fittest_genome_index = i;
856         this.best_fitness = highest_so_far;
858       }
860       // Update worst if necessary.
862       if ( lowest_so_far > this.population[ i ].fitness  )
          {
864
            lowest_so_far = this.population[ i ].fitness;
866
            this.weakest_genome_index = i;
868
            this.worst_fitness = lowest_so_far;
870       }
872
          this.total_fitness += this.population[ i ].fitness;
874
876     } // Next genome.
878     this.average_fitness = this.total_fitness / this.population_size;
880   }
882   this.reset_population_evaluation = function ( )
      {
884
        this.total_fitness         = 0;
886     this.best_fitness          = 0;
        this.worst_fitness         = 0;
888     this.average_fitness       = 0;
        this.fittest_genome_index  = -1;
890     this.weakest_genome_index  = -1;
892   }
894   this.compute_population_makeup = function ( )
      {
896
        var randoms            = 0;
898     var crossovers         = 0;
        var mutants            = 0;
900     var crossover_mutants  = 0;
        var elites             = 0;
902
        for ( var i = 0; i < this.population_size; ++i )
904     {
906       if ( this.population[ i ].created_by === 0 )
          {
908
            randoms = randoms + 1;
910
          }
912       else if ( this.population[ i ].created_by === 1 )
          {
914
            crossovers = crossovers + 1;
916
          }
918       else if ( this.population[ i ].created_by === 2 )
          {
920
            mutants = mutants + 1;
922
          }
924       else if ( this.population[ i ].created_by === 3 )
          {
926
            crossover_mutants = crossover_mutants + 1;
928
          }
930       else if ( this.population[ i ].created_by === 4 )
          {
932
            elites = elites + 1;
934
          }
936
      }
```

```
938
      this.population_makeup = randoms + " " + crossovers + " " + mutants + " " + crossover_mutants + " " +
        elites;
940
    }
942
    this.adjust_crossover_and_mutation_probabilities = function ( )
944 {
946   // Calculate the crossover and mutation operators' progress where
      // their progress is based on how well they produced offspring that
948   // had a better fitness than their parent.

950   var crossover_operator_progress_sum = 0;
      var number_of_crossovers            = 0;
952
      var mutation_operator_progress_sum  = 0;
954   var number_of_mutations             = 0;

956   // Sum all of the progresses.

958   for ( var i = 0; i < this.population_size; ++i )
      {
960
        if ( this.population[ i ].created_by === 1 ) // Created by crossover.
962     {

964       crossover_operator_progress_sum += ( this.population[ i ].fitness - this.population[ i ].
        parent_fitness );

966       number_of_crossovers += 1;

968     }
        else if ( this.population[ i ].created_by === 2 ) // Created by mutation.
970     {

972       mutation_operator_progress_sum  += ( this.population[ i ].fitness - this.population[ i ].
        parent_fitness );

974       number_of_mutations += 1;

976     }

978   }

980   // Now calculate the average crossover and mutation progress for the population.

982   this.crossover_operator_progress_average = 0.0;
      this.mutation_operator_progress_average  = 0.0;
984
      if ( number_of_crossovers != 0 )
986   {

988     this.crossover_operator_progress_average = ( crossover_operator_progress_sum ) / (
        number_of_crossovers );

990   }

992   if ( number_of_mutations != 0 )
      {
994
        this.mutation_operator_progress_average  = ( mutation_operator_progress_sum ) / (
        number_of_mutations );
996
      }
998
      // Adjust crossover and mutation rate adjustments.
1000
      if ( this.best_fitness > this.worst_fitness )
1002  {

1004    this.crossover_probability_adjustment = 0.01 * ( ( this.best_fitness - this.average_fitness ) / (
        this.best_fitness - this.worst_fitness ) );

1006    this.mutation_probability_adjustment  = 0.01 * ( ( this.best_fitness - this.average_fitness ) / (
        this.best_fitness - this.worst_fitness ) );

1008  }
      else if ( this.best_fitness == this.average_fitness )
1010  {

1012    this.crossover_probability_adjustment = 0.01;

1014    this.mutation_probability_adjustment  = 0.01;

1016  }

1018  // Adjust crossover and mutation rates.

1020  if ( this.crossover_operator_progress_average > this.mutation_operator_progress_average )
      {
1022
        this.crossover_probability = this.crossover_probability + this.crossover_probability_adjustment;
1024
        this.mutation_probability  = this.mutation_probability  - this.mutation_probability_adjustment;
1026
      }
1028  else if ( this.crossover_operator_progress_average < this.mutation_operator_progress_average )
      {
1030
        this.crossover_probability = this.crossover_probability - this.crossover_probability_adjustment;
1032
```

```
      this.mutation_probability  = this.mutation_probability  + this.mutation_probability_adjustment;

    }
    else if ( this.crossover_operator_progress_average == this.mutation_operator_progress_average )
    {

      // Do not adjust.

    }

    this.crossover_probability = get_clamped_value( this.crossover_probability, this.
      crossover_probability_minimum, 1.0 );

    this.mutation_probability  = get_clamped_value( this.mutation_probability,  this.
      mutation_probability_minimum,  1.0 );

  }

  this.sort_population = function ( descending )
  {

    if ( descending == undefined || descending == false )
    {

      this.population.sort( function ( a, b ) { return a.fitness - b.fitness; } );

    }
    else
    {

      this.population.sort( function ( a, b ) { return b.fitness - a.fitness; } );

    }

  }

  this.generate_new_generation = function ( )
  {

    // Assumes population is sorted in ascending order.

    // Assumes population is completely evaluated.

    // Assumes crossover probability and mutation probability have been adjusted if using self-adaptation.

    // Create a temporary population to store newly created generation.

    var new_population = new Array( );

    // Allow the top N elite to pass into the next generation.

    this.elitism_operator( new_population );

    // Perform crossover and mutation separately?

    if ( !this.perform_crossover_and_mutation_sequentially )
    {

      // Now we enter the GA loop.

      // Repeat until a new population is generated.

      while ( new_population.length < this.population_size )
      {

        // Perform crossover and mutation separately.

        // Try to generate an offspring via crossover first.

        this.total_number_of_crossover_attempts += 1;

        // Select two genome indexes.

        var parents = this.selection_operator( 2 );

        var crossover_offspring = this.crossover_operator( parents[ 0 ], parents[ 1 ] );

        if ( crossover_offspring != 0 )
        {

          new_population.push( crossover_offspring );

          this.total_number_of_crossovers += 1;

        }

        // There is the possibility of adding up to two
        // offspring per while loop.
        // Don't create more than the population size.

        if ( new_population.length === this.population_size ) break;

        // Try to generate an offspring via mutation second.

        this.total_number_of_mutation_attempts += 1;

        // Select one genome index.

        var parent = this.selection_operator( 1 );

        var mutation_offspring = this.mutation_operator( parent[ 0 ] );
```

```
        if ( mutation_offspring != 0 )
        {

          new_population.push( mutation_offspring );

          this.total_number_of_mutations += 1;

        }

      }

      if ( new_population.length > this.population_size )
      {

        console.warn( "[Genetic_Algorithm:generate_new_generation] New population larger than population
      size." );

      }

      // Finished so assign new pop to the current population.

      this.population = [ ];
      this.population = deep_copy( new_population );
      new_population = [ ];

    }
    else // Perform crossover and mutation in sequence.
    {

      // Now we enter the GA loop.

      // Repeat until a new population is generated.

      while ( new_population.length < this.population_size )
      {

        // Attempt crossover and then mutation in sequence.

        var parents = this.selection_operator( 2 );

        var offspring = this.crossover_then_mutate_operator( parents[ 0 ], parents[ 1 ] );

        if ( offspring != 0 )
        {

          // First offspring.

          if ( offspring.one.created_by === 1 )
          {

            this.total_number_of_crossovers += 1;
            this.total_number_of_crossover_attempts += 1;
            this.total_number_of_mutation_attempts  += 1;

            new_population.push( offspring.one );

          }
          else if ( offspring.one.created_by === 2 )
          {

            this.total_number_of_mutations += 1;
            this.total_number_of_crossover_attempts += 1;
            this.total_number_of_mutation_attempts  += 1;

            new_population.push( offspring.one );

          }
          else if ( offspring.one.created_by === 3 )
          {

            this.total_number_of_crossovers += 1;
            this.total_number_of_mutations  += 1;
            this.total_number_of_crossover_attempts += 1;
            this.total_number_of_mutation_attempts  += 1;

            new_population.push( offspring.one );

          }

          // Old population size should match this new population size.

          if ( new_population.length === this.population_size ) break;

          // Second offspring.

          if ( offspring.two.created_by === 1 )
          {

            this.total_number_of_crossovers += 1;
            this.total_number_of_crossover_attempts += 1;
            this.total_number_of_mutation_attempts  += 1;

            new_population.push( offspring.two );

          }
          else if ( offspring.two.created_by === 2 )
          {

            this.total_number_of_mutations += 1;
            this.total_number_of_crossover_attempts += 1;
            this.total_number_of_mutation_attempts  += 1;
```

```
            new_population.push( offspring.two );

          }
          else if ( offspring.two.created_by === 3 )
          {

            this.total_number_of_crossovers += 1;
            this.total_number_of_mutations  += 1;
            this.total_number_of_crossover_attempts += 1;
            this.total_number_of_mutation_attempts  += 1;

            new_population.push( offspring.two );

          }

        }

      }

      if ( new_population.length > this.population_size )
      {

        console.warn( "[Genetic_Algorithm:generate_new_generation] New population larger than population size." );

      }

      // Finished so assign new pop to the current population.

      this.population = [ ];
      this.population = deep_copy( new_population );
      new_population = [ ];

    }

    // Calculate the observed rates.

    this.observed_crossover_rate = this.total_number_of_crossovers / this.total_number_of_crossover_attempts;
    this.observed_mutation_rate  = this.total_number_of_mutations  / this.total_number_of_mutation_attempts;

    // Advance generation counter.

    this.generation_number += 1;

  }

  // Getter methods.

  this.get_population = function ( )
  {

    return deep_copy( this.population );

  }

  this.get_population_size = function ( )
  {

    return deep_copy( this.population_size );

  }

  this.get_number_of_genes_per_genome = function ( )
  {

    return deep_copy( this.number_of_genes_per_genome );

  }

  this.get_genome_fitness = function ( index )
  {

    index = parseInt( index );

    if ( ( index > ( this.population_size - 1 ) ) || ( index < 0 ) )
    {

      console.error( "[Genetic_Algorithm:get_genome_fitness] Index out of bounds of population size." );

      return;

    }

    return deep_copy( this.population[ index ].fitness );

  }

  this.get_genome_genes = function ( index )
  {

    index = parseInt( index );

    if ( ( index > ( this.population_size - 1 ) ) || ( index < 0 ) )
    {

      console.error( "[Genetic_Algorithm:get_genome_genes] Index out of bounds of population size." );

      return;

    }
```

```
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    {
1422      return deep_copy( this.crossover_probability );
1424    }
1426    this.get_mutation_probability = function ( )
        {
1428
          return deep_copy( this.mutation_probability );
1430
        }
1432
        this.get_generation_number = function ( )
```

```
  {

    return deep_copy( this.generation_number );

  }

  // Setter methods.

  this.set_crossover_probability = function ( rate )
  {

    this.crossover_probability = parseFloat( rate );

  }

  this.set_mutation_probability = function ( rate )
  {

    this.mutation_probability = parseFloat( rate );

  }

  this.set_generation_number = function ( number )
  {

    number = parseInt( number );

    this.generation_number = number;

  }

  this.set_genome_fitness = function ( index, fitness )
  {

    index   = parseInt( index );
    fitness = parseFloat( fitness );

    if ( ( index > ( this.population_size - 1 ) ) || ( index < 0 ) )
    {

      console.error( "[Genetic_Algorithm:set_genome_fitness] Index out of bounds of population size." );

      return;

    }

    this.population[ index ].fitness = fitness;
  }

}
```

Listing B.3: Neural_Network.js

```
/*
 *
 * David Lettier (C) 2013.
 *
 * http://www.lettier.com/
 *
 * Code ported to JS and HEAVILY modified from original C++ source
 * found at http://www.ai-junkie.com/ann/evolved/nnt1.html and
 * written by Mat Buckland.
 *
 * Implements a neural network for learning.
 *
 */


function Neuron( nInputs )
{

  // Plus one for the bias/threshold.
  // x_1*w_1+x_2*w_2+...+x_n*w_n >= t
  // x_1*w_1+x_2*w_2+...+x_n*w_n+(-1)*t >= 0
  // Where t=w_(n+1).

  this.number_of_inputs = nInputs + 1;

  if ( this.number_of_inputs < 0 ) this.number_of_inputs = 1;

  this.weights = new Array( );

  for ( var i = 0; i < this.number_of_inputs; ++i )
  {

    this.weights.push( get_random_float( -1.0, 1.0 ) );

  }

}

function Neuron_Layer( nNumberNeurons, nNumberInputsPerNeuron )
{

  this.number_of_neurons = nNumberNeurons;

  this.neurons = new Array( );

  for ( var i = 0; i < nNumberNeurons; ++i )
  {
```

```
    this.neurons.push( new Neuron( nNumberInputsPerNeuron ) );
  }
}
function Neural_Net( params )
{
  this.number_of_inputs = params.nInputs;
  this.number_of_outputs = params.nOutputs;
  this.number_of_hidden_layers = params.nHiddenLayers;
  this.neurons_per_hidden_layer = params.nNeuronsPerHiddenLayer;
  this.bias = params.bias;
  this.layers = new Array( );
  this.create_network = function ( )
  {
    // Create the layers of the network.
    if ( this.number_of_hidden_layers > 0 )
    {
      // Create first hidden layer.
      this.layers.push( new Neuron_Layer( this.neurons_per_hidden_layer, this.number_of_inputs ) );
      // Create the subsequent hidden layers.
      for ( var i = 0; i < this.number_of_hidden_layers - 1; ++i )
      {
        this.layers.push( new Neuron_Layer( this.neurons_per_hidden_layer, this.neurons_per_hidden_layer )
       );
      }
      // Create output layer.
      this.layers.push( new Neuron_Layer( this.number_of_outputs, this.neurons_per_hidden_layer ) );
    }
    else
    {
      // Create output layer.
      this.layers.push( new Neuron_Layer( this.number_of_outputs, this.number_of_inputs ) );
    }
  }
  this.create_network( );
  this.get_weights = function ( )
  {
    var weights = new Array( );
    // For each layer.
    for ( var i = 0; i < this.number_of_hidden_layers + 1; ++i ) // Plus one for the output layer.
    {
      // For each neuron.
      for ( var j = 0; j < this.layers[ i ].number_of_neurons; ++j )
      {
        // For each weight.
        for ( var k = 0; k < this.layers[ i ].neurons[ j ].number_of_inputs; ++k )
        {
          weights.push( this.layers[ i ].neurons[ j ].weights[ k ] );
        }
      }
    }
    return weights;
  }
  this.get_number_of_weights = function ( )
  {
    var weights = 0;
    // For each layer.
    for ( var i = 0; i < this.number_of_hidden_layers + 1; ++i ) // Plus one for the output layer.
    {
      // For each neuron.
      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         {
157           weights += 1;
159         }
161       }
163     }
165     return weights;
167   }
169   this.put_weights = function ( weights )
171   {
173     var weight = 0;
175     // For each layer.
177     for ( var i = 0; i < this.number_of_hidden_layers + 1; ++i ) // Plus one for the output layer.
179     {
181       // For each neuron.
183       for ( var j = 0; j < this.layers[ i ].number_of_neurons; ++j )
185       {
187         // For each weight.
189         for ( var k = 0; k < this.layers[ i ].neurons[ j ].number_of_inputs; ++k )
191         {
193           this.layers[ i ].neurons[ j ].weights[ k ] = weights[ weight++ ];
195         }
197       }
199     }
201     return null;
203   }
205   this.update = function ( inputs )
207   {
209     // Stores the resultant outputs from each layer.
211     var outputs = new Array( );
213     var weight = 0;
215     // First check that we have the correct amount of inputs.
217     if ( inputs.length != this.number_of_inputs )
219     {
221       console.error( "[Neural_Net:update] Passed inputs length does not equal neural network number of
          inputs setting." );
223       // Just return an empty vector if incorrect.
225       return outputs;
227     }
229     // For each layer.
231     for ( var i = 0; i < this.number_of_hidden_layers + 1; ++i )
233     {
235       if ( i > 0 )
237       {
239         inputs = deep_copy( outputs ); // Deep copy outputs.
241       }
243       outputs.length = 0;
245       weight = 0;
247       // For each neuron sum the ( inputs * corresponding weights ).
249       // Throw the total at our sigmoid function to get the output.
       for ( var j = 0; j < this.layers[ i ].number_of_neurons; ++j )
       {
         var net_input = 0.0;
         var number_of_inputs = this.layers[ i ].neurons[ j ].number_of_inputs;
         // For each weight.
         for ( var k = 0; k < number_of_inputs - 1; ++k )
         {
           // Sum the weights x inputs.
           net_input += this.layers[ i ].neurons[ j ].weights[ k ] * inputs[ weight++ ];
         }
```

```
        // Add in the bias/threshold.
        // x_1*w_1+x_2*w_2+...+x_n*w_n >= t
        // x_1*w_1+x_2*w_2+...+x_n*w_n+(-1)*t >= 0
        // Where t=w_(n+1).

        net_input += this.bias * this.layers[ i ].neurons[ j ].weights[ number_of_inputs - 1 ];

        // We can store the outputs from each layer as we generate them.
        // The combined activation is first filtered through the Sigmoid function.

        outputs.push( this.sigmoid( net_input ) );

        weight = 0;
      }
    }

    return outputs;

  }

  this.sigmoid = function ( input )
  {

    // Use the hyperbolic tangent function to get a range of outputs [-1,1].

    var math_exp    = Math.exp;

    var numerator   = math_exp( 2 * input ) - 1;
    var denominator = math_exp( 2 * input ) + 1;

    return numerator / denominator;

    /*

    Old way.


    // Returns [ -1, 1 ].

    var numerator   = 1.0;
    var denominator = 1.0 + Math.pow( Math.E, ( -input / response ) );

    return ( ( numerator / denominator ) - 0.5 ) * 2.0;

    */

  }

}
```

B.2 BBAutoTune

Listing B.4: GA.py

```
#! /usr/bin/env python

'''

David Lettier (C) 2014.

http://www.lettier.com/

The main GA python file.

'''

import sys;
import atexit;
import random;
import math;
import copy;
import signal;
import time;
import itertools;
import subprocess;
import mysql.connector
import os;
import bpy;
import pickle;
import scipy;
import numpy;
import sklearn.covariance;
from bpy.props import *;

'''

Helper functions.

'''

def get_clamped_value( value, minimum, maximum ):

  return max( min( maximum, value ), minimum );

def get_scripts_location( ):

  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
67    );
      bpy.context.scene[ "GA_POPULATION_SIZE" ] = 10;
69
      bpy.types.Scene.GA_MAX_GENERATIONS = IntProperty(
71
73      name = "Max Generations",
74      description = "Max generations.",
75      min = 0
77    );
      bpy.context.scene[ "GA_MAX_GENERATIONS" ] = 100;
79    bpy.types.Scene.GA_NUMBER_OF_ELITE = IntProperty(
81      name = "Number of Elite",
82      description = "Number of elite.",
83      min = 0
85    );
      bpy.context.scene[ "GA_NUMBER_OF_ELITE" ] = 2;
87
      bpy.types.Scene.GA_CROSSOVER_PROBABILITY = FloatProperty(
89
91      name        = "Crossover Probability",
92      description = "Crossover probability.",
93      default     = 0.8,
94      min         = 0.0,
95      max         = 1.0
97    );
      bpy.types.Scene.GA_MUTATION_PROBABILITY = FloatProperty(
99
101     name        = "Mutation Probability",
102     description = "Mutation probability.",
103     default     = 0.2,
104     min         = 0.0,
105     max         = 1.0
107   );
      bpy.types.Scene.GA_MAX_TORQUE = FloatProperty(
109
111     name        = "Max Torque",
112     description = "Maximum torque value possible during search.",
113     default     = 50.0,
114     min         = 0.0,
115     max         = 340282346638528859811704183484516925440.0
117   );
      bpy.types.Scene.GA_USE_RANK_FITNESS_SELECTION = BoolProperty(
119
121     name        = "Use Rank Fitness Selection",
122     description = "Use rank fitness selection otherwise tournament selection will be used."
123   );
      bpy.context.scene[ "GA_USE_RANK_FITNESS_SELECTION" ] = False;
125
      bpy.types.Scene.GA_PERFORM_CROSSOVER_AND_MUTATION_SEQUENTIALLY = BoolProperty(
127
129     name        = "Perform Crossover and Mutation Sequentially",
130     description = "Perform crossover and mutation sequentially."
131   );
      bpy.context.scene[ "GA_PERFORM_CROSSOVER_AND_MUTATION_SEQUENTIALLY" ] = False;
133
      bpy.types.Scene.GA_USE_SELF_ADAPTATION = BoolProperty(
135
137     name        = "Use Self-adaptation",
138     description = "Adapt the crossover and mutation probabilities."
139   );
      bpy.context.scene[ "GA_USE_SELF_ADAPTATION" ] = False;
141
      bpy.types.Scene.BBAUTOTUNE_OPEN_GA_MONITOR_BROWSER_WINDOW = BoolProperty(
143
145     name        = "Open GA Monitor Browser Window",
146     description = "Open a browser window to the GA monitor."
```

```
  );
  bpy.context.scene[ "BBAUTOTUNE_OPEN_GA_MONITOR_BROWSER_WINDOW" ] = False;

  bpy.types.Scene.BBAUTOTUNE_DEBUG = BoolProperty(

    name        = "Debug",
    description = "Log debug information."

  );
  bpy.context.scene[ "BBAUTOTUNE_DEBUG" ] = False;

initialize_bbautotune_parameter_properties( );

'''

The start button operator on the BBAutoTune UI panel.
Starts BBAutoTune with the values from the UI panel properties.

'''

class BBAUTOTUNE_UI_START_BUTTON_OPERATOR( bpy.types.Operator ):

  bl_idname = "bbautotune.start";
  bl_label  = "Start"

  def execute( self, context ):

    bpy.bbautotune.start(

      bpy.context.scene.GA_POPULATION_SIZE,
      bpy.context.scene.GA_MAX_GENERATIONS,
      bpy.context.scene.GA_NUMBER_OF_ELITE,
      bpy.context.scene.GA_CROSSOVER_PROBABILITY,
      bpy.context.scene.GA_MUTATION_PROBABILITY,
      bpy.context.scene.GA_MAX_TORQUE,
      bpy.context.scene.GA_USE_RANK_FITNESS_SELECTION,
      bpy.context.scene.GA_PERFORM_CROSSOVER_AND_MUTATION_SEQUENTIALLY,
      bpy.context.scene.GA_USE_SELF_ADAPTATION,
      bpy.context.scene.BBAUTOTUNE_OPEN_GA_MONITOR_BROWSER_WINDOW,
      bpy.context.scene.BBAUTOTUNE_DEBUG

    );

    # Clean up on premature exit.

    atexit.register( bpy.bbautotune.stop );

    return { 'FINISHED' };

'''

The BBAutoTune UI panel and its layout in Blender.

'''

class GA_UI_PANEL( bpy.types.Panel ):

  bl_label       = "BBAutoTune Parameters";
  bl_space_type  = "PROPERTIES";
  bl_region_type = "WINDOW";
  bl_context     = "render";

  def draw( self, context ):

    self.layout.prop( context.scene, "GA_POPULATION_SIZE"                           );
    self.layout.prop( context.scene, "GA_MAX_GENERATIONS"                           );
    self.layout.prop( context.scene, "GA_NUMBER_OF_ELITE"                           );
    self.layout.prop( context.scene, "GA_CROSSOVER_PROBABILITY"                     );
    self.layout.prop( context.scene, "GA_MUTATION_PROBABILITY"                      );
    self.layout.prop( context.scene, "GA_MAX_TORQUE"                                );
    self.layout.prop( context.scene, "GA_USE_RANK_FITNESS_SELECTION"                );
    self.layout.prop( context.scene, "GA_PERFORM_CROSSOVER_AND_MUTATION_SEQUENTIALLY" );
    self.layout.prop( context.scene, "GA_USE_SELF_ADAPTATION"                       );
    self.layout.prop( context.scene, "BBAUTOTUNE_OPEN_GA_MONITOR_BROWSER_WINDOW"    );
    self.layout.prop( context.scene, "BBAUTOTUNE_DEBUG"                             );

    self.layout.operator( "bbautotune.start" );

'''

A single genome.

'''

class Genome( ):

  new_id = itertools.count( ).__next__;

  def __init__( self, genes = None, fitness = None ):

    self.id = Genome.new_id( );

    if ( not genes == None ):

      self.genes = list( genes );

    else:

      self.genes = [ ];

    self.fitness = fitness or 0.0;
```

```
    # Used to calculate either the crossover progress or mutation progress.
    # If this genome is created via crossover, use the weighted average
    # based on the cross over point.
    # So if the crossover point is say 9 and the genome length is 10,
    # then the weighted average pf = (p1.f*.9) + (p2.f*.1).
    # In other words the offspring received 90% of its genes from parent one
    # and it received 10% of its genes from parent two so its parent fitness is
    # 90% of parent one's fitness and 10% of parent two's fitness.

    self.parent_fitness = 0.0;

    # Created by means if this genome was generated either by randomness, crossover,
    # mutation, both crossover and mutation, or elitism.
    # Initially it is created from nothing so set it to -1.
    # 0 = randomness, 1 = crossover, 2 = mutation, 3 = crossover & mutation, 4 = elitism

    # This encoding is to facilitate crossover's and mutation's progress at producing fitter
    # offspring than the offspring's parents.

    self.created_by = 0;

  def set_genes( self, genes = None ):

    if ( not genes == None ):

      self.genes = list( genes );

    else:

      self.genes = [ ];

  def get_genes( self ):

    return list( self.genes );

  def get_genes_as_string( self ):

    return ",".join( map( str, self.genes ) );

  def set_fitness( self, fitness = None ):

    self.fitness = fitness or 0.0;

  def get_fitness( self ):

    return self.fitness;

  def set_parent_fitness( self, parent_fitness = None ):

    self.parent_fitness = parent_fitness or 0.0;

  def get_parent_fitness( self ):

    return self.parent_fitness;

  def set_created_by( self, created_by = None ):

    self.created_by = created_by or 0;

  def get_created_by( self ):

    return self.created_by;

  def __repr__( self ):

    return repr( ( self.id, self.created_by, self.fitness, self.parent_fitness, self.genes ) );

'''

The main genetic algorithm object.

'''

class Genetic_Algorithm( ):

  def __init__(

    self,
    population_size                             = None,
    max_generations                             = None,
    number_of_elite                             = None,
    crossover_probability                       = None,
    mutation_probability                        = None,
    use_rank_fitness_selection                  = None,
    perform_crossover_and_mutation_sequentially = None,
    use_self_adaptation                         = None

  ):

    '''

    Begin parameters.

    '''

    # Size of population.

    self.population_size = population_size or 0;

    # Number of generations to run until termination of the algorithm.

```

```
    self.max_generations = max_generations or 0;

    # Set the number of elite that go on to the next generation.

    self.number_of_elite = number_of_elite or 0;

    # Probability of genome's crossing over bits.
    # 0.7 is pretty good.

    self.crossover_probability               = crossover_probability or 0.0;
    self.crossover_probability_minimum       = 0.001;
    self.crossover_probability_adjustment    = 0.01;
    self.crossover_operator_progress_average = 0.0;
    self.observed_crossover_rate             = 0.0;
    self.total_number_of_crossovers          = 0;
    self.total_number_of_crossover_attempts  = 0;

    # Probability that a genomes bits will mutate.
    # Try figures around 0.05 to 0.3-ish.

    self.mutation_probability               = mutation_probability or 0.0;
    self.mutation_probability_minimum       = 0.001;
    self.mutation_probability_adjustment    = 0.01;
    self.mutation_operator_progress_average = 0.0;
    self.observed_mutation_rate             = 0.0;
    self.total_number_of_mutations          = 0;
    self.total_number_of_mutation_attempts  = 0;

    # Use rank fitness selection?

    self.use_rank_fitness_selection = use_rank_fitness_selection or False;

    # Perform crossover and mutation sequentially or separately?

    self.perform_crossover_and_mutation_sequentially = perform_crossover_and_mutation_sequentially or
      False;

    # Use self-adaptation?

    self.use_self_adaptation = use_self_adaptation or False;

    '''

    End parameters.

    '''

    # Log file if debugging.

    self.log_file_name = "";

    # The amount of genes per genome.

    self.number_of_genes_per_genome = 18;

    # This holds the entire population of genomes.

    self.population = [ ];

    # Total fitness of population.

    self.total_fitness = 0.0;

    # Average fitness.

    self.average_fitness = 0.0;

    # Highest fitness.

    self.highest_fitness = 0.0;

    # Lowest fitness.

    self.lowest_fitness = 0.0;

    # Keeps track of the best genome.

    self.fittest_genome_index = -1;

    # Keep track of the worst genome.

    self.weakest_genome_index = -1;

    # Generation number.

    self.generation_number = 0;

    # Current population makeup of randoms, crossovers, mutants, crossover mutants, and elites.

    self.population_makeup = "";

  def set_population_size( self, size = None ):

    self.population_size = size or 0;

  def get_population_size( self ):

    return self.population_size;

  def set_max_generations( self, maximum = None ):

    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_fittest_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[ i ].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     # Assumes replacement_population_genes is one big array.
        # Splices the big array based on the number of genes per genome.
669
        # Big array:  [ 1,2,3,4,5,6,7,8,9,10 ]
671     # Number of genes per genome: 2
        # Population: [ [ 1,  2 ] G0
673     #               [ 3,  4 ] G1
        #               [ 5,  6 ] ...
675     #               [ 7,  8 ] ...
        #               [ 9, 10 ] GN-1
677     #             ]

679     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,
          "Too few or too many replacement genes."

683     k = 0;

685     for i in range( self.population_size ):

687       self.population[ i ].genes = [ ];

689       for j in range( self.number_of_genes_per_genome ):

691         self.population[ i ].genes.append( replacement_population_genes[ k ] );

693         k += 1;

695   def selection_operator( self, number_of_indexes ):

697     self.log( "Entering selection operator." );

699     self.log( str( number_of_indexes ) + " genomes requested." );

701     # Assumes the population has been evaluated.

703     # Assumes the population is sorted in descending order according to fitness.

705     if ( not self.use_rank_fitness_selection ):

707       tournament_size = number_of_indexes + 1;

709       assert tournament_size <= self.population_size and tournament_size >= 0, "Tournament size too large/
          small.";

711       genome_indexes_selected = [ ];

713       self.log( "Selecting random players." );

715       for i in range( number_of_indexes ):

717         tournament_players = [ ];

719         for j in range( tournament_size ):

721           random_int = random.randint( 0, self.population_size - 1 );

723           tournament_players.append( [ random_int, self.population[ random_int ].get_fitness( ) ] );

725           self.log( "Random players selected." );

727           self.log( str( tournament_players ) );

729         # Remember, lower fitness values are a higher fitness.
            # Sorts list in ascending orderer.
731         # tournament_players = [ [genome_index,genome_fitness], ... ]
            # [ 0 ][ 0 ] = get the index with the lowest fitness value (thus the highest fitness ).
733
            player_ranking = sorted( tournament_players, key = lambda x: x[ 1 ] );
735
            self.log( "Player ranking." );
737
            self.log( str( player_ranking ) );
739
            winner = player_ranking[ 0 ][ 0 ];
741
            self.log( "Winner." );
743
            self.log( str( winner ) );
745
            genome_indexes_selected.append( winner );
747
          self.log( "Selected genomes with tournament selection." );
749
          self.log( str( genome_indexes_selected ) );
751
          return genome_indexes_selected;
753
        else:
```

```
755
        self.log( "Sorted?" );
757
        self.log( str( self.population[ 0 ].get_fitness( ) ) + " " + str( self.population[ -1 ].get_fitness(
         ) ) );
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.
        # ...
765     # Give the best genome a rank fitness of the population size.

767     genome_indexes_selected = [ ];

769     # Genomes:               1, 2, 3,  4
        # Genomes fitnesses:     4, 2, 3,  1 (Lowest is highest. )
771     # Rank fitnesses:        1, 2, 3,  4 (Highest is highest.)
        # Partial sums:          1, 3, 6, 10
773     # Random number U(0,10): 7
        # Genome 4 is selected.
775     # Since all random numbers are uniform,
        # 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%
        # 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%
        # Total of any being selected:                                = 100%
781     #
        #
783     # 0.10 ---------------------------------------
        #      |   |       |           |               |
785     #      |   |       |           |               |
        #      0---1---2---3---4---5---6---7---8---9---10
787     #          G1      G2          G3              G4

789     # Individual rank fitnesses.

791     rank_fitnesses = [ ];

793     for i in range( self.population_size ):

795       rank_fitnesses.append( i + 1 );

797     total_rank_fitness = sum( rank_fitnesses );

799     self.log( "Total rank fitness." );

801     self.log( str( total_rank_fitness ) );

803     # Partial sum. P[i] = sum( P[1:i] ) where i is in range [1,n].

805     partial_sums = list( itertools.accumulate( rank_fitnesses ) );

807     self.log( "Partial sums." );

809     self.log( str( partial_sums ) );

811     while ( len( genome_indexes_selected ) < number_of_indexes ):

813       random_number = random.randint( 0, total_rank_fitness );

815       self.log( "Random number: " + str( random_number ) );

817       for i in range( self.population_size ):

819         if ( partial_sums[ i ] >= random_number ):

821           self.log( "Partial sum: " + str( partial_sums[ i ] ) );

823           self.log( "Genome index selected: " + str( i ) );

825           genome_indexes_selected.append( i );

827           break;

829         if ( len( genome_indexes_selected ) == number_of_indexes ):

831           break;

833     self.log( "Selected genomes with rank selection." );

835     self.log( str( genome_indexes_selected ) );

837     return genome_indexes_selected;

839   def elitism_operator( self, new_population ):

841     self.log( "Attempting to add elite genomes." );

843     if ( self.number_of_elite > self.population_size ):

845       self.number_of_elite = self.population_size;

847     # Assumes the population is sorted in ascending order of fitness.

849     # A = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
        # |A| = 10
851     # i = 2 check.
        # i = 1 decrement.
853     # A[ ( ( 10 - 1 = 9 ) - i ) = 8 ]
        # i = 1 check.
855     # i = 0 decrement.
```

```
      # A[ ( ( 10 - 1 = 9 ) - i ) = 9 ]
      # i = 0 check.
      # Stop.

      i = self.number_of_elite;

      while ( i ):

        i -= 1;

        genome_temp = copy.deepcopy( self.population[ ( self.population_size - 1 ) - i ] );

        self.log( "Adding elite genome. Its fitness: " + str( genome_temp.get_fitness( ) ) );

        genome_temp.fitness        = 0.0;
        genome_temp.parent_fitness = 0.0;
        genome_temp.created_by     = 4;

        new_population.append( genome_temp );

        if ( len( new_population ) == self.population_size ):

          break;

    def crossover_operator( self, parent_one_index, parent_two_index ):

      # One point crossover operator.

      # Only returns one crossed offspring.

      # Do we crossover?

      self.log( "Attempting crossover." );

      if ( random.uniform( 0.0, 1.0 ) <= self.crossover_probability ):

        # If the parents are the same genome then this is not a true crossover.

        if ( parent_one_index == parent_two_index ):

          self.log( "Parents are the same. Crossover failed." );

          return 0;

        self.log( "Parent indexes." );

        self.log( str( parent_one_index ) + " " + str( parent_two_index ) );

        # Create a blank offspring.

        offspring = Genome( );

        offspring.genes          = [ ];
        offspring.fitness        = 0.0;
        offspring.parent_fitness = 0.0;
        offspring.created_by     = 0;

        # Determine a crossover point.

        # Let the uniform sample be in the range of [1,n-1].
        # If the crossover point was zero than no true crossover takes place
        # as all of one parent's genes get copied into the offspring.
        # If the cp = n-1 then at least you get n-1 from one parent and 1
        # from another parent.

        crossover_point = random.randint( 1, ( self.number_of_genes_per_genome - 1 ) );

        self.log( "Crossover point." );

        self.log( str( crossover_point ) );

        # Cross the parent's genes into the offspring's genes.

        for i in range( crossover_point ):

          offspring.genes.append( copy.deepcopy( self.population[ parent_one_index ].genes[ i ] ) );

        for i in range( crossover_point, self.number_of_genes_per_genome ):

          offspring.genes.append( copy.deepcopy( self.population[ parent_two_index ].genes[ i ] ) );

        # Determine if a crossover actually took place.
        # The offspring should not match the parent one's genes and
        # it should not match parent two's genes as the offspring
        # should be a combination of the two.

        if (    ( offspring.genes != self.population[ parent_one_index ].genes ) and
          ( offspring.genes != self.population[ parent_two_index ].genes )     ):

          # Weighted average fitness of the parents based on crossover point
          # determining percentage of genes received from parent one and parent two.
          # Let the number of genes per genome be 41 and let the crossover point be 1.
          # Offspring gets [0,1) = 1 gene from parent one and [1,41) = 40 genes from parent two.
          # PF = PF1 * (1/41) + PF2 * ((41-1)/41).

          parent_one_contribution = ( self.population[ parent_one_index ].fitness * ( (
                      crossover_point ) / ( self.number_of_genes_per_genome ) ) );
          parent_two_contribution = ( self.population[ parent_two_index ].fitness * ( ( self.
        number_of_genes_per_genome - crossover_point ) / ( self.number_of_genes_per_genome ) ) );

          offspring.parent_fitness = parent_one_contribution + parent_two_contribution;
```

```
        self.log( "Parent fitnesses." );
        self.log( str( self.population[ parent_one_index ].fitness ) );
        self.log( str( self.population[ parent_two_index ].fitness ) );
        self.log( "Parent fitness contributions." );
        self.log( str( parent_one_contribution ) );
        self.log( str( parent_two_contribution ) );
        offspring.created_by = 1;
        self.log( "Returning crossed offspring." );
        return offspring;
      else:
        self.log( "Did not actually perform crossover. Offspring genes match parents." );
        return 0;
    else:
      self.log( "Greater than crossover probability." );
      return 0;
  def mutation_operator( self, parent_index ):
    # Mutates parent genome's genes on a whole genome basis based on the mutation probability.
    # Gaussian distribution mutation.
    # Do we mutate?
    self.log( "Attempting mutation." );
    if ( random.uniform( 0.0, 1.0 ) <= self.mutation_probability ):
      # Create a blank offspring and fill its genes with the parent's genes.
      offspring                = Genome( );
      offspring.genes          = [ ];
      offspring.genes          = copy.deepcopy( self.population[ parent_index ].genes );
      offspring.fitness        = 0.0;
      offspring.parent_fitness = 0.0;
      offspring.created_by     = 0;
      # Begin to mutate.
      mutated = False;
      for i in range( self.number_of_genes_per_genome ):
        # Mutate this gene by sampling a value from a normal distribution
        # where the mean is the current gene value and the standard deviation
        # is the mutation step equal to the mutation probability in the range [0,1].
        # A low mutation probability will give a mutated gene value closer to the original gene
        # value (the mean) (most of the time) as the standard deviation is small and therefore the
      mutation step is small.
        # A high mutation probability will give a mutated gene value farther from the original gene
        # value (the mean) as the standard deviation is large and therefore the mutation step is large.
        # Note that gv = gv +   *N(0,1) is the same as gv = N(gv,  ).
        # Clamp the gene to range [-1,1].
        temp_gene_value = copy.deepcopy( offspring.genes[ i ] );
        offspring.genes[ i ] = random.gauss( offspring.genes[ i ], self.mutation_probability );
        offspring.genes[ i ] = get_clamped_value( offspring.genes[ i ], 0.0, 1.0 );
        self.log( "Mutation value before/after." );
        self.log( str( temp_gene_value ) + " " + str( offspring.genes[ i ] ) );
        # Test if it was truly mutated.
        if ( temp_gene_value != offspring.genes[ i ] ):
          mutated = True;
      if ( mutated ): # If truly mutated.
        offspring.parent_fitness = self.population[ parent_index ].fitness;
        offspring.created_by = 2;
        self.log( "Returning mutated offspring." );
        return offspring;
      else:
        self.log( "Mutation did not actually take place." );
        return 0;
    else:
```

```
1057
            self.log( "Greater than mutation probability." );
1059
            return 0;
1061
        def crossover_then_mutate_operator( self, parent_one_index, parent_two_index ):
1063
          # Crossover and mutation done sequentially as in more traditional genetic algorithms.
1065
          # First attempts crossover and then attempts mutation.
1067
          self.log( "Attempting crossover and then mutation." );
1069
          offspring_one = copy.deepcopy( self.population[ parent_one_index ] );
1071      offspring_two = copy.deepcopy( self.population[ parent_one_index ] );

1073      offspring_one.fitness = 0.0;
          offspring_two.fitness = 0.0;
1075
          offspring_one.parent_fitness = 0.0;
1077      offspring_two.parent_fitness = 0.0;

1079      offspring_one.created_by = 0;
          offspring_two.created_by = 0;
1081
          self.log( "Attempting crossover first." );
1083
          # Attempt crossover.
1085
          crossover_point = random.randint( 0, ( self.number_of_genes_per_genome - 1 ) );
1087
          self.log( "Crossover point." );
1089
          self.log( str( crossover_point ) );
1091
          if ( ( random.uniform( 0.0, 1.0 ) <= self.crossover_probability ) and ( parent_one_index !=
            parent_two_index ) and ( crossover_point != 0 ) ):
1093
            # Cross the parent's genes in the offspring.
1095
            offspring_one.genes = [ ];
1097        offspring_two.genes = [ ];

1099        for i in range( crossover_point ):

1101          offspring_one.genes.append( copy.deepcopy( self.population[ parent_one_index ].genes[ i ] ) );
              offspring_two.genes.append( copy.deepcopy( self.population[ parent_two_index ].genes[ i ] ) );
1103
            for i in range( crossover_point, self.number_of_genes_per_genome ):
1105
              offspring_one.genes.append( copy.deepcopy( self.population[ parent_two_index ].genes[ i ] ) );
1107          offspring_two.genes.append( copy.deepcopy( self.population[ parent_one_index ].genes[ i ] ) );

1109        parent_one_contribution = ( self.population[ parent_one_index ].fitness * ( (
                      crossover_point ) / ( self.number_of_genes_per_genome ) ) );
            parent_two_contribution = ( self.population[ parent_two_index ].fitness * ( ( self.
            number_of_genes_per_genome - crossover_point ) / ( self.number_of_genes_per_genome ) ) );
1111
            offspring_one.parent_fitness = parent_one_contribution + parent_two_contribution;
1113        offspring_one.created_by     = offspring_one.created_by + 1;

1115        parent_two_contribution = ( self.population[ parent_two_index ].fitness * ( (
                      crossover_point ) / ( self.number_of_genes_per_genome ) ) );
            parent_one_contribution = ( self.population[ parent_one_index ].fitness * ( ( self.
            number_of_genes_per_genome - crossover_point ) / ( self.number_of_genes_per_genome ) ) );
1117
            offspring_two.parent_fitness = parent_two_contribution  + parent_one_contribution;
1119        offspring_two.created_by     = offspring_two.created_by + 1;

1121        self.log( "Crossover complete." );

1123        self.log( str( offspring_one.created_by ) + " " + str( offspring_two.created_by ) );

1125      else:

1127        self.log( "Crossover failed." );

1129      # Crossover may or may not have happened but now try mutation.

1131      self.log( "Attempting mutation second." );

1133      # Attempt to mutate offspring one.

1135      self.log( "Attempting to mutate the first offspring." );

1137      mutated_one = False;

1139      for i in range( self.number_of_genes_per_genome ):

1141        self.log( "Attempting to mutate gene: " + str( i ) );

1143        # Mutate this gene by sampling a value from a normal distribution where the mean
          # is the current gene value and the standard deviation is mutation step equal to the
1145        # mutation probability in the range [0,1]. A low mutation probability will give a mutated gene
          # value close to the original gene value (the mean) (most of the time) as the standard
1147        # deviation is small and therefore the mutation step is small. A high mutation probability
          # will give, or it can more easily, mutate a gene value farther from the original gene value
1149        # (the mean) as the standard deviation is large and therefore the mutation step is large.

1151        # Note that gv = gv +    * N( 0, 1 ) is the same as gv = N( gv,    ).

1153        # Clamp the gene to range [0,1].
```

```
1155        if ( random.uniform( 0.0, 1.0 ) <= self.mutation_probability ): # Mutate this gene?
1157          temp_gene_value_one = copy.deepcopy( offspring_one.genes[ i ] );
1159          offspring_one.genes[ i ] = random.gauss( offspring_one.genes[ i ], self.mutation_probability );
              # offspring_one.genes[ i ] = gaussian_distribution( offspring_one.genes[ i ], 0.5 );
1161          # offspring_one.genes[ i ] = offspring_one.genes[ i ] + ( get_random_float( -1.0, 1.0 ) * .3 );
1163          offspring_one.genes[ i ] = get_clamped_value( offspring_one.genes[ i ], 0.0, 1.0 );
1165          self.log( "Mutation value before/after." );
1167          self.log( str( temp_gene_value_one ) + " " + str( offspring_one.genes[ i ] ) );
1169          # Test if it was truly mutated.
1171          if ( temp_gene_value_one != offspring_one.genes[ i ] ):
1173            mutated_one = True;
1175        else:
1177          self.log( "Greater than mutation probability." );
1179      # Attempt to mutate offspring two.
1181      self.log( "Attempting to mutate the second offspring." );
1183      mutated_two = False;
1185      for i in range( self.number_of_genes_per_genome ):
1187        self.log( "Attempting to mutate gene: " + str( i ) );
1189        # Mutate this gene by sampling a value from a normal distribution where the mean
          # is the current gene value and the standard deviation is mutation step equal to the
1191        # mutation probability in the range [0,1]. A low mutation probability will give a mutated gene
          # value close to the original gene value (the mean) (most of the time) as the standard
1193        # deviation is small and therefore the mutation step is small. A high mutation probability
          # will give (or it can more easily) mutate a gene value farther from the original gene value
1195        # (the mean) as the standard deviation is large and therefore the mutation step is large.
1197        # Note that gv = gv +    * N( 0, 1 ) is the same as gv = N( gv,    ).
1199        # Clamp the gene to range [0,1].
1201        if ( random.uniform( 0.0, 1.0 ) <= self.mutation_probability ): # Mutate this gene?
1203          temp_gene_value_two = copy.deepcopy( offspring_two.genes[ i ] );
1205          offspring_two.genes[ i ] = random.gauss( offspring_two.genes[ i ], self.mutation_probability );
              # offspring_two.genes[ i ] = gaussian_distribution( offspring_two.genes[ i ], 0.5 );
1207          # offspring_two.genes[ i ] = offspring_two.genes[ i ] + ( get_random_float( -1.0, 1.0 ) * .3 );
1209          offspring_two.genes[ i ] = get_clamped_value( offspring_two.genes[ i ], 0.0, 1.0 );
1211          self.log( "Mutation value before/after." );
1213          self.log( str( temp_gene_value_two ) + " " + str( offspring_one.genes[ i ] ) );
1215          # Test if it was truly mutated.
1217          if ( temp_gene_value_two != offspring_two.genes[ i ] ):
1219            mutated_two = True;
1221        else:
1223          self.log( "Greater than mutation probability." );
1225      if ( mutated_one ): # If truly mutated.
1227        # Mutation = 2, crossover = 1, crossover + mutation = 3.
1229        offspring_one.created_by = offspring_one.created_by + 2;
1231        # If this offspring was only mutated, that is, it was not crossed then it gets its parent fitness.
          # If it was crossed before being mutated then offspring_one.created_by would equal 3.
1233
1235        if ( offspring_one.created_by == 2 ):
1237          self.log( "Offspring one only mutated." );
1239          offspring_one.parent_fitness = copy.deepcopy( self.population[ parent_one_index ].fitness );
1241      if ( mutated_two ): # If truly mutated.
1243        offspring_two.created_by = offspring_two.created_by + 2;
1245        # If this offspring was only mutated, that is, it was not crossed then it gets its parent fitness.
          # If it was crossed before being mutated then offspring_two.created_by would equal 3.
1247        if ( offspring_two.created_by == 2 ):
1249          self.log( "Offspring two only mutated." );
1251          offspring_two.parent_fitness = copy.deepcopy( self.population[ parent_two_index ].fitness );
1253      # No parents->offspring not crossed and/or not mutated enter into the new population.
          # Each offspring going into the new population must either be crossed, mutated, or both.
1255
```

```
    if ( ( offspring_one.created_by == 0 ) or ( offspring_two.created_by == 0 ) ):

      self.log( "No crossover and/or mutation occurred." );

      return 0;

    elif ( ( offspring_one.genes == self.population[ parent_one_index ].genes ) or ( offspring_two.genes
      == self.population[ parent_two_index ].genes ) ):

      self.log( "Offspring genes not different from parents." );

      return 0;

    else:

      self.log( "Returning offspring." );

      return { "one": offspring_one, "two": offspring_two };

  def reset_population_metrics( self ):

    self.total_fitness         =  0.0;
    self.highest_fitness       =  0.0;
    self.lowest_fitness        =  0.0;
    self.average_fitness       =  0.0;
    self.fittest_genome_index  = -1;
    self.weakest_genome_index  = -1;

  def update_population_metrics( self ):

    # 0.0 is the highest fitness
    # 1.7976931348623157e+308 is lowest fitness.

    self.reset_population_metrics( );

    highest_so_far = self.population[ 0 ].fitness;
    lowest_so_far  = self.population[ 0 ].fitness;

    self.fittest_genome_index = 0;
    self.weakest_genome_index = 0;

    self.total_fitness   = self.population[ 0 ].fitness;
    self.highest_fitness = self.population[ 0 ].fitness;
    self.lowest_fitness  = self.population[ 0 ].fitness;

    for i in range( self.population_size ):

      # Update fittest if necessary.

      if ( highest_so_far > self.population[ i ].fitness ):

        highest_so_far = self.population[ i ].fitness;

        self.fittest_genome_index = i;

        self.highest_fitness = highest_so_far;

      # Update worst if necessary.

      if ( lowest_so_far < self.population[ i ].fitness ):

        lowest_so_far = self.population[ i ].fitness;

        self.weakest_genome_index = i;

        self.lowest_fitness = lowest_so_far;

      self.total_fitness += self.population[ i ].fitness;

      # Next genome.

    self.average_fitness = self.total_fitness / float( self.population_size );

    self.log( "Population metrics. T H L A Fi Wi." );

    self.log( str( self.total_fitness        ) );
    self.log( str( self.highest_fitness      ) );
    self.log( str( self.lowest_fitness       ) );
    self.log( str( self.average_fitness      ) );
    self.log( str( self.fittest_genome_index ) );
    self.log( str( self.weakest_genome_index ) );

  def compute_population_makeup( self ):

    randoms           = 0;
    crossovers        = 0;
    mutants           = 0;
    crossover_mutants = 0;
    elites            = 0;

    for i in range( self.population_size ):

      if ( self.population[ i ].created_by == 0 ):

        randoms = randoms + 1;

      elif ( self.population[ i ].created_by == 1 ):

        crossovers = crossovers + 1;

      elif ( self.population[ i ].created_by == 2 ):
```

```
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(
            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
          # 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;
          number_of_crossovers            = 0;
1383
1385      mutation_operator_progress_sum  = 0.0;
          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
            fitness score
          # than its parent.
1391
1393      for i in range( self.population_size ):
1395        if ( self.population[ i ].created_by == 1 ): # Created by crossover.
1397          crossover_operator_progress_sum += ( self.population[ i ].parent_fitness - self.population[ i ].
            fitness );
1399          number_of_crossovers += 1;
1401        elif ( self.population[ i ].created_by == 2 ): # Created by mutation.
1403          mutation_operator_progress_sum  += ( self.population[ i ].parent_fitness - self.population[ i ].
            fitness );
1405          number_of_mutations += 1;
1407      # Now calculate the average crossover and mutation progress for the population.
1409      self.crossover_operator_progress_average = 0.0;
          self.mutation_operator_progress_average  = 0.0;
1411      if ( number_of_crossovers != 0 ):
1413        self.crossover_operator_progress_average = ( crossover_operator_progress_sum ) / float(
            number_of_crossovers );
1415      if ( number_of_mutations != 0 ):
1417        self.mutation_operator_progress_average  = ( mutation_operator_progress_sum ) / float(
            number_of_mutations );
1419      # Adjust crossover and mutation rate adjustments.
1421      self.log( "Adjusting crossover and mutation rate adjustments." );
1423      if ( self.lowest_fitness > self.highest_fitness ):
1425        self.log( "L > H" );
1427        self.crossover_probability_adjustment = 0.01 * ( ( self.lowest_fitness - self.average_fitness ) / (
            self.lowest_fitness - self.highest_fitness ) );
1429        self.mutation_probability_adjustment  = 0.01 * ( ( self.lowest_fitness - self.average_fitness ) / (
            self.lowest_fitness - self.highest_fitness ) );
1431        self.log( str( self.crossover_probability_adjustment ) + " " + str( self.
            mutation_probability_adjustment ) );
1433      elif ( self.lowest_fitness == self.average_fitness ):
1435        self.log( "L == A" );
1437        self.crossover_probability_adjustment = 0.01;
1439        self.mutation_probability_adjustment  = 0.01;
1441      self.log( "Crossover progress average." );
1443      self.log( str( self.crossover_operator_progress_average ) );
1445      self.log( "Mutation progress average." );
1447      self.log( str( self.mutation_operator_progress_average ) );
1449      # Adjust crossover and mutation rates.
```

```
1451      if ( self.crossover_operator_progress_average > self.mutation_operator_progress_average ):
1453        self.log( "Adjusting crossover/mutation probabilities. CPA > MPA." );
1455        self.crossover_probability = self.crossover_probability + self.crossover_probability_adjustment;
1457        self.mutation_probability  = self.mutation_probability  - self.mutation_probability_adjustment;
1459      elif ( self.crossover_operator_progress_average < self.mutation_operator_progress_average ):
1461        self.log( "Adjusting crossover/mutation probabilities. CPA < MPA." );
1463        self.crossover_probability = self.crossover_probability - self.crossover_probability_adjustment;
1465        self.mutation_probability  = self.mutation_probability  + self.mutation_probability_adjustment;
1467      elif ( self.crossover_operator_progress_average == self.mutation_operator_progress_average ):
1469        self.log( "Not adjusting crossover/mutation probabilities. CPA == MPA." );
1471        # Do not adjust.
1473        pass;
1475      self.crossover_probability = get_clamped_value( self.crossover_probability, self.
            crossover_probability_minimum, 1.0 - self.mutation_probability_minimum  );
1477      self.mutation_probability  = get_clamped_value( self.mutation_probability,  self.
            mutation_probability_minimum,  1.0 - self.crossover_probability_minimum );
1479    def sort_population( self, descending = None ):
1481      self.log( "Sorting population. Descending: " + str( descending ) );
1483      if ( descending == None or descending == False ):
1485        self.population = sorted( self.population, key = lambda genome: genome.fitness, reverse = False );
1487      elif ( descending == True ):
1489        self.population = sorted( self.population, key = lambda genome: genome.fitness, reverse = True );
1491      self.log( "Sorted." );
1493      self.log( str( self.population[ 0 ].fitness ) + " " + str( self.population[ -1 ].fitness ) );
1495    def generate_new_generation( self ):
1497      self.log( "Creating a new generation." );
1499      # Sort the population based on fitness in ascending order.
          # 0.0 is the highest fitness and infinity is the lowest fitness.
1501      # So sort in non-increasing order.
1503      self.log( "Sorting population." );
1505      self.sort_population( descending = True );
1507      # Update population metrics.
1509      self.log( "Updating population metrics." );
1511      self.update_population_metrics( );
1513      # Adapt crossover and mutation probabilities if using self-adaptation.
1515      if ( self.use_self_adaptation == True ):
1517        self.log( "Adapting crossover/mutation probabilities." );
1519        self.adapt_crossover_and_mutation_probabilities( );
1521      # Calculate current population makeup.
1523      self.log( "Computing population makeup." );
1525      self.compute_population_makeup( );
1527      # Create a temporary population to store newly created generation.
1529      self.log( "Creating empty new population." );
1531      new_population = [ ];
1533      # Allow the top N elite to pass into the next generation.
1535      self.log( "Performing elitism." );
1537      self.elitism_operator( new_population );
1539      # Perform crossover and mutation separately?
1541      if ( not self.perform_crossover_and_mutation_sequentially ):
1543        self.log( "Performing crossover and mutation separately." );
1545        self.log( "Entering the loop." );
1547        # Now we enter the GA loop.
1549        # 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
              # 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 = [ ];
            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;
                  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;
```

```
            self.total_number_of_mutation_attempts  += 1;

            new_population.append( offspring[ "one" ] );

          elif ( offspring[ "one" ].created_by == 3 ):

            self.log( "Adding crossed over and mutated offspring." );

            self.total_number_of_crossovers         += 1;
            self.total_number_of_mutations          += 1;
            self.total_number_of_crossover_attempts += 1;
            self.total_number_of_mutation_attempts  += 1;

            new_population.append( offspring[ "one" ] );

          # Old population size should match this new population size.

          if ( len( new_population ) == self.population_size ):

            break;

          self.log( "Adding second offspring." );

          # Second offspring.

          if ( offspring[ "two" ].created_by == 1 ):

            self.log( "Adding crossover offspring." );

            self.total_number_of_crossovers         += 1;
            self.total_number_of_crossover_attempts += 1;
            self.total_number_of_mutation_attempts  += 1;

            new_population.append( offspring[ "two" ] );

          elif ( offspring[ "two" ].created_by == 2 ):

            self.log( "Adding mutation offspring." );

            self.total_number_of_mutations          += 1;
            self.total_number_of_crossover_attempts += 1;
            self.total_number_of_mutation_attempts  += 1;

            new_population.append( offspring[ "two" ] );

          elif ( offspring[ "two" ].created_by == 3 ):

            self.log( "Adding crossed over and mutated offspring." );

            self.total_number_of_crossovers         += 1;
            self.total_number_of_mutations          += 1;
            self.total_number_of_crossover_attempts += 1;
            self.total_number_of_mutation_attempts  += 1;

            new_population.append( offspring[ "two" ] );

        assert len( new_population ) == self.population_size, "New population size does not equal population
         size setting.";

        # Finished so assign new pop to the current population.

        self.population = [ ];
        self.population = copy.deepcopy( new_population );
        new_population  = [ ];

      # Calculate the observed rates.

      self.observed_crossover_rate = self.total_number_of_crossovers / self.
        total_number_of_crossover_attempts;
      self.observed_mutation_rate  = self.total_number_of_mutations  / self.
        total_number_of_mutation_attempts;

      self.log( "Observed cross/mut rates." );

      self.log( str( self.observed_crossover_rate ) + " " + str( self.observed_mutation_rate ) );

      # Advance generation counter.

      self.generation_number += 1;
  '''

  The BBAutoTune object.

  '''

  class BBAutoTune( ):

    def __init__( self, ga, dbm ):

      # The genetic algorithm and the database manager.

      self.ga  = ga;
      self.dbm = dbm;

      # Run ID.

      self.run_id = None;

      # The current genome being evaluated.

      self.current_genome = 0;
```

```
    # Log debug statements.

    self.debug = False

    self.log_file_name = "";

    # Open a browser window to the GA monitor.

    self.open_ga_monitor_browser_window = False;

    # Load in real robot data.

    real_forward_x_primes = pickle.load( open( get_scripts_location( ) + "data/real_robot_motion/forward/
      x_p_values.pkl", "rb" ) );
    real_forward_y_primes = pickle.load( open( get_scripts_location( ) + "data/real_robot_motion/forward/
      y_p_values.pkl", "rb" ) );
    real_forward_t_primes = pickle.load( open( get_scripts_location( ) + "data/real_robot_motion/forward/
      t_p_values.pkl", "rb" ) );

    real_forward_motion = [ ];

    for i in range( len( real_forward_x_primes ) ):

      x = real_forward_x_primes[ i ];
      y = real_forward_y_primes[ i ];
      t = real_forward_t_primes[ i ];

      real_forward_motion.append(

        [ x, y, t ]

      );

    real_forward_motion = numpy.array( real_forward_motion );

    # Calculate the robust covariance matrix and the robust mean (location).

    self.mcd_fitted = sklearn.covariance.MinCovDet( assume_centered = False, support_fraction = 0.5 * (
      len( real_forward_motion ) + 3.0 + 1.0 ) ).fit( real_forward_motion );

    self.rcm = self.mcd_fitted.covariance_;

    self.rcm_inv = numpy.linalg.inv( self.rcm );

    self.rm  = self.mcd_fitted.location_;

    # The threshold for Chi-square with 3 degrees of freedom (x,y,t) and an alpha value of 0.005.

    self.genome_fitness_threshold = math.sqrt( scipy.stats.chi2.isf( 1.0 - 0.995, 3 ) );

  def start(

    self,
    population_size,
    max_generations,
    number_of_elite,
    crossover_probability,
    mutation_probability,
    max_torque,
    use_rank_fitness_selection,
    perform_crossover_and_mutation_sequentially,
    use_self_adaptation,
    open_ga_monitor_browser_window,
    debug

  ):

    self.run_id = int( round( time.time( ) * 1000 ) );

    self.debug = debug or False;

    if ( self.debug == True ):

      logs_location = get_scripts_location( ) + "logs/";

      self.log_file_name = logs_location + "log_" + str( self.run_id ) + ".dat";

      self.ga.set_log_file_name( self.log_file_name );

      bpy.data.objects[ "robot_monitor" ].game.properties[ "log_file_name" ].value = self.log_file_name;

    self.log( "Run ID." );

    self.log( str( self.run_id ) );

    # Pass the file name and directory where the robot
    # monitor will record the robot's initial and final state.

    self.log( "Setting the robot monitor's shared data file name." );

    bpy.data.objects[ "robot_monitor" ].game.properties[ "shared_data_file_name" ].value =
      get_scripts_location( ) + "shared_data/genome_I_F.dat";

    self.log( "Connecting to the database." );

    self.dbm.connect_to_database( );

    self.log( "Starting the GA manager." );

    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 ) );
          self.log( str( max_generations ) );
1853      self.log( str( number_of_elite ) );
          self.log( str( crossover_probability ) );
1855      self.log( str( mutation_probability ) );
          self.log( str( use_rank_fitness_selection ) );
1857      self.log( str( perform_crossover_and_mutation_sequentially ) );
          self.log( str( use_self_adaptation ) );
1859
          self.log( "Max torque." );
1861
          self.max_torque = max_torque;
1863
          self.log( str( self.max_torque ) );
1865
          self.ga.set_population_size( population_size );
1867      self.ga.set_max_generations( max_generations );
          self.ga.set_number_of_elite( number_of_elite );
1869      self.ga.set_crossover_probability( crossover_probability );
          self.ga.set_mutation_probability( mutation_probability );
1871      self.ga.set_use_rank_fitness_selection( use_rank_fitness_selection );
          self.ga.set_perform_crossover_and_mutation_sequentially( perform_crossover_and_mutation_sequentially )
            ;
1873      self.ga.set_use_self_adaptation( use_self_adaptation );

1875      self.log( "Creating a random GA population." );

1877      self.ga.create_randomized_population( );

1879      self.log( "Setting the current genome to 0." );

1881      self.current_genome = 0;

1883      self.log( "Entering loop." );

1885      while ( self.ga.get_generation_number( ) < self.ga.get_max_generations( ) ):

1887        self.log( "Current genome." );

1889        self.log( str( self.ga.get_genome( self.current_genome ) ) );

1891        # Populate physics engine parameters.

1893        self.log( "Populating the physics engine parameters." );

1895        self.populate_physics_engine_parameters( self.ga.get_genome( self.current_genome ).get_genes( ) );

1897        # Run game engine.

1899        self.log( "Starting the game engine." );

1901        self.start_game_engine( );

1903        # Calculate current genome fitness.

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,
            e_time)
            # which was recorded by the robot monitor while the game engine was running.
1907
            self.log( "Game engine stopped." );
1909
            self.log( "Getting genome (I)nitial and (F)inal state." );
1911        self.log( "x_pos , y_pos , z_pos , x_ori , y_ori , z_ori, s/e_tim" );

1913        shared_data_file = open( get_scripts_location( ) + "shared_data/genome_I_F.dat", "r" );

1915        I = shared_data_file.readline( ).rstrip( );
            I = I.split( "," );
1917        I[ 0 ] = float( I[ 0 ] ); # x position.
            I[ 1 ] = float( I[ 1 ] ); # y position.
1919        I[ 2 ] = float( I[ 2 ] ); # z position.
            I[ 3 ] = float( I[ 3 ] ); # x orientation.
1921        I[ 4 ] = float( I[ 4 ] ); # y orientation.
            I[ 5 ] = float( I[ 5 ] ); # z orientation.
1923        I[ 6 ] = float( I[ 6 ] ); # Start time.

1925        F = shared_data_file.readline( ).rstrip( );
            F = F.split( "," );
1927        F[ 0 ] = float( F[ 0 ] ); # x' position.
            F[ 1 ] = float( F[ 1 ] ); # y' position.
1929        F[ 2 ] = float( F[ 2 ] ); # z' position.
            F[ 3 ] = float( F[ 3 ] ); # x' orientation.
1931        F[ 4 ] = float( F[ 4 ] ); # y' orientation.
            F[ 5 ] = float( F[ 5 ] ); # z' orientation.
1933        F[ 6 ] = float( F[ 6 ] ); # End time.

1935        shared_data_file.close( );

1937        os.remove( get_scripts_location( ) + "shared_data/genome_I_F.dat" );

1939        # Record simulated robot motion and its resulting fitness.

1941        simulated_robot_motion_file = open( get_scripts_location( ) + "data/
            simulated_robot_motion_with_fitness/forward/" + "srmfwf_" + str( self.run_id ) + ".dat", "a" );

1943        write_string  = str( I[ 0 ] ) + ",";
            write_string += str( I[ 1 ] ) + ",";
1945        write_string += str( I[ 2 ] ) + ",";
```

```
        write_string += str( I[ 3 ] ) + ",";
1947    write_string += str( I[ 4 ] ) + ",";
        write_string += str( I[ 5 ] ) + ";";
1949
        write_string += str( F[ 0 ] ) + ",";
1951    write_string += str( F[ 1 ] ) + ",";
        write_string += str( F[ 2 ] ) + ",";
1953    write_string += str( F[ 3 ] ) + ",";
        write_string += str( F[ 4 ] ) + ",";
1955    write_string += str( F[ 5 ] ) + ";";

1957    simulated_robot_motion_file.write( write_string );

1959    simulated_robot_motion_file.close( );

1961    self.log( str( I ) );

1963    self.log( str( F ) );

1965    self.log( "Calculating genome fitness." );

1967    current_genome_fitness = self.calculate_genome_fitness( I, F );

1969    self.log( "Genome fitness." );

1971    self.log( str( current_genome_fitness ) );

1973    self.ga.set_genome_fitness( self.current_genome, current_genome_fitness );

1975    # Add the fitness to the last line of the simulated robot motion recorded.

1977    simulated_robot_motion_file = open( get_scripts_location( ) + "data/
        simulated_robot_motion_with_fitness/forward/" + "srmfwf_" + str( self.run_id ) + ".dat", "a" );

1979    simulated_robot_motion_file.write( str( current_genome_fitness ) + "\n" );

1981    simulated_robot_motion_file.close( );

1983    # Record the genome's phenotype (the physics parameters) and its corresponding fitness.

1985    physics_parameters_with_fitness_file = open( get_scripts_location( ) + "data/
        physics_parameters_with_fitness/" + "ppwf_" + str( self.run_id ) + ".dat", "a" );

1987    physics_parameters_with_fitness_file.write( "fitness," + str( current_genome_fitness ) + "\n\n" );

1989    physics_parameters_with_fitness_file.close( );

1991    # Increase current genome + 1.

1993    self.log( "Increasing current genome count." );

1995    self.current_genome += 1;

1997    self.log( str( self.current_genome ) );

1999    # If current genome is equal to the population size.

2001    if ( self.current_genome == self.ga.get_population_size( ) ):

2003      self.log( "Evaluated all genomes in population." );

2005      # Separate recorded simulated robot motion by generation.

2007      simulated_robot_motion_file = open( get_scripts_location( ) + "data/
          simulated_robot_motion_with_fitness/forward/" + "srmfwf_" + str( self.run_id ) + ".dat", "a" );

2009      simulated_robot_motion_file.write( "\n" );

2011      simulated_robot_motion_file.close( );

2013      self.log( "Current generation number: " + str( self.ga.get_generation_number( ) ) );

2015      # Update the population metrics.

2017      self.log( "Updating population metrics." );

2019      self.ga.update_population_metrics( );

2021      # Store the population metrics in the database.

2023      self.log( "Storing population metrics in the database." );

2025      a = str( self.ga.get_generation_number( ) );
          b = str( self.ga.get_highest_fitness( ) );
2027      c = str( self.ga.get_average_fitness( ) );
          d = str( self.ga.get_lowest_fitness( ) );
2029      e = str( self.ga.get_crossover_probability( ) );
          f = str( self.ga.get_mutation_probability( ) );
2031
          mysql_string  = "INSERT INTO `population_metrics` ( `generation_number`, `highest_fitness`, `
          average_fitness`, `lowest_fitness`, `crossover_probability`, `mutation_probability` ) ";
2033      mysql_string += "VALUES( " + a + ", " + b + ", " + c + ", " + d + ", " + e + ", " + f + " );";

2035      self.dbm.execute( mysql_string );

2037      self.log( mysql_string );

2039      # Generate a new population.

2041      self.log( "Generating a new generation." );

2043      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.
         # If this is the case, set the fitness to some large value.
2113
         # initial/final structure:
2115     #   x_pos   0
         #   y_pos   1
2117     #   z_pos   2
         #   x_rot   3
2119     #   y_rot   4
         #   z_rot   5
2121     # s/e_tim   6
2123      fitness = 9999999999.0;
2125      for i in range( len( final ) ):
2127        if ( numpy.isnan( final[ i ] ) ):
2129          return fitness;
2131      # Only 3 dof was recorded for the real robot.
         # So assemble for the simulated robot its x' position, y' position, and z' orientation.
2133
         final_trimmed = [ final[ 0 ], final[ 1 ], final[ 5 ] ];
2135
         md, md2 = self.calculate_mahalanobis_distance( final_trimmed );
2137
         self.log( "Genome mahalanobis distance." );
2139
         self.log( str( md ) );
2141
         # Penalties?
2143
         self.log( "Penalties." );
2145
```

```
        # Time > 1 seconds.
2147
        elapsed_time = abs( ( ( final[ 6 ] - initial[ 6 ] ) / 1000.0 ) - 1.0 );
2149
        self.log( "Elapsed time > 1 seconds: " + str( elapsed_time ) );
2151
        # Rotation in x.
2153
        rotation_x = abs( final[ 3 ] - initial[ 3 ] );
2155
        self.log( "X rotation: " + str( rotation_x ) );
2157
        # Rotation in y.
2159
        rotation_y = abs( final[ 4 ] - initial[ 4 ] );
2161
        self.log( "Y rotation: " + str( rotation_y ) );
2163
        # Translation in z.
2165
        translation_z = abs( final[ 2 ] - initial[ 2 ] );
2167
        self.log( "Z translation: " + str( translation_z ) );
2169
        fitness = md + elapsed_time + rotation_x + rotation_y + translation_z;
2171
        return fitness;
2173
    def populate_physics_engine_parameters( self, genome_genes ):
2175
        self.log( "Genome genes." );
2177
        self.log( str( genome_genes ) );
2179
        assert len( genome_genes ) == self.ga.get_number_of_genes_per_genome( ), "Cannot populate physics
          engine parameters.";
2181
        # Blender API call examples:
2183    #
        # bpy.data.objects["Cylinder"].game.actuators["Motion"].torque = [0,401,0];
2185    # bpy.data.objects["Cylinder"].game.sensors["Always"].use_tap = False;
        # bpy.ops.logic.actuator_add( type="MOTION", name="motion1", object="Cylinder");
2187    # bpy.data.objects["Cylinder"].game.controllers[ "Python" ].link( sensor=None, actuator=bpy.data.
          objects["Cylinder"].game.actuators["motion1"] );
        # bpy.data.objects["Cylinder"].game.mass = 10000.0;
2189    # bpy.data.scenes["Scene"].game_settings.physics_gravity;

2191    # Blender's largest number called "inf".
        # To find, set an appropriate field to a very large
2193    # number and then click on the field and copy the number.
        # When clicking the field, the "inf" will turn numeric.
2195
        INF = 340282346638528859811704183484516925440.0;
2197
        BOOLEANS = [ False, True ];
2199
        # Assumes the correct scene, wheel object, sensor, controller, and actuator names.
2201    # These names were set by hand in the .blend file. If they are changed, Blender
        # will throw an exception.
2203
        scene          = "bbautotune";
2205    front_wheel_l  = "robot_1_wheel_front_L";
        front_wheel_r  = "robot_1_wheel_front_R";
2207    back_wheel_l   = "robot_1_wheel_back_L";
        back_wheel_r   = "robot_1_wheel_back_R";
2209    wheel_material = "wheel";
        actuator       = "torque_z";
2211
        ### WORLD
2213
        # Gravity.
2215
        #gravity = get_clamped_value( ( genome_genes[ 0 ] * 10000.0 ), 0.0, 10000.0 );
2217
        gravity = get_clamped_value( ( genome_genes[ 0 ] * 15.0 ), 0.0, 100.0 );
2219
        self.log( "Setting gravity." );
2221
        self.log( str( gravity ) );
2223
        bpy.data.scenes[ scene ].game_settings.physics_gravity = gravity;
2225
        # Sub-steps.
2227
        # You can input 1 up to 50. However, if only sliding the values, the value only goes from 1 to 5.
2229    # Not sure if this is a bug in the Blender code.

2231    sub_steps = get_clamped_value( math.floor( ( genome_genes[ 1 ] * ( 5 - 1 ) ) + 1 ), 1, 5 );

2233    self.log( "Setting sub steps." );

2235    self.log( str( sub_steps ) );

2237    bpy.data.scenes[ scene ].game_settings.physics_step_sub = sub_steps;

2239    # FPS.

2241    # Setting FPS within [1,30) makes the game engine run extremely slow so allow the GA to find a
          solution
        # within the range of 30 to 10000 FPS.
2243
        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 ];
        bpy.data.objects[ back_wheel_l  ].scale = [ scale, scale, scale ];
2263    bpy.data.objects[ back_wheel_r  ].scale = [ scale, scale, scale ];
2265    '''
2267    ### MATERIAL
2269    # Use physics?
2271    # Round returns incorrect values but converting its return value to a string does return the right
          value.
        # Convert only the numbers before the decimal '.' to an integer.
2273
        '''
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 =
          use_material_physics;
2285    bpy.data.objects[ front_wheel_r ].material_slots[ 0 ].material.game_settings.physics =
          use_material_physics;
        bpy.data.objects[ back_wheel_l  ].material_slots[ 0 ].material.game_settings.physics =
          use_material_physics;
2287    bpy.data.objects[ back_wheel_r  ].material_slots[ 0 ].material.game_settings.physics =
          use_material_physics;
2289    '''
2291    # 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;
        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;
        bpy.data.objects[ back_wheel_r  ].material_slots[ 0 ].material.physics.friction = material_friction;
2303
        # Elasticity.
2305
        material_elasticity = get_clamped_value( ( genome_genes[ 6 ] ), 0.0, 1.0 );
2307
        self.log( "Setting material elasticity." );
2309
        self.log( str( material_elasticity ) );
2311
        bpy.data.objects[ front_wheel_l ].material_slots[ 0 ].material.physics.elasticity =
          material_elasticity;
2313    bpy.data.objects[ front_wheel_r ].material_slots[ 0 ].material.physics.elasticity =
          material_elasticity;
        bpy.data.objects[ back_wheel_l  ].material_slots[ 0 ].material.physics.elasticity =
          material_elasticity;
2315    bpy.data.objects[ back_wheel_r  ].material_slots[ 0 ].material.physics.elasticity =
          material_elasticity;
2317    ### PHYSICS
2319    # Type?
2321    '''
2323    PHYSICS_TYPES = [ "NO_COLLISION", "STATIC", "DYNAMIC", "RIGID_BODY", "SOFT_BODY", "OCCLUDE", "SENSOR",
          "NAVMESH", "CHARACTER" ];
2325    physics_type = get_clamped_value( math.floor( genome_genes[ 7 ] * len( PHYSICS_TYPES ) ), 0, len(
          PHYSICS_TYPES ) - 1 );
2327    physics_type = PHYSICS_TYPES[ physics_type ];
2329    bpy.data.objects[ front_wheel_l ].game.physics_type = physics_type;
        bpy.data.objects[ front_wheel_r ].game.physics_type = physics_type;
2331    bpy.data.objects[ back_wheel_l  ].game.physics_type = physics_type;
        bpy.data.objects[ back_wheel_r  ].game.physics_type = physics_type;
2333
        '''
2335
```

```
        # Ghost?
2337
        '''
2339
        index = int( str( round( genome_genes[ 8 ] ) ).split( "." )[ 0 ] );
2341
        use_ghost = BOOLEANS[ index ];
2343
        bpy.data.objects[ front_wheel_l ].game.use_ghost = use_ghost;
2345    bpy.data.objects[ front_wheel_r ].game.use_ghost = use_ghost;
        bpy.data.objects[ back_wheel_l  ].game.use_ghost = use_ghost;
2347    bpy.data.objects[ back_wheel_r  ].game.use_ghost = use_ghost;
2349    '''
2351    # Mass.
2353    #mass = get_clamped_value( ( genome_genes[ 9 ] * ( 10000.0 - 0.01 ) ) + 0.010, 0.010, 10000.0 );
2355    mass = get_clamped_value( ( genome_genes[ 9 ] * ( 15.0 - 0.010 ) ) + 0.010, 0.010, 15.0 );
2357    self.log( "Setting mass." );
2359    self.log( str( mass ) );
2361    bpy.data.objects[ front_wheel_l ].game.mass = mass;
        bpy.data.objects[ front_wheel_r ].game.mass = mass;
2363    bpy.data.objects[ back_wheel_l  ].game.mass = mass;
        bpy.data.objects[ back_wheel_r  ].game.mass = mass;
2365
        # Form factor?
2367
        '''
2369
        form_factor = get_clamped_value( ( genome_genes[ 10 ] ), 0.0, 1.0 );
2371
        self.log( "Setting form factor." );
2373
        self.log( str( form_factor ) );
2375
        bpy.data.objects[ front_wheel_l ].game.form_factor = form_factor;
2377    bpy.data.objects[ front_wheel_r ].game.form_factor = form_factor;
        bpy.data.objects[ back_wheel_l  ].game.form_factor = form_factor;
2379    bpy.data.objects[ back_wheel_r  ].game.form_factor = form_factor;
2381    '''
2383    # Velocity maximum.
2385    velocity_max = get_clamped_value( ( genome_genes[ 11 ] * 1000.0 ), 0.0, 1000.0 );
2387    self.log( "Setting velocity max." );
2389    self.log( str( velocity_max ) );
2391    bpy.data.objects[ front_wheel_l ].game.velocity_max = velocity_max;
        bpy.data.objects[ front_wheel_r ].game.velocity_max = velocity_max;
2393    bpy.data.objects[ back_wheel_l  ].game.velocity_max = velocity_max;
        bpy.data.objects[ back_wheel_r  ].game.velocity_max = velocity_max;
2395
        # Damping translation.
2397
        damping = get_clamped_value( ( genome_genes[ 12 ] ), 0.0, 1.0 );
2399
        self.log( "Setting translation damping." );
2401
        self.log( str( damping ) );
2403
        bpy.data.objects[ front_wheel_l ].game.damping = damping;
2405    bpy.data.objects[ front_wheel_r ].game.damping = damping;
        bpy.data.objects[ back_wheel_l  ].game.damping = damping;
2407    bpy.data.objects[ back_wheel_r  ].game.damping = damping;
2409    # Damping rotation.
2411    rotation_damping = get_clamped_value( ( genome_genes[ 13 ] ), 0.0, 1.0 );
2413    self.log( "Setting rotation damping." );
2415    self.log( str( rotation_damping ) );
2417    bpy.data.objects[ front_wheel_l ].game.rotation_damping = rotation_damping;
        bpy.data.objects[ front_wheel_r ].game.rotation_damping = rotation_damping;
2419    bpy.data.objects[ back_wheel_l  ].game.rotation_damping = rotation_damping;
        bpy.data.objects[ back_wheel_r  ].game.rotation_damping = rotation_damping;
2421
        # Use collision bounds?
2423
        '''
2425
        index = int( str( round( genome_genes[ 14 ] ) ).split( "." )[ 0 ] );
2427
        use_collision_bounds = BOOLEANS[ index ];
2429
        self.log( "Setting use collision bounds." );
2431
        self.log( str( use_collision_bounds ) );
2433
        bpy.data.objects[ front_wheel_l ].game.use_collision_bounds = use_collision_bounds;
2435    bpy.data.objects[ front_wheel_r ].game.use_collision_bounds = use_collision_bounds;
        bpy.data.objects[ back_wheel_l  ].game.use_collision_bounds = use_collision_bounds;
2437    bpy.data.objects[ back_wheel_r  ].game.use_collision_bounds = use_collision_bounds;
```

```
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;
          bpy.data.objects[ front_wheel_r ].game.collision_margin = collision_margin;
2453      bpy.data.objects[ back_wheel_l  ].game.collision_margin = collision_margin;
          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(
            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;
2473      bpy.data.objects[ front_wheel_r ].game.collision_bounds_type = collision_bounds_type;
          bpy.data.objects[ back_wheel_l  ].game.collision_bounds_type = collision_bounds_type;
2475      bpy.data.objects[ back_wheel_r  ].game.collision_bounds_type = collision_bounds_type;
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
          # 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
          # 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
            space,
          # the z-axis points out of the wheel hub.
2495      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;
          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 ];
          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 ];
          bpy.data.objects[ back_wheel_r  ].game.actuators[ actuator ].torque = [ 0, 0, torque_z ];
2505
          # Record the genome's phenotype (the physics parameters) and its eventual fitness.
2507
          physics_parameters_with_fitness_file = open( get_scripts_location( ) + "data/
            physics_parameters_with_fitness/" + "ppwf_" + str( self.run_id ) + ".dat", "a" );
2509      physics_parameters_with_fitness_file.write( "gravity," + str( gravity ) + "\n" );
2511      physics_parameters_with_fitness_file.write( "sub_steps," + str( sub_steps ) + "\n" );
          physics_parameters_with_fitness_file.write( "fps," + str( fps ) + "\n" );
2513      #physics_parameters_with_fitness_file.write( "scale," + str( scale ) + "\n" );
          #physics_parameters_with_fitness_file.write( "use_material_physics," + str( use_material_physics ) +
            "\n" );
2515      physics_parameters_with_fitness_file.write( "material_friction," + str( material_friction ) + "\n" );
          physics_parameters_with_fitness_file.write( "material_elasticity," + str( material_elasticity ) + "\n"
            );
2517      #physics_parameters_with_fitness_file.write( "physics_type," + str( physics_type ) + "\n" );
          #physics_parameters_with_fitness_file.write( "use_ghost," + str( use_ghost ) + "\n" );
2519      physics_parameters_with_fitness_file.write( "mass," + str( mass ) + "\n" );
          #physics_parameters_with_fitness_file.write( "form_factor," + str( form_factor ) + "\n" );
2521      physics_parameters_with_fitness_file.write( "velocity_max," + str( velocity_max ) + "\n" );
          physics_parameters_with_fitness_file.write( "damping," + str( damping ) + "\n" );
2523      physics_parameters_with_fitness_file.write( "rotation_damping," + str( rotation_damping ) + "\n" );
          #physics_parameters_with_fitness_file.write( "use_collision_bounds," + str( use_collision_bounds ) +
            "\n" );
2525      #physics_parameters_with_fitness_file.write( "collision_margin," + str( collision_margin ) + "\n" );
          physics_parameters_with_fitness_file.write( "collision_bounds_type," + str( collision_bounds_type ) +
            "\n" );
2527      physics_parameters_with_fitness_file.write( "torque_z," + str( torque_z ) + "\n" );
2529      physics_parameters_with_fitness_file.close( );
2531  class Database_Manager( ):
```

```
2533    def __init__( self, database_name ):
2535      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,
          "password":           self.password,
2555        "database":           self.database_name,
          '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:
2583
          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__ );
```

B.3 BlenderSim

Listing B.5: Robot_1_Controller.py

```
'''
David Lettier (C) 2014.

http://www.lettier.com/

BlenderSim Version 2.0

This file controls the robot's movements.

'''

# Imports.

import mathutils, math, time, copy, datetime, os, pickle, random;

def robot_1_moving( ):

  robot_1_base = bge.logic.getCurrentScene( ).objects[ "robot_1_base" ];

  test = {

      "xpos": robot_1_base.worldPosition.x * 100.0,
      "ypos": robot_1_base.worldPosition.y * 100.0,
      "zpos": robot_1_base.worldPosition.z * 100.0,
      "xrot": robot_1_base.worldOrientation.to_euler( ).x,
      "yrot": robot_1_base.worldOrientation.to_euler( ).y,
      "zrot": robot_1_base.worldOrientation.to_euler( ).z

  };

  stopped = len( set( test.items( ) ) ^ set( bge.logic.globalDict[ "robot_1_state" ].items( ) ) );

  if ( stopped == 0 ):

    return False;

  else:

    x1 = bge.logic.globalDict[ "robot_1_simulated_poses" ][ -2 ][ 1 ];
    y1 = bge.logic.globalDict[ "robot_1_simulated_poses" ][ -2 ][ 2 ];
    z1 = bge.logic.globalDict[ "robot_1_simulated_poses" ][ -2 ][ 3 ];

    x2 = test[ "xpos" ];
    y2 = test[ "ypos" ];
    z2 = test[ "zrot" ];

    dx = abs( x1 - x2 );
    dy = abs( y1 - y2 );
    dz = abs( z1 - z2 );

    if ( dx <= 0.02 and dy <= 0.02 and dz <= 0.00087 ):

      return False;
    else:

      return True;

def update_robot_1_state( ):

  robot_1_base = bge.logic.getCurrentScene( ).objects[ "robot_1_base" ];

  bge.logic.globalDict[ "robot_1_state" ] = {

      "xpos": robot_1_base.worldPosition.x * 100.0,
      "ypos": robot_1_base.worldPosition.y * 100.0,
      "zpos": robot_1_base.worldPosition.z * 100.0,
      "xrot": robot_1_base.worldOrientation.to_euler( ).x,
      "yrot": robot_1_base.worldOrientation.to_euler( ).y,
      "zrot": robot_1_base.worldOrientation.to_euler( ).z

  };

def stop_robot_1( ):

  bge.logic.robot_1_wheel_front_L.applyTorque( [ 0.0, 0.0, 0.0 ], True );
  bge.logic.robot_1_wheel_front_R.applyTorque( [ 0.0, 0.0, 0.0 ], True );
  bge.logic.robot_1_wheel_back_L.applyTorque(  [ 0.0, 0.0, 0.0 ], True );
  bge.logic.robot_1_wheel_back_R.applyTorque(  [ 0.0, 0.0, 0.0 ], True );

def drop_robot_1_campose_marker( ):

  robot_1_campose_marker = bge.logic.getCurrentScene( ).addObject( "robot_1_campose_marker", obj );
  robot_1_campose_marker.worldPosition = obj.worldPosition;
  robot_1_campose_marker.worldPosition.z = 23.0 / 100.0;

def drop_robot_1_waypoint_marker( waypoint ):

  robot_1_waypoint_marker = bge.logic.getCurrentScene( ).addObject( "waypoint_marker", obj );
  robot_1_waypoint_marker.worldPosition.x = waypoint[ 1 ] / 100.0;
  robot_1_waypoint_marker.worldPosition.y = waypoint[ 0 ] / 100.0;
  robot_1_waypoint_marker.worldPosition.z = 25.0 / 100.0;

def add_sim_pose( ):

  bge.logic.globalDict[ "robot_1_simulated_poses" ].append(

    [
      time.time( ) * 1000.0,
      obj.worldPosition[ 0 ] * 100.0,
      obj.worldPosition[ 1 ] * 100.0,
      obj.worldOrientation.to_euler( ).z
```

```
    ]

  );

# Globals.

local = True;

world = False;

# Get the controller.

controller = bge.logic.getCurrentController( );

# Get the game object that the controller is attached to.

obj = controller.owner;

# Initialize variables and flags.

if ( obj[ "init" ] == True ):

  obj[ "init" ] = False;

  # Robot 1's wheels.

  bge.logic.robot_1_wheel_front_L = bge.logic.getCurrentScene( ).objects[ "robot_1_wheel_front_L" ];
  bge.logic.robot_1_wheel_front_R = bge.logic.getCurrentScene( ).objects[ "robot_1_wheel_front_R" ];
  bge.logic.robot_1_wheel_back_L  = bge.logic.getCurrentScene( ).objects[ "robot_1_wheel_back_L"  ];
  bge.logic.robot_1_wheel_back_R  = bge.logic.getCurrentScene( ).objects[ "robot_1_wheel_back_R"  ];

  bge.logic.robot_1_last_move_time = time.time( ) * 1000.0;

  update_robot_1_state( );

  bge.logic.globalDict[ "robot_1_waypoints" ] = [

    "stop"

  ];

  bge.logic.globalDict[ "robot_1_simulated_poses" ] = [ ];

  add_sim_pose( );
  add_sim_pose( );

if ( ( bge.logic.globalDict[ "robot_1_waypoints" ][ 0 ] != "stop" ) and ( not robot_1_moving( ) ) and ( (
      time.time( ) * 1000.0 ) - bge.logic.robot_1_last_move_time >= 0.0 ) ):

  # Get and drop the next waypoint in the arena.

  waypoint = bge.logic.globalDict[ "robot_1_waypoints" ][ 0 ];

  drop_robot_1_waypoint_marker( waypoint );

  # Calculate the angle to turn in order to face the waypoint.

  # Convert the waypoint to a 3D vector.

  waypoint = mathutils.Vector( ( waypoint[ 1 ], waypoint[ 0 ], 1.0 ) );

  # Translate the waypoint to the robot's local space.

  mat_trans = mathutils.Matrix.Translation( ( -obj.worldPosition[ 0 ] * 100.0, -obj.worldPosition[ 1 ] *
      100.0, -obj.worldPosition[ 2 ] * 100.0 ) );

  waypoint_trans = mat_trans * waypoint;

  waypoint_rot = mathutils.Matrix.Rotation( -obj.worldOrientation.to_euler( )[ 2 ], 4, "Z" ) *
      waypoint_trans;

  # Now that the waypoint world coordinate is transformed to the robot's local space,
  # compute the angle between the robot's x-axis and the waypoint line going from the
  # robot's origin to the waypoint.

  rotateZ = math.atan2( waypoint_rot[ 1 ], waypoint_rot[ 0 ] );

  # First turn and then move forward.

  rotateZ = rotateZ * 180.0 / math.pi;

  if ( abs( rotateZ ) < 1.0 ): # Turn error threshold.

    a = 82.7271515601;
    b = 23.12349975;
    c = waypoint_rot[ 0 ];
    f = ( a / b ) * c;

    bge.logic.robot_1_wheel_front_L.applyTorque( [ 0.0, 0.0, f ], True );
    bge.logic.robot_1_wheel_front_R.applyTorque( [ 0.0, 0.0, f ], True );
    bge.logic.robot_1_wheel_back_L.applyTorque(  [ 0.0, 0.0, f ], True );
    bge.logic.robot_1_wheel_back_R.applyTorque(  [ 0.0, 0.0, f ], True );

    bge.logic.globalDict[ "robot_1_waypoints" ] = bge.logic.globalDict[ "robot_1_waypoints" ][ 1 :  ];

  elif ( rotateZ < 0.0 ):

    a  = 20.8;
    b  = 44.260811;
    c  = abs( rotateZ );
    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 );
      bge.logic.robot_1_wheel_back_L.applyTorque(  [ 0.0, 0.0,  t ], True );
206   bge.logic.robot_1_wheel_back_R.applyTorque(  [ 0.0, 0.0, -t ], True );

208  elif ( rotateZ > 0.0 ):

210    a  = 20.8;
      b  = 44.260811;
212   c  = abs( rotateZ );
      t  = ( a / b ) * c;
214
      bge.logic.robot_1_wheel_front_L.applyTorque( [ 0.0, 0.0, -t ], True );
216   bge.logic.robot_1_wheel_front_R.applyTorque( [ 0.0, 0.0,  t ], True );
      bge.logic.robot_1_wheel_back_L.applyTorque(  [ 0.0, 0.0, -t ], True );
218   bge.logic.robot_1_wheel_back_R.applyTorque(  [ 0.0, 0.0,  t ], True );

220  bge.logic.robot_1_last_move_time = time.time( ) * 1000.0;

222 update_robot_1_state( );

224 stop_robot_1( );

226 add_sim_pose( );

228 drop_robot_1_campose_marker( );

230 if ( not robot_1_moving( ) ):

232  pickle_file = open( "./pickled_data/robot_1_simulated_motion.pkl", "wb" );

234  pickle.dump( bge.logic.globalDict[ "robot_1_simulated_poses" ], pickle_file, protocol = 2, fix_imports =
          True );

236  pickle_file.close( );
```

Bibliography

[1] 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.

[2] 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 (Demonstration). 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.

[3] Elizabeth I. Sklar, A. Tuna Ozgelen, J. Pablo Munoz, Joel Gonzalez, Mark Manashirov, Susan L. Epstein, and Simon Parsons. Designing the HRTeam Framework: Lessons Learned from a Rough-and-Ready Human/Multi-Robot Team. In *Proceedings of the Workshop on Autonomous Robots and Multirobot Systems (ARMS) at Autonomous Agents and MultiAgent Systems (AAMAS)*, Taipei, Taiwan, May 2011.

[web] Bullet collision detection & physics library. `http://www.continuousphysics.com/Bullet/BulletFull/index.html`, . Accessed March 26, 2014.

[4] 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`.

[5] David Beasley, David R. Bull, and Ralph R. Martin. An overview of genetic algorithms: Part 1, fundamentals, 1993.

[6] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.

[7] Zbigniew Michalewicz, Robert Hinterding, and Maciej Michalewicz. Evolutionary algorithms. In Witold Pedrycz, editor, *Fuzzy Evolutionary Computation*, pages 3–31. Springer US, 1997. ISBN 978-1-4613-7811-2. doi: 10.1007/978-1-4615-6135-4_1. URL http://dx.doi.org/10.1007/978-1-4615-6135-4_1.

[8] 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.

[9] John R. Koza and James P. Rice. Automatic programming of robots using genetic programming. In William R. Swartout, editor, *AAAI*, pages 194–201. AAAI Press / The MIT Press, 1992. ISBN 0-262-51063-4.

[10] 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.

[11] F. Herrera, M. Lozano, and J.L. Verdegay. Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artificial Intelligence Review*, 12(4):265–319, 1998. ISSN 0269-2821. doi: 10.1023/A:1006504901164. URL http://dx.doi.org/10.1023/A%3A1006504901164.

[12] Daniel W. Dyer. Practical evolutionary computation: Elitism. http://blog.uncommons.org/2009/02/12/practical-evolutionary-computation-elitism/, February 2009. Accessed Apr 6, 2014.

[13] 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.

[14] 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.

[15] 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.

[16] 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`.

[17] 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] Blender 2.70.0 19e627c - API documentation. `http://www.blender.org/documentation/blender_python_api_2_70_release/contents.html`, . Accessed April 12, 2014.

[18] 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.

[19] R. De Maesschalck, D. Jouan-Rimbaud, and D.L. Massart. The mahalanobis distance. *Chemometrics and Intelligent Laboratory System*, 50(1):1–18, January 2000.

[20] 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.

[21] 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`.

[22] Peter J. Rousseeuw and Katrien Van Driessen. 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`.

[23] 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.

[24] Pankaj K. Agarwal, Rinat Ben Avraham, Haim Kaplan, and Micha Sharir. Computing the discrete fréchet distance in subquadratic time. *CoRR*, abs/1204.5333, 2012.

[25] 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.

[26] 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.

[27] Stage - the player project. `http://playerstage.sourceforge.net/wiki/Stage`, 2011. Accessed August 14, 2013.

[28] A. Tuna Özgelen, Eric Schneider, Elizabeth I. Sklar, Michael Costantino, Susan L. Epstein, and Simon Parsons. A first step toward testing multiagent coordination mechanisms on multi-robot teams. In *Proceedings of the Workshop on Autonomous Robots and Multirobot Systems (ARMS) at Autonomous Agents and MultiAgent Systems (AAMAS)*, St Paul, MN, USA, May 2013.

[29] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013. ISBN 0124077269, 9780124077263.

[30] Floating point arithmetic: Issues and limitations. `http://docs.python.org/3.4/tutorial/floatingpoint.html`, March 2014. Accessed Apr 11, 2014.