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

March 29, 2014

Acknowledgments

Abstract

TODO: Write something about how the GA *was* successful and how BlenderSim worked out.

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

[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 [2].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

TODO: Write this second to last.

1.1 Blender and Bullet

Stub.

1.2 Reality Gap

Stub.

1.3 SimPL

Stub.

1.4 BBAutoTune

Stub.

1.5 BlenderSim

Stub.

Chapter 2

Genetic Algorithms

2.1 Overview

Stub.

2.2 Genomes

Stub.

2.2.1 Genes

Stub.

2.2.1.1 Encoding

Stub.

2.3 Evaluation

Stub.

2.3.1 Criteria

Stub.

2.3.2 Fitness Function

Stub.

2.3.3 Fitness Landscape

Stub.

2.4 Operators

Stub.

2.4.1 Selection

Stub.

2.4.2 Elitism

Stub.

2.4.3 Crossover

Stub.

2.4.4 Mutation

Stub.

2.5 Population

Stub.

2.5.1 Initialization

Stub.

2.5.2 Reproduction

Stub.

2.6 Termination

Stub.

2.6.1 Criteria

Stub.

Chapter 3

SimPL

3.1 Overview

TODO: Change to past tense in regards to what was learned by SimPL.

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 will be used as a basis for a genetic algorithm needed to solve a harder problem of tuning a 3D physics engine.

3.2 Implementation

3.2.1 Arena

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

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

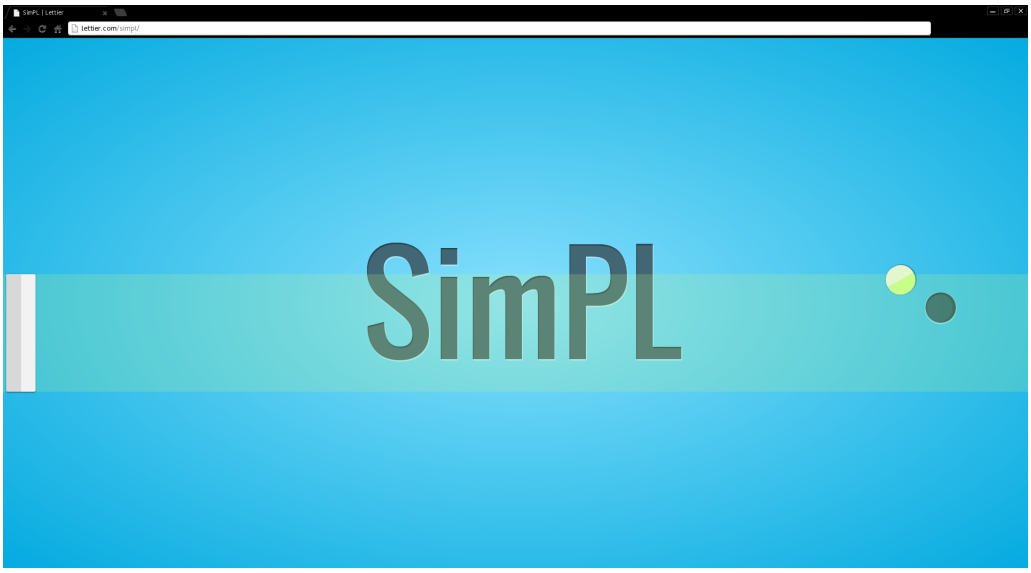

FIGURE 3.1: Here you see the SimPL arena containing the paddle and ball.

3.2.2 Ball

The ball is a physics-based dynamic object. Its starting position and starting velocity magnitude are the same at the start of every 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

[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 leaves the left side of the arena or at the time in which the ball's velocity magnitude drops below 100.

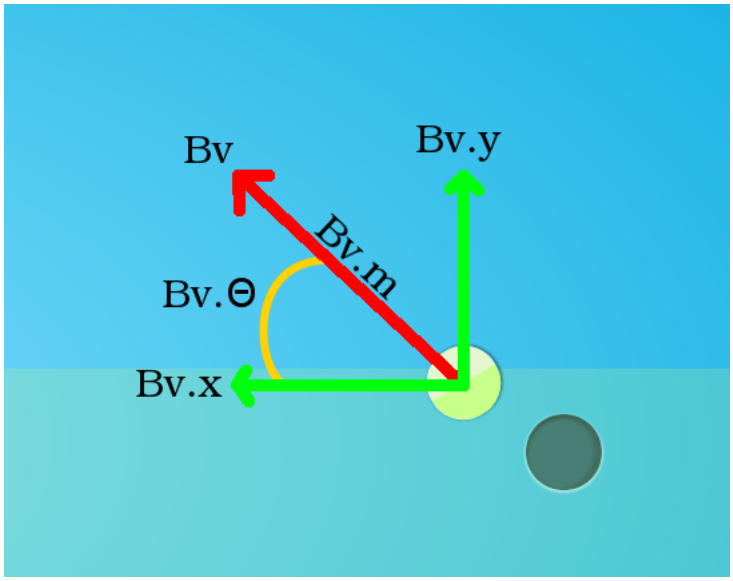

FIGURE 3.2: Here you see the ball's dynamic physics properties.

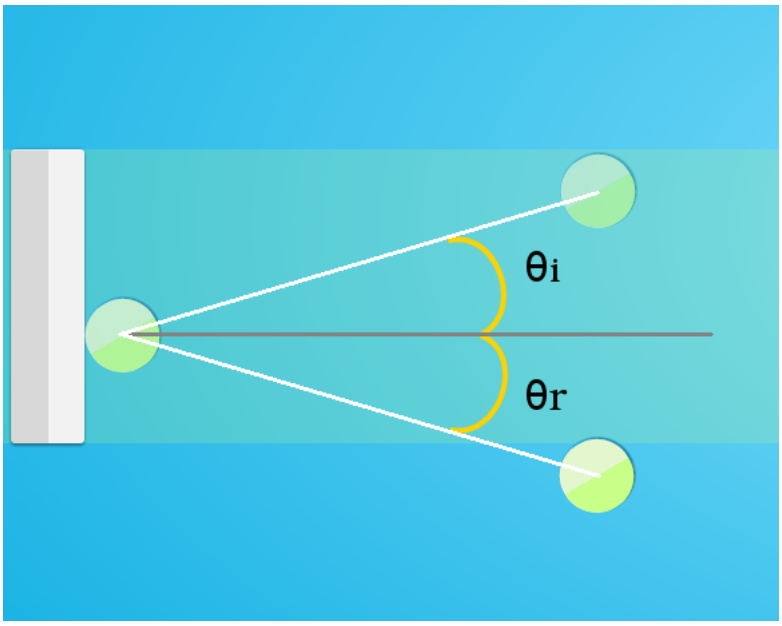

FIGURE 3.3: Here you see the ball's collision angle-of-incidence θ_i and its angle-of-reflection θ_r.

the round is over.

3.2.3 Paddle

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

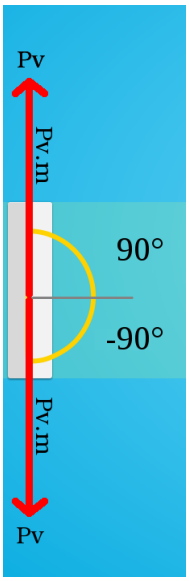

FIGURE 3.4: Here you see the paddle's dynamic physics properties.

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

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

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

3.2.4 Physics Engine

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

3.2.5 Neural Network

The neural network is a feed-forward neural network that contains one input layer consisting of six input nodes, one hidden layer consisting of 5 hidden nodes, and one output layer consisting of one output node [3]. 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 $[-1, 1]$.

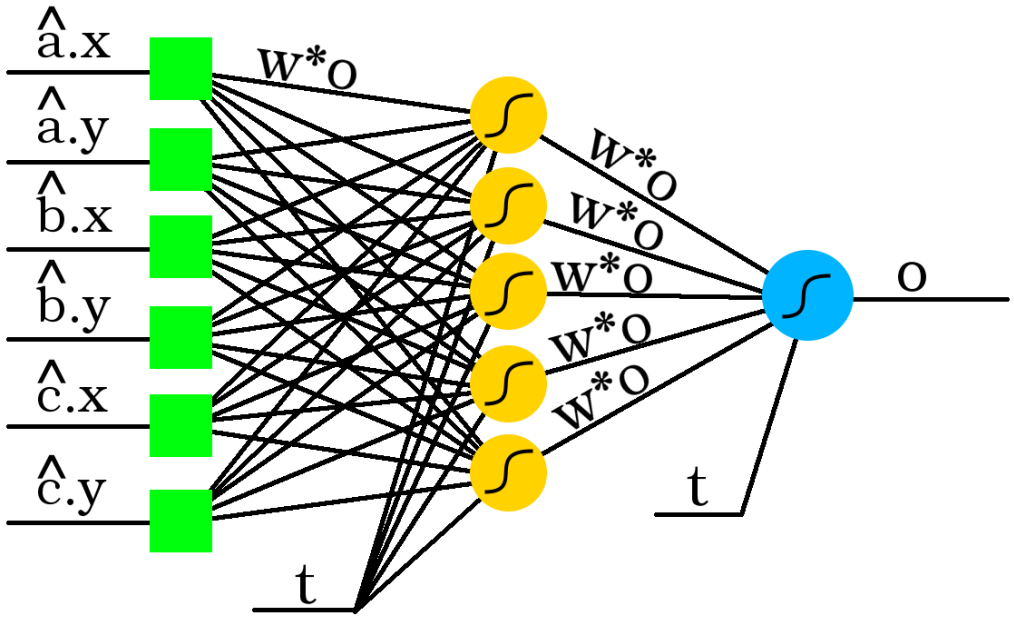

FIGURE 3.5: Here you see the neural network as constructed in SimPL.

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

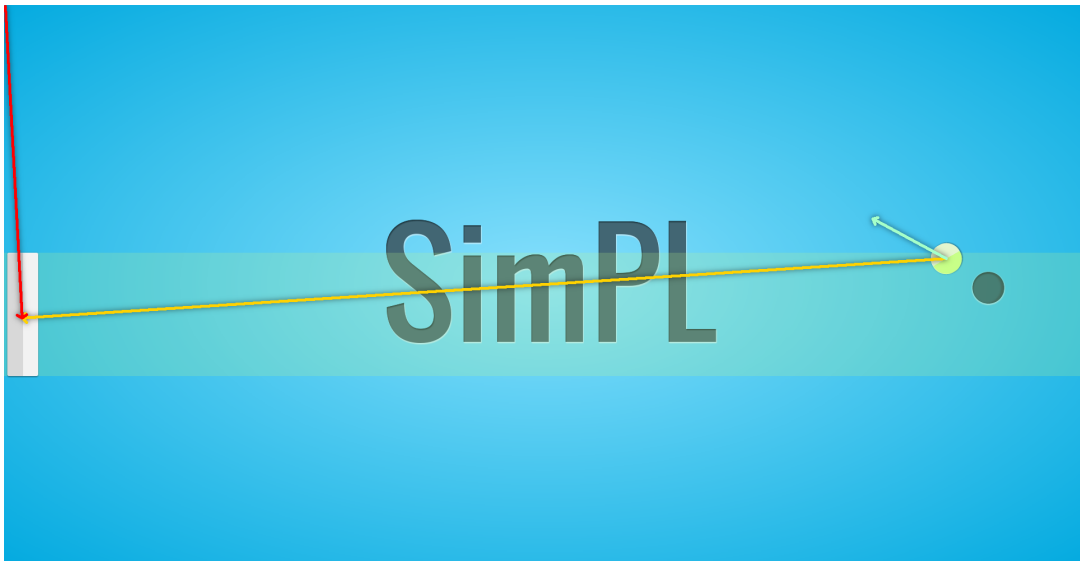

FIGURE 3.6: Here you see the three input vectors to the neural network. Note that these vectors are normalized thereby turning them into unit vectors.

3.2.6 Genetic Algorithm

TODO: Consider moving most of this to the GA chapter and only leave what was specific to SimPL.

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 [3]. 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

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

the input parameters to the mechanism the genetic algorithm is working to optimize. In the case of SimPL, the mechanism is the neural network and the parameters—of each genome—represent the 41 weights of the neural network. Each parameter has a valid range of $[-1, 1]$.

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

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

FIGURE 3.8: Here you see the basic genetic algorithm.

The selection operator uses roulette selection where the probability of some genome being selected

is proportional to their fitness. Roulette selection and roulette selection with rank fitness was experimented with as outlined later [4]. The crossover operator takes two selected genomes to produce offspring where the offspring have some combination of their parents' genes. SimPL uses one-point crossover throughout each experiment. To select the crossover point per crossover operation, the crossover operator samples a random integer from a uniform distribution ($crossoverPoint = X \sim U(0, n-1)$ where n is the number of genes per genome). The mutation operator takes a selected genome and mutates its genes using various means. If the mutation scope is at the gene level, the mutation operator traverses through the genome's gene array where for each gene, the mutation operator samples a random integer from a uniform distribution ($X \sim U(0,1)$) and if this random integer X is less than or equal to the mutation probability, the mutation operator mutates the gene value either via uniform mutation ($geneValue_i = X \sim U(0,1) * \varepsilon$) or Gaussian mutation ($geneValue_i = X \sim N(\mu, \sigma^2)$). If the mutation scope is at the genome level, the mutation operator samples a random integer from a uniform distribution ($X \sim U(0,1)$) and if this random integer X is less than or equal to the mutation probability, the mutation operator mutates all of the genome's gene values one after the other via either uniform mutation ($geneValue_i = X \sim U(0,1) * \varepsilon$) or Gaussian mutation ($geneValue_i = X \sim N(\mu, \sigma^2)$). These means of mutation were experimented with as outlined later. Crossover and mutation can be carried out in sequence or can be done separately with one operator not interfering the other operator's offspring. This was experimented with as outlined later.

Crossover and mutation are not guaranteed to always occur as the genetic algorithm produces new genomes from the old genomes. Rather, crossover and mutation occur based on some probability. If the crossover and mutation probabilities are both set to 1 then they always occur. Before the genetic algorithm is initialized, the crossover and mutation probabilities are set [5]. These initial crossover and mutation probability values can be arbitrary or can be based on some a-priori knowledge of the fitness landscape. The probabilities can change over time or can remain static throughout the run of the genetic algorithm. Different static settings and self-adaptation of the probabilities were experimented with as outlined later.

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

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

3.2.7 Database Manager

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

3.3 Platform

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

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

wide by 50 pixels tall and the actual size was the same. The paddle had $681 - 200 = 481$ pixels of available screen-space to traverse.

3.4 Experimental Designs

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

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

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

GA Parameters × Experiment	*One*	*Two*	*Three*	*Four*	*Five*	*Six & Seven*
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: Here you see the genetic algorithm parameters used per experiment. The highlighted cells indicate the experimental variables per experiment. Experiment six and seven are included in the table for completeness but no evolution ever took place.

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

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

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

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

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

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

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

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

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

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

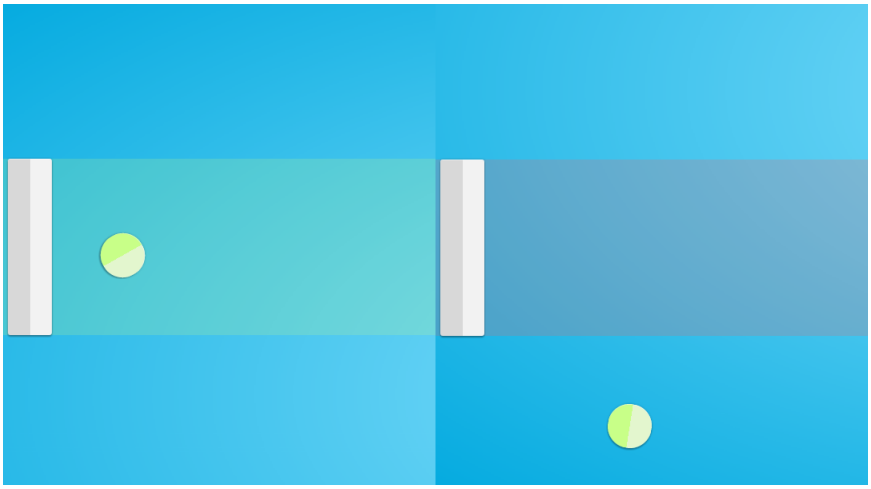

FIGURE 3.9: Here you see the simplified fitness function graphically where the paddle on the left is gaining fitness (as indicated by the green-hued bar) while the paddle on the right is gaining no fitness (as indicated by the purple-hued bar).

Instead of using the actual fitness of any particular genome in the population to determine its probability of being selected during the roulette wheel selection process, a genome's rank fitness was used to determine its probability of being selected [4]. 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 [6]. After sorting the population, genome one is given a fitness of one, genome two is given a fitness of two and so on and so forth until genome n is given a fitness of n or rather the population size. See Figure 3.10. Now all genomes have a better chance of being selected to undergo crossover and/or mutation thereby keeping the population diversity high and thus keeping the search scope of the fitness landscape large resulting in better performance of the genetic algorithm.

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

Based on empirical studies performed by others, setting the mutation probability to $\frac{1}{n}$—where n is the number of genes per genome—is a sufficient enough amount of random search to allow the genetic algorithm to escape local maxima in the fitness landscape [7]. 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

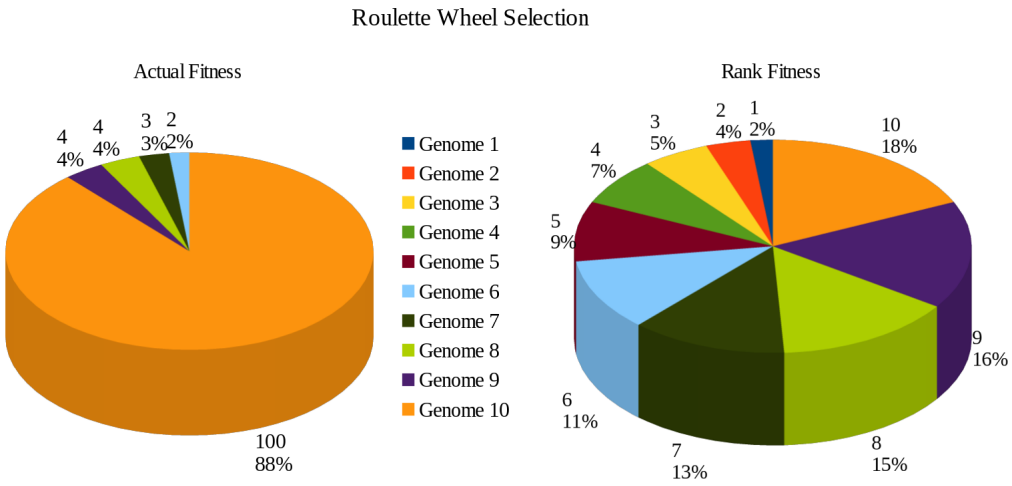

FIGURE 3.10: Here you see roulette selection using actual fitness versus rank fitness for a population of 10 genomes. On the left, the integers are the genomes' actual fitness and the percentages are their probabilities of being selected. On the right, the integers are the genomes' rank fitness and the percentages are the genomes' probabilities of being selected. Observe that Genome 10 no longer dominates the wheel when using its rank fitness (10) versus using its actual fitness (100).

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

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

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

As outlined in [5], the crossover and mutation probabilities self-adapt based on the crossover and mutation operators' ability to produce fitter genomes from one generation to the next. To facilitate the self-adaptation, the crossover and mutation operators' viability to produce fitter and fitter offspring needed to be tracked from one generation to another. As the operators were being evaluated on their own accord, they were not allowed to interfere with each others' offspring and thus they were not run in sequence but rather were run in parallel. With each new generation produced, elite offspring were marked accordingly as well as offspring produced by crossover and offspring produced by mutation. For those offspring that were created by crossover, their parents' weighted-mean fitness was annotated along with their offspring. Annotated meaning it was recorded in the offspring's data structure later being used to calculate the crossover operator's progress. This weighted-mean fitness was calculated based on the crossover point which determines what percentage of genes came from parent one and what percentage of genes came from parent two. For example, let there be 41 genes per genome and let the crossover point be gene 10. Thus, offspring one received 10 genes from parent one and received 31 genes from parent two while offspring two received 10 genes from parent two and received 31 genes from parent one. Therefore, offspring one's weighted-mean parent fitness would be calculated as $\overline{PF} = \left(PF_1 * \frac{10}{41}\right) + \left(PF_2 * \frac{(41-10)}{41}\right)$ 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. In other words, for example, if the crossover operator produced fitter genomes greater than the mutation operator did for the last generation, then the probability of creating offspring via crossover would be increased and the probability of creating offspring via mutation would be decreased for the creation of the next generation. Adjustment of the crossover and mutation probabilities were handled by the adjustment parameter. Let the adjustment parameter be denoted as θ. Here θ was self-adjusted as well, as outlined in [5]. Once the probabilities were adjusted they were clamped to the range $[0.001, 1.0]$. With a minimum probability of 0.001, there would always be some crossover and/or mutation albeit not much. See Figure 3.11. Note that to insure a level playing field, both the crossover and mutation probabilities were set to an initial value of 0.5 before the start of the genetic algorithm.

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

```
BEGIN
    Population P has been evaluated
    cCount = mCount = 0
    CP_sum = MP_sum = 0
    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: Here you see the self-adaptation algorithm.

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

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

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

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

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

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

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

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

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

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

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

3.4.7 Experiment seven: random paddles.

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

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

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

where $m_{initial}$ was the paddle's starting velocity magnitude (specifically 1000 $\frac{pixels}{second}$). 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 $m_{new} = X * m_{initial}$. Otherwise, if X was zero, the paddle did not move from its current position.

3.5 Experimental Results

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

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

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

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

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

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

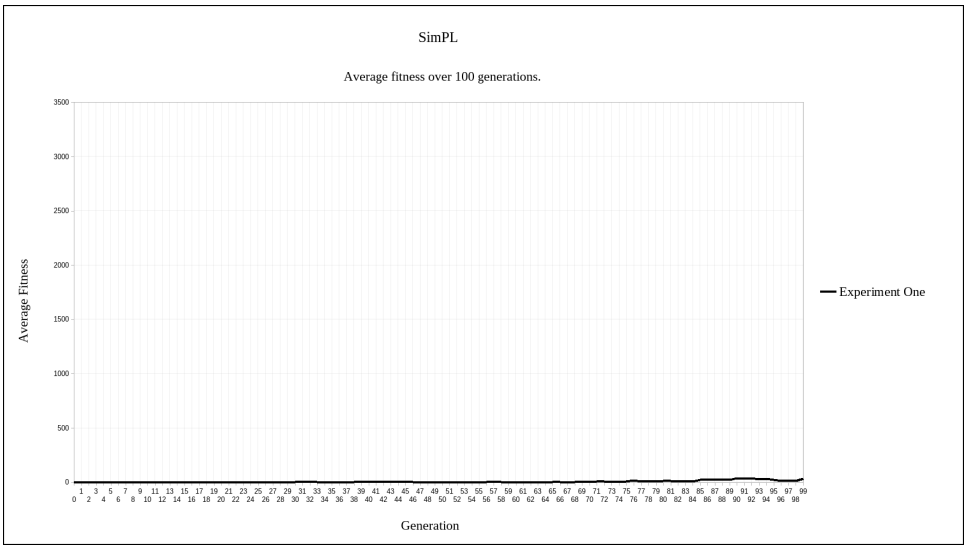

FIGURE 3.12: Here you see the average fitness over 100 generations for experiment one.

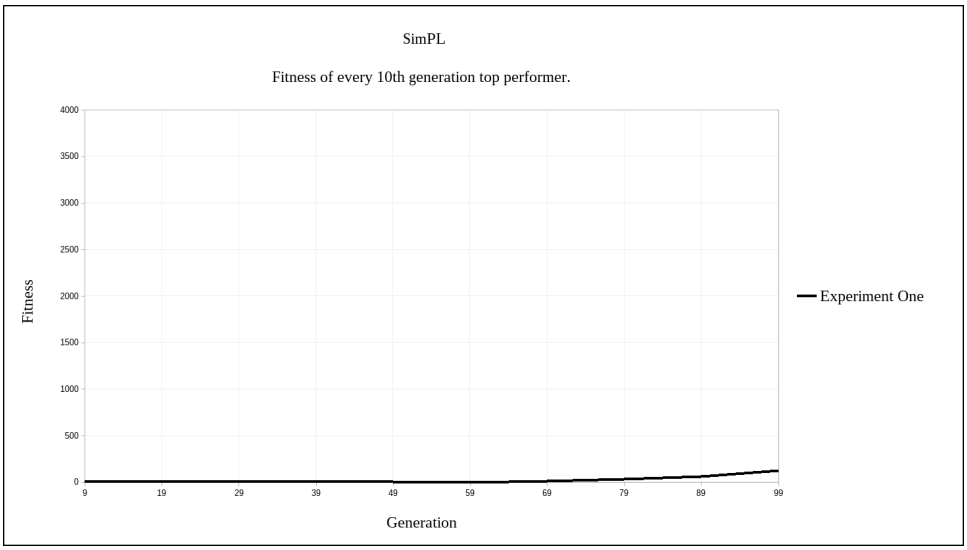

FIGURE 3.13: Here you see the fitness of every 10th generation top performer for experiment one.

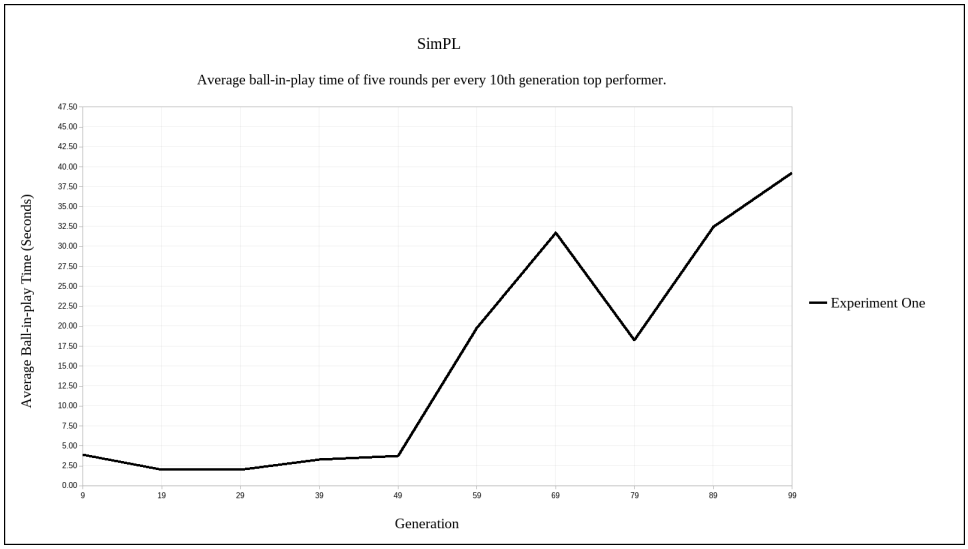

FIGURE 3.14: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment one.

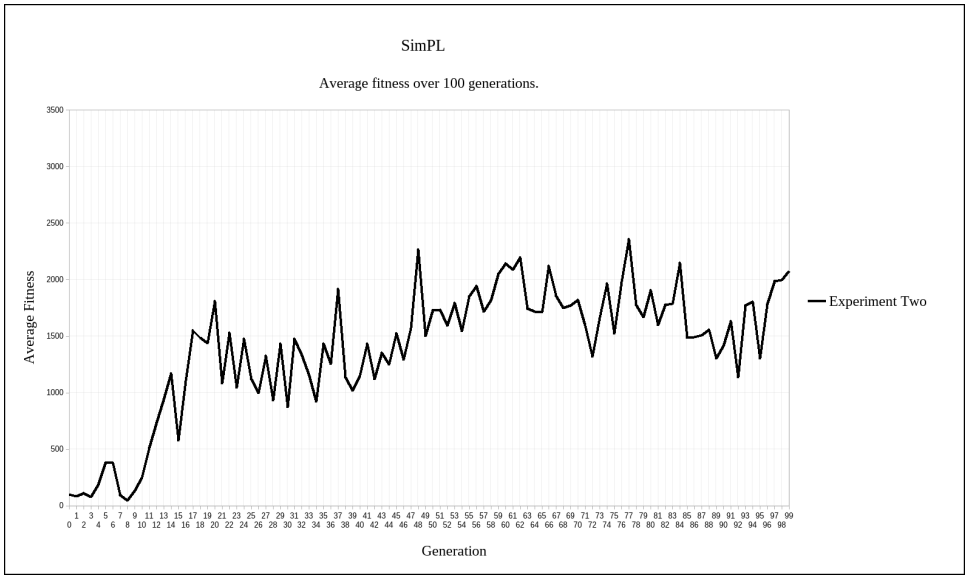

FIGURE 3.15: Here you see the average fitness over 100 generations for experiment two.

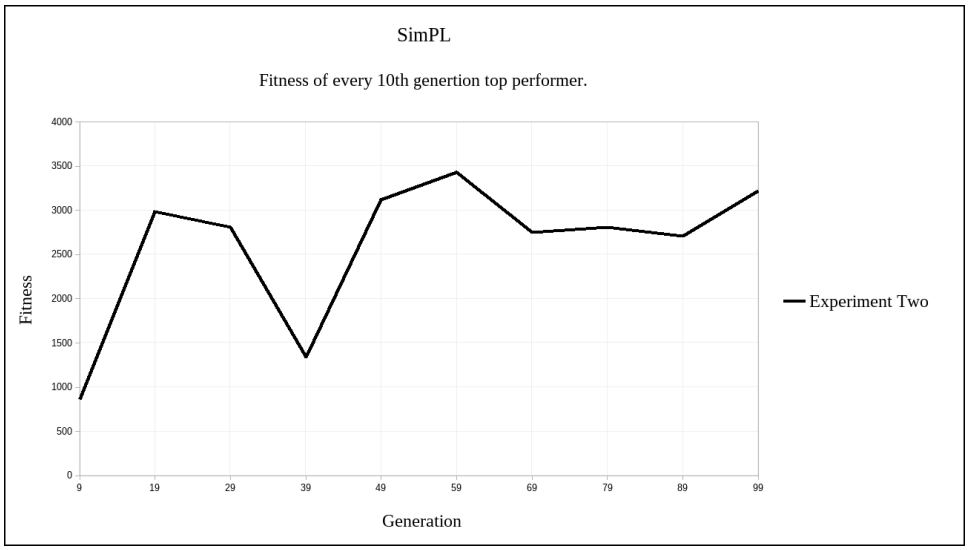

FIGURE 3.16: Here you see the fitness of every 10th generation top performer for experiment two.

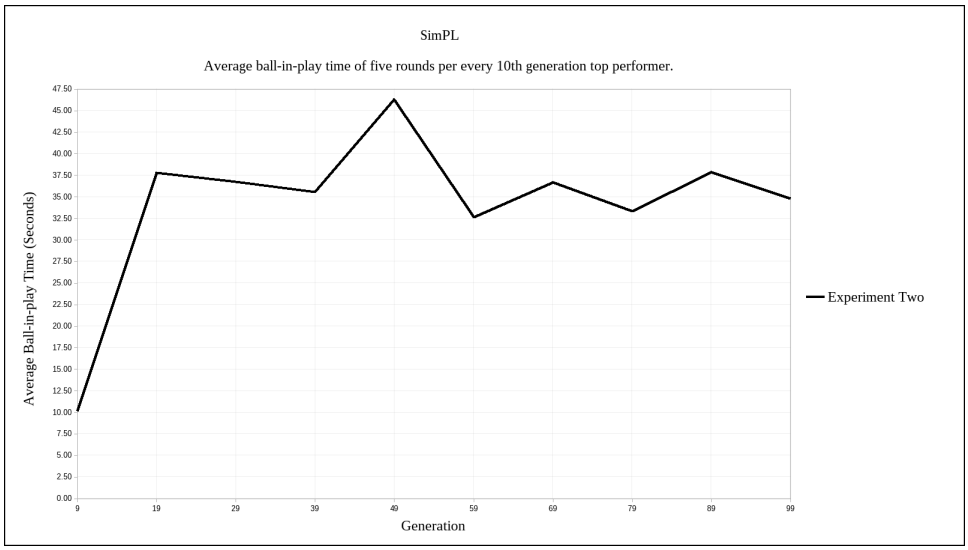

FIGURE 3.17: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment two.

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

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

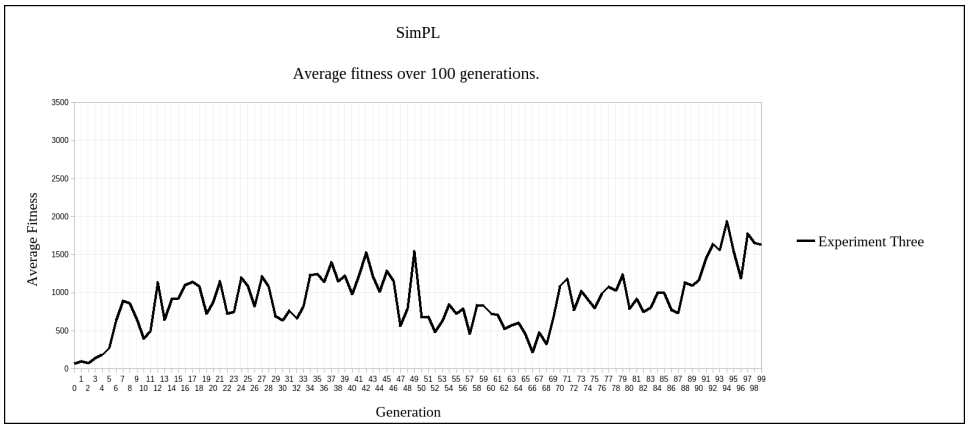

FIGURE 3.18: Here you see the average fitness over 100 generations for experiment three.

The crossover and mutation probabilities were both initially set to 0.5 before the start of the experiment. After the experiment was over, the genetic algorithm self-adapted the crossover probability to 0.7816 and self-adapted the mutation probability to 0.2184. You'll notice in Figure 3.19 that the mutation probability overtook the crossover probability at first but the two probabilities eventually diverged with mutation becoming less probable and crossover becoming more probable as the genetic algorithm produced fitter generations. This outcome is almost the exact opposite of the outcome shown in [5].

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

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

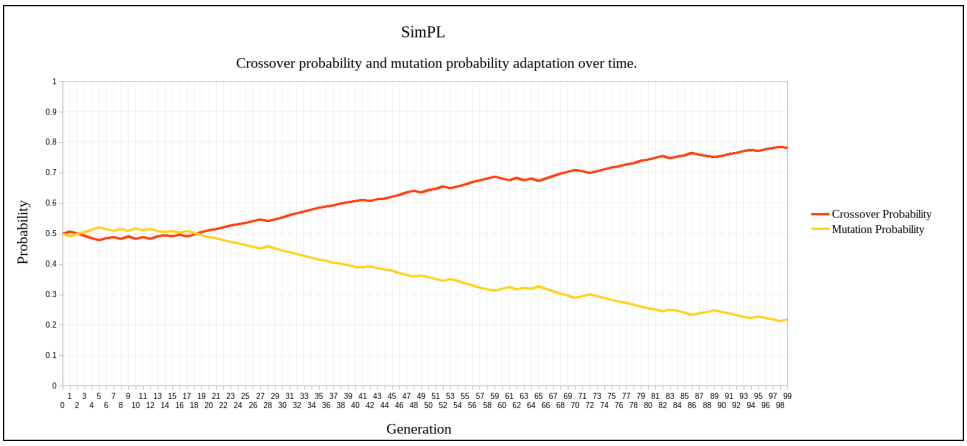

FIGURE 3.19: Here you see the self-adaptation of the crossover and mutation probabilities for experiment three.

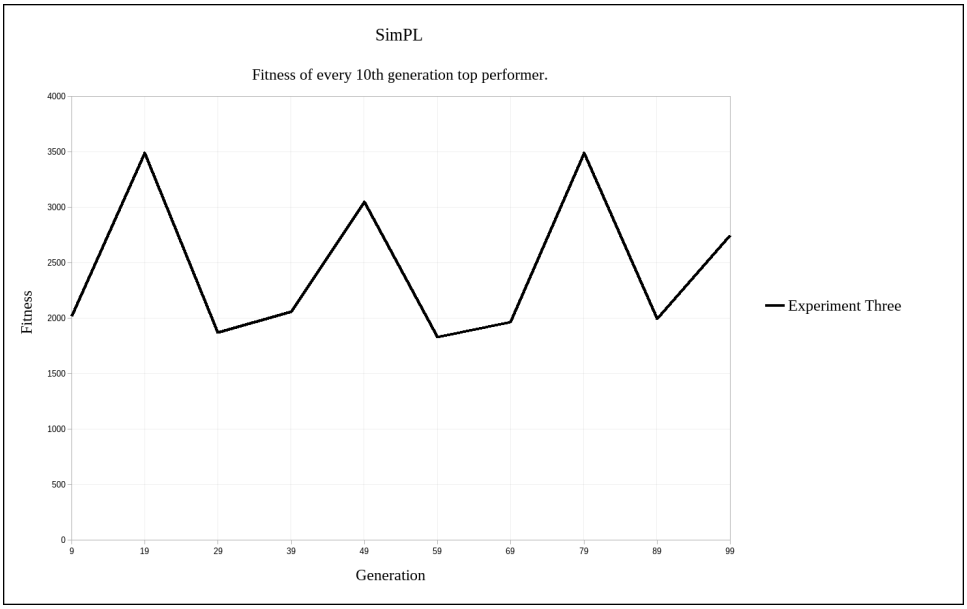

FIGURE 3.20: Here you see the fitness of every 10th generation top performer for experiment three.

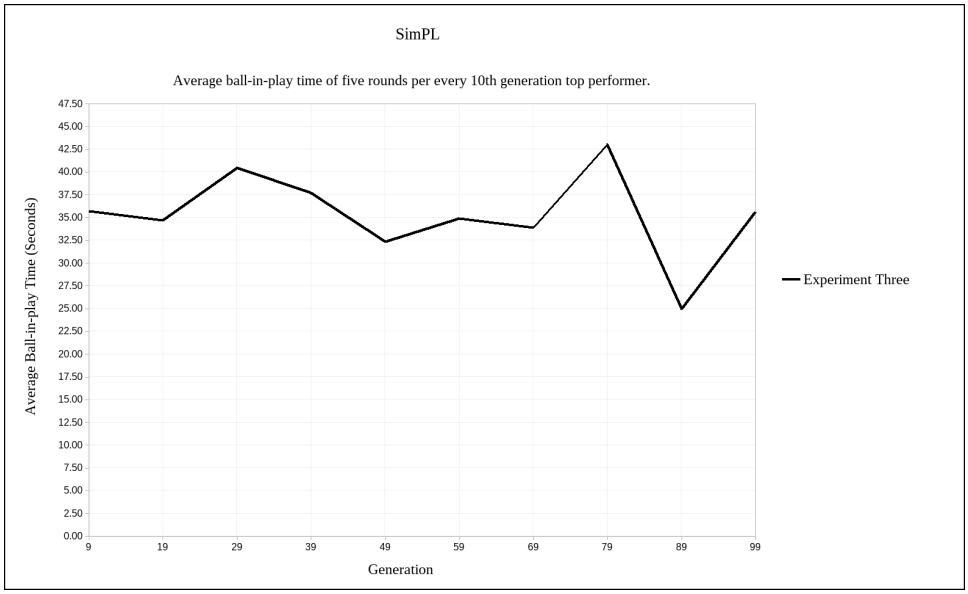

FIGURE 3.21: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment three.

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

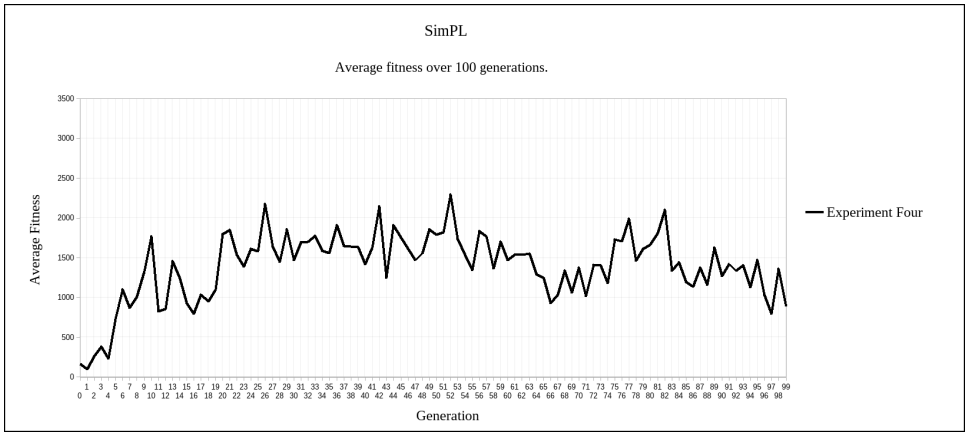

FIGURE 3.22: Here you see the average fitness over 100 generations for experiment four.

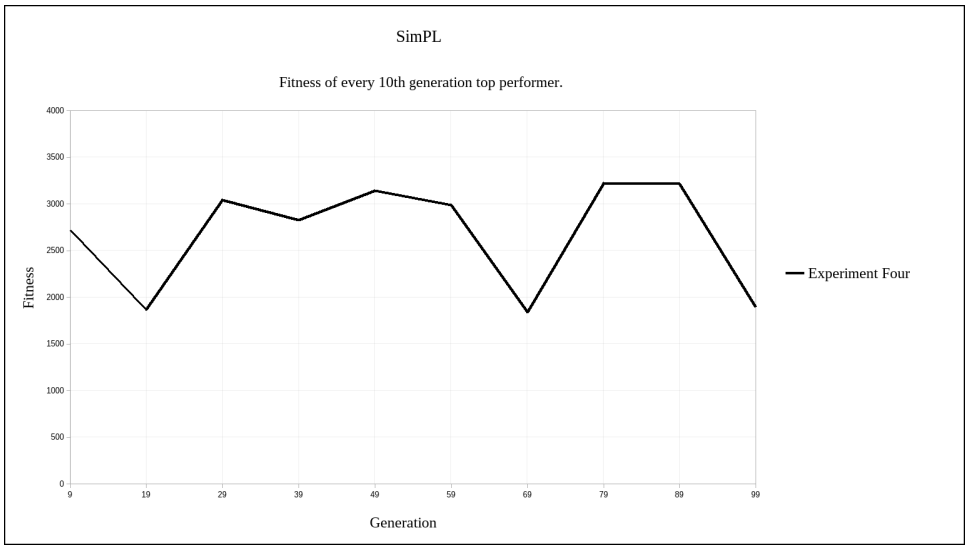

FIGURE 3.23: Here you see the fitness of every 10th generation top performer for experiment four.

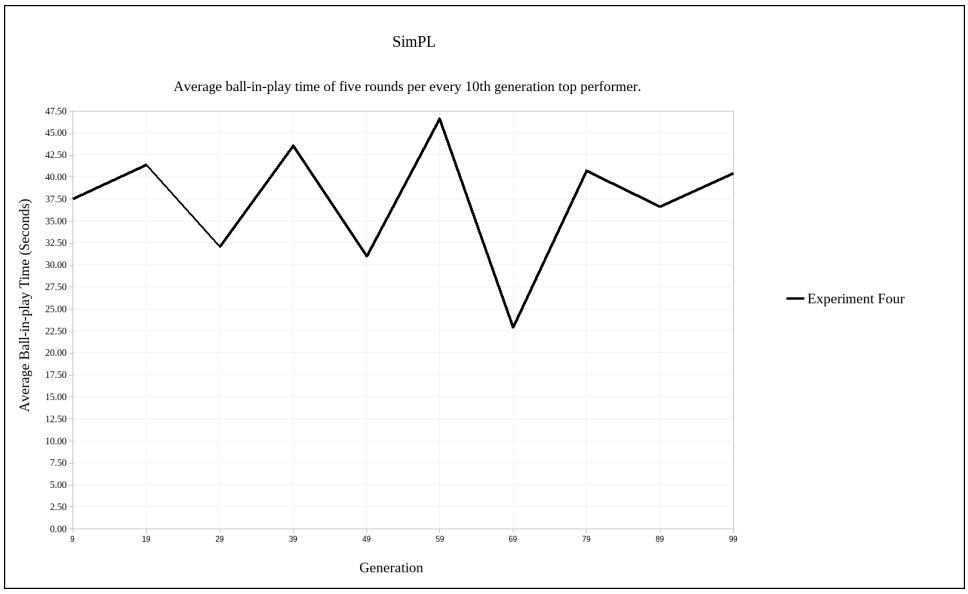

FIGURE 3.24: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment four.

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

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

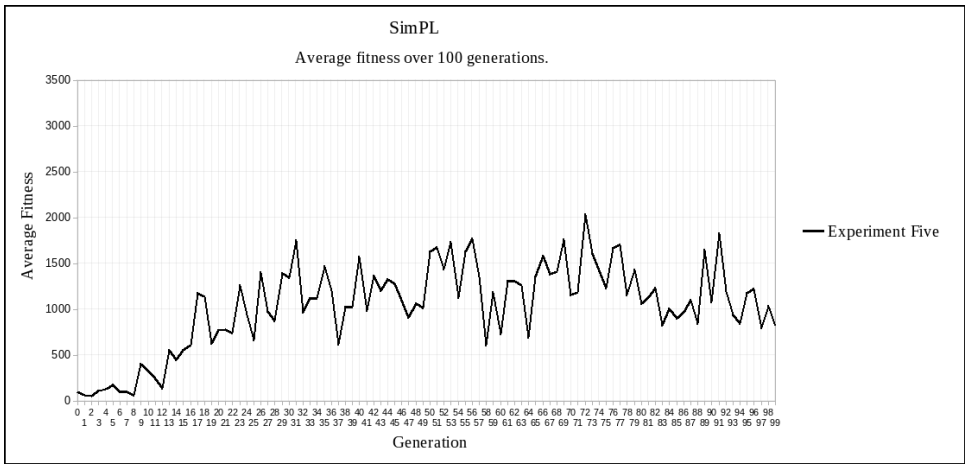

FIGURE 3.25: Here you see the average fitness over 100 generations for experiment five.

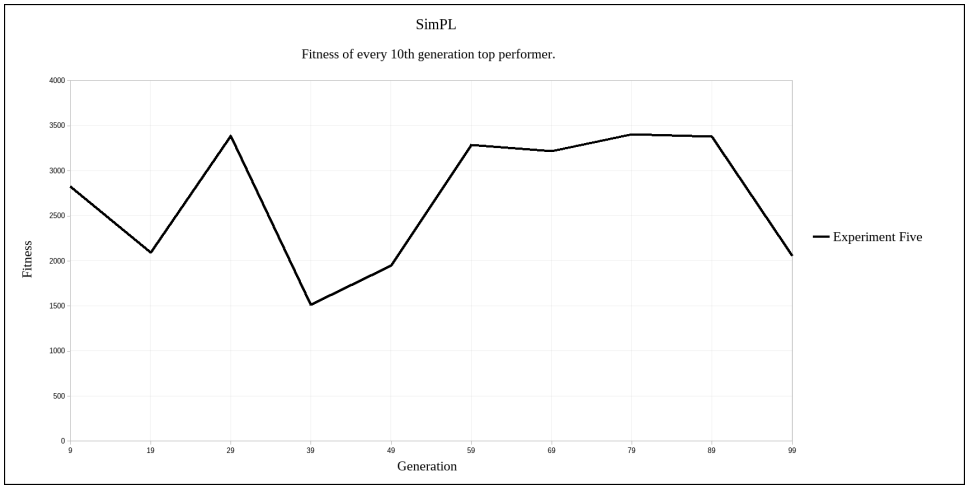

FIGURE 3.26: Here you see the fitness of every 10th generation top performer for experiment five.

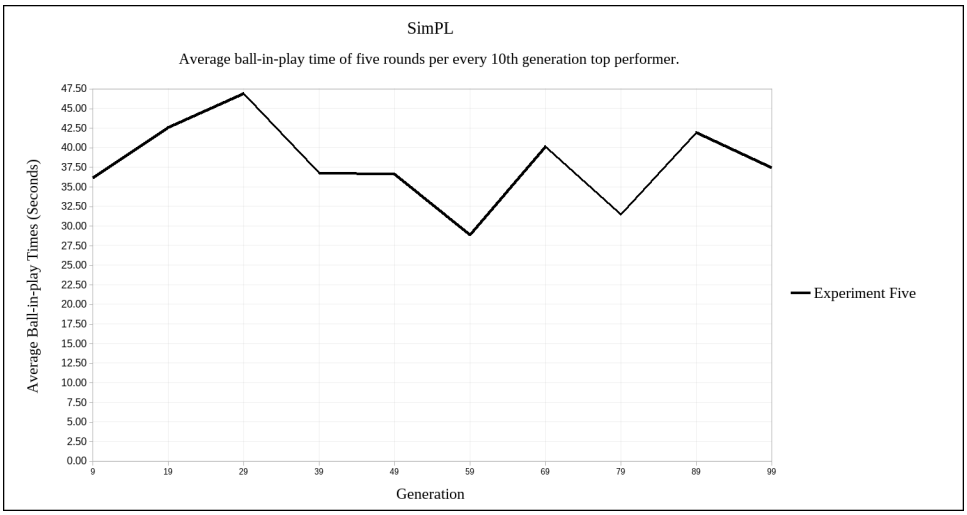

FIGURE 3.27: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment five.

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

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

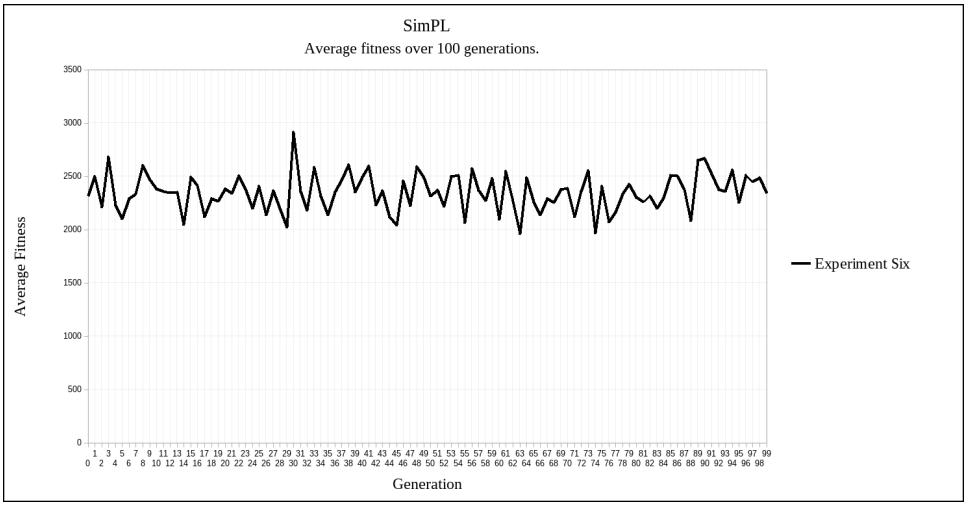

FIGURE 3.28: Here you see the average fitness over 100 generations for experiment six.

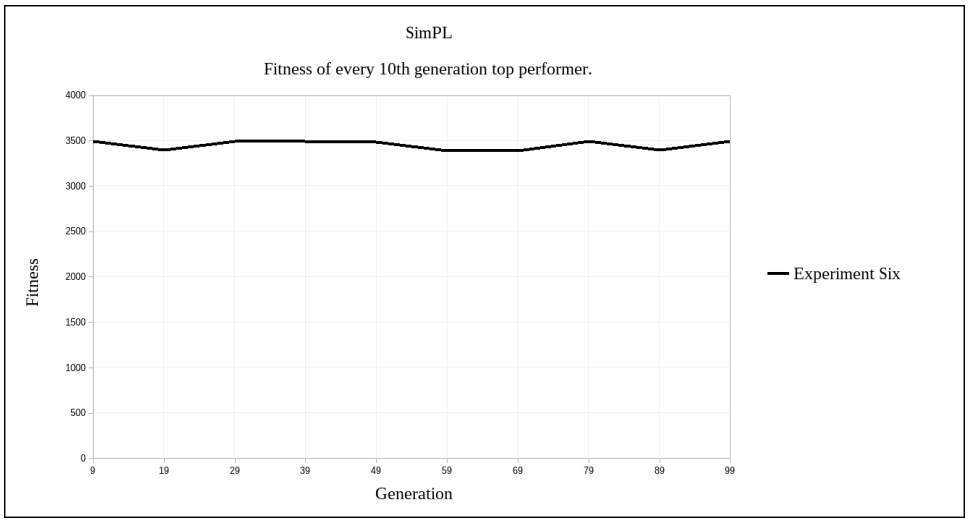

FIGURE 3.29: Here you see the fitness of every 10th generation top performer for experiment six.

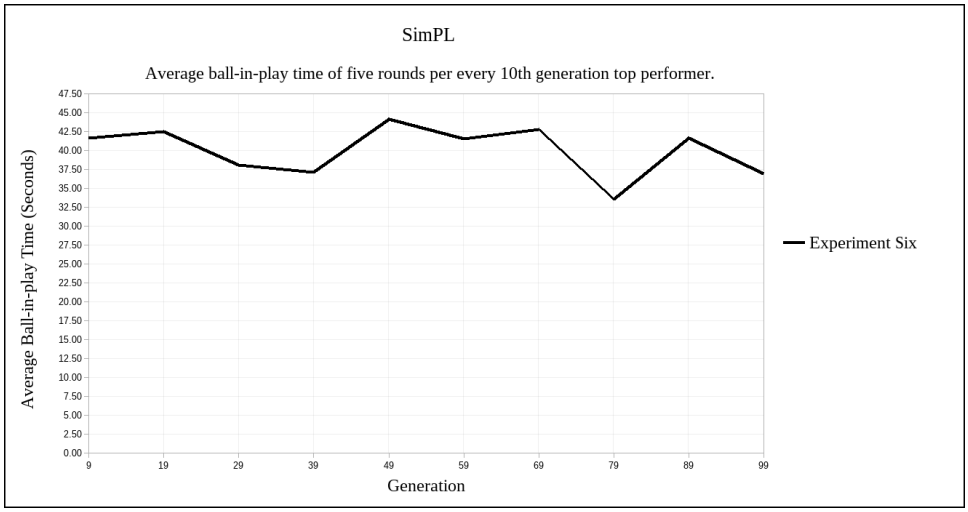

FIGURE 3.30: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment six.

3.5.7 Experiment seven: random paddles.

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

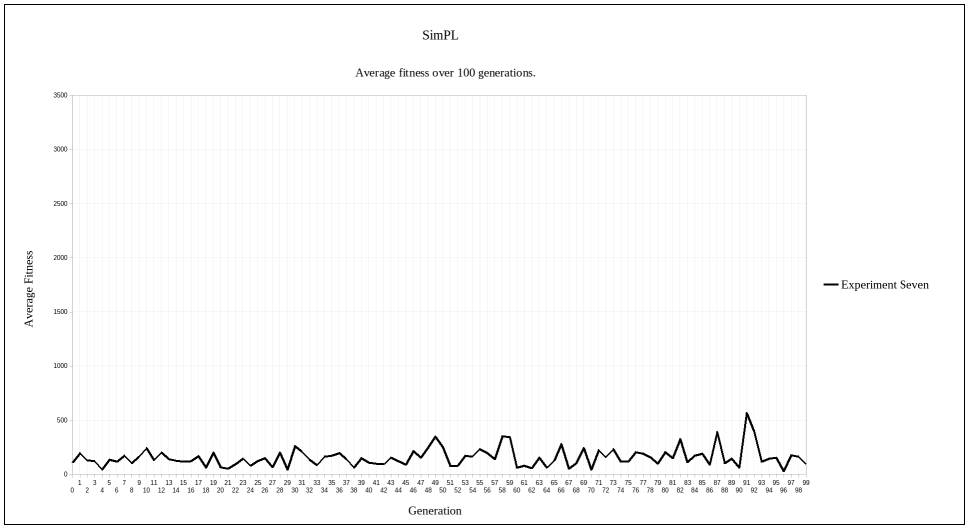

FIGURE 3.31: Here you see the average fitness over 100 generations for experiment seven.

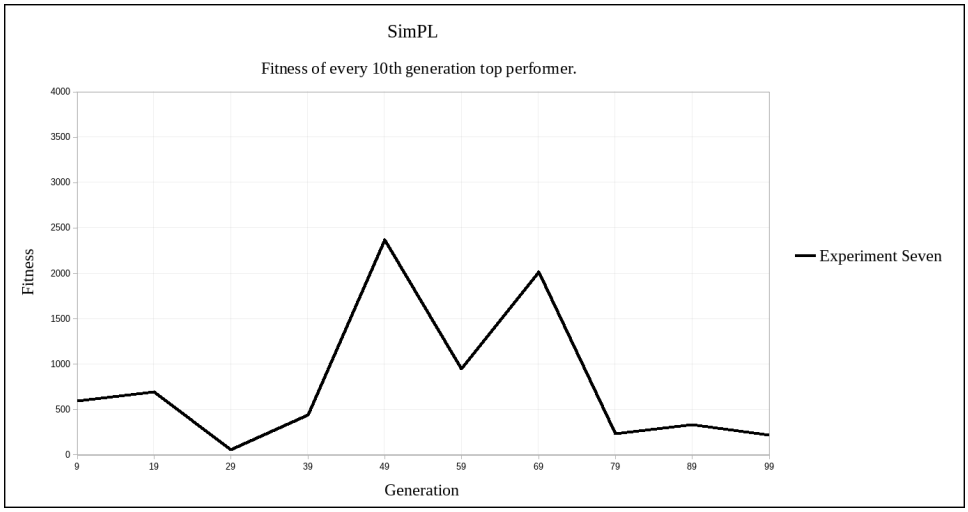

FIGURE 3.32: Here you see the fitness of every 10th generation top performer for experiment seven.

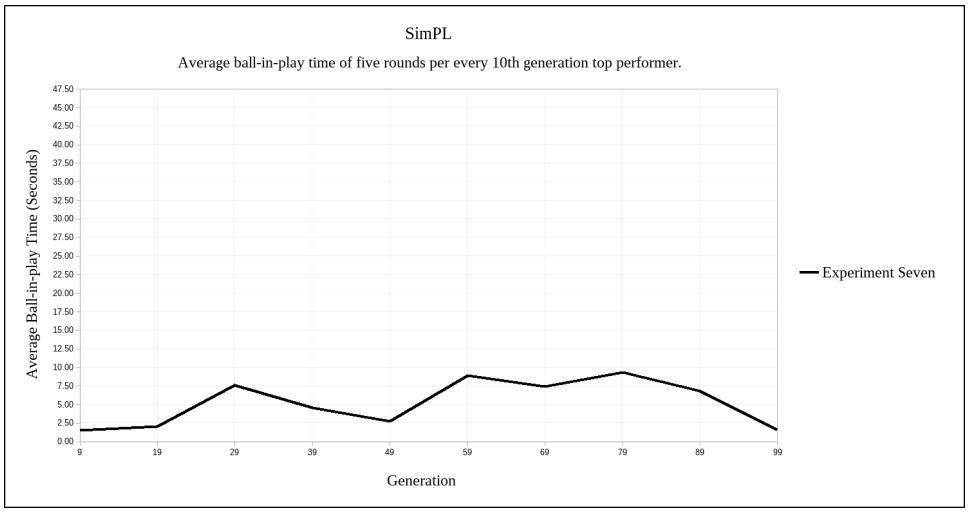

FIGURE 3.33: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment seven.

3.5.8 Comparative Results

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

3.5.8.1 Average fitness over 100 generations.

See Figure 3.34.

3.5.8.2 Fitness of every 10th generation top performer.

See Figure 3.35.

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

See Figure 3.36.

3.6 Conclusion

TODO: Change the wording to reflect that it *was* helpful to do SimPL since BBAutoTune was successful.

The genetic algorithm developed for SimPL should prove to be a robust basis for the genetic algorithm needed to solve a harder problem of tuning a 3D physics engine (project BBAutoTune). The principles and techniques of evolutionary algorithms learned during the SimPL project will certainly carry over to the

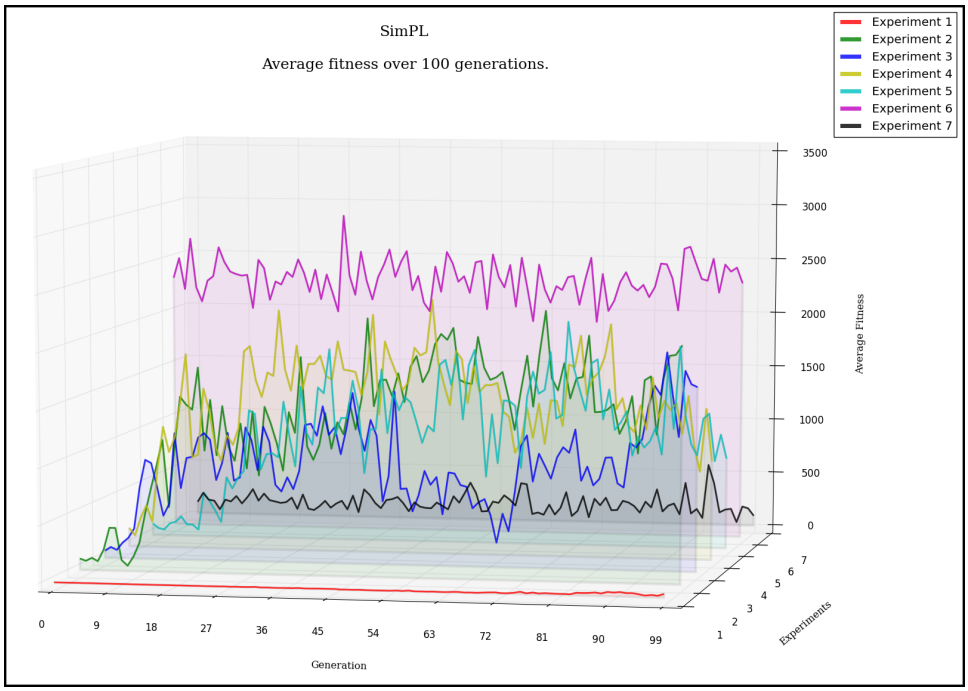

FIGURE 3.34: Here you see the average fitness over 100 generations for experiments one through seven.

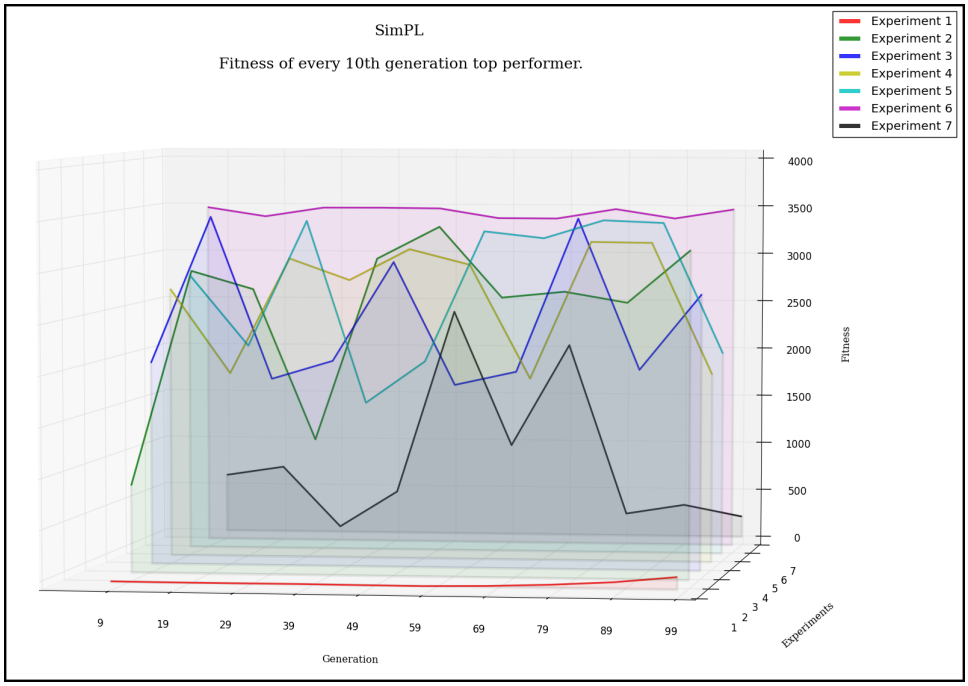

FIGURE 3.35: Here you see the fitness of every 10th generation top performer for experiments one through seven.

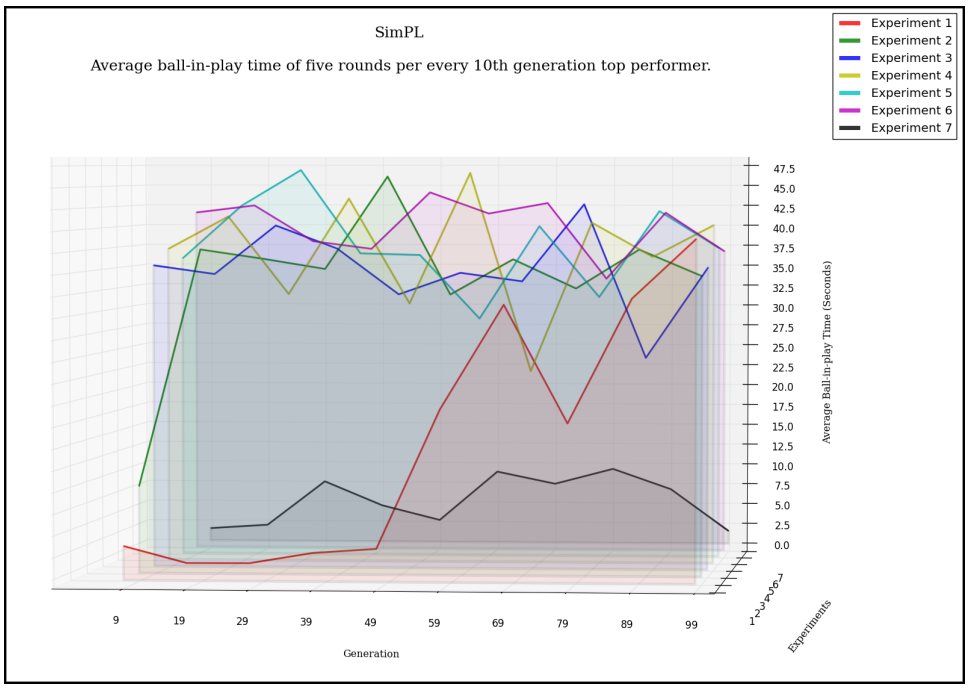

FIGURE 3.36: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiments one through seven.

more difficult project, BBAutoTune. And while the problem domain of SimPL and BBAutoTune are only somewhat similar, the problems faced and worked out during the development of SimPL should alleviate problems faced while developing BBAutoTune. As the results show, the genetic algorithm for SimPL performed well, producing neural network weight solutions that had the paddle keeping the ball in the arena for almost a minute. Had it not been for the round termination criteria of the ball's velocity magnitude dropping below 100, most of the paddles (with high fitnesses) would have kept the ball in the arena indefinitely. Thus, the goal to learn about and to cultivate a genetic algorithm capable of tuning parameters with respect to a fitness landscape was certainly accomplished.

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	$X \sim U[-1, 1] *$ Max-Perturbation
Max-Perturbation	0.3
Sequential Crossover and Mutation	True
Fitness Function	Partial/Full
Fitness Credit for Hitting the Ball	1.0
Fitness Credit for Following the Ball	0.1
Fitness Credit for Matching the Ball's Center Y-coordinate	0.1
Fitness Credit for not Following the Ball	-0.1

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 4

BBAutoTune

4.1 Overview

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

4.2 Implementation

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

4.2.1 Surveyor SRV-1 Blackfin 3D Model

The robot used during experimentation was a 3D model of the Surveyor SRV-1 Blackfin. This model is dimensionally and aesthetically based on the real robot. The extents of the model are $17.64cm \times 14.53cm \times 14.33cm$ including the braille hat the rests above the body of the robot. The base and the wheels are the only physics based objects on the model with the wheels being connected to the base via a rigid body hinge joint. See Figure 5.5.

4.2.2 GUI

Blender's graphical user interface can be extended via its Python API. BBAutoTune adds a custom panel to Blender that allows the user to specify certain parameters pertaining to the GA and the overall tuning loop. Once the user specifies their preferred parameters, they press start to begin the tuning process. After pressing start, BBAutoTune will continue to run until the GA has reached the max generations specified. See Figure 4.1.

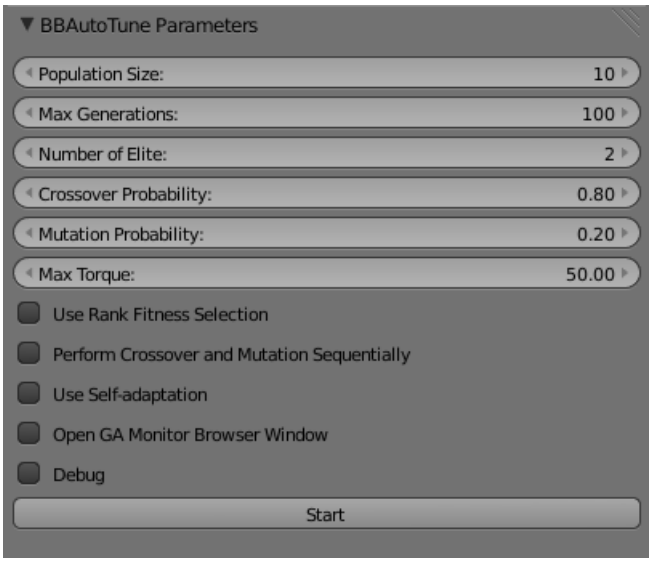

FIGURE 4.1: Here you see the BBAutoTune GUI panel added to Blender.

4.2.3 Physics Engine API

Blender exposes the Bullet[1] physics engine API via its own Python API. Most if not all of the parameters to Bullet can be modified via the physics engine API during or before running the Blender game engine. There are 41 different physics engine parameters that can be set via the API.

4.2.3.1 Parameter Set and Ranges

See Table 4.1.

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

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

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

4.2.4 Database Manager

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

4.2.5 Robot Monitor

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

4.2.6 Progress Monitor

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

4.2.7 Genetic Algorithm

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

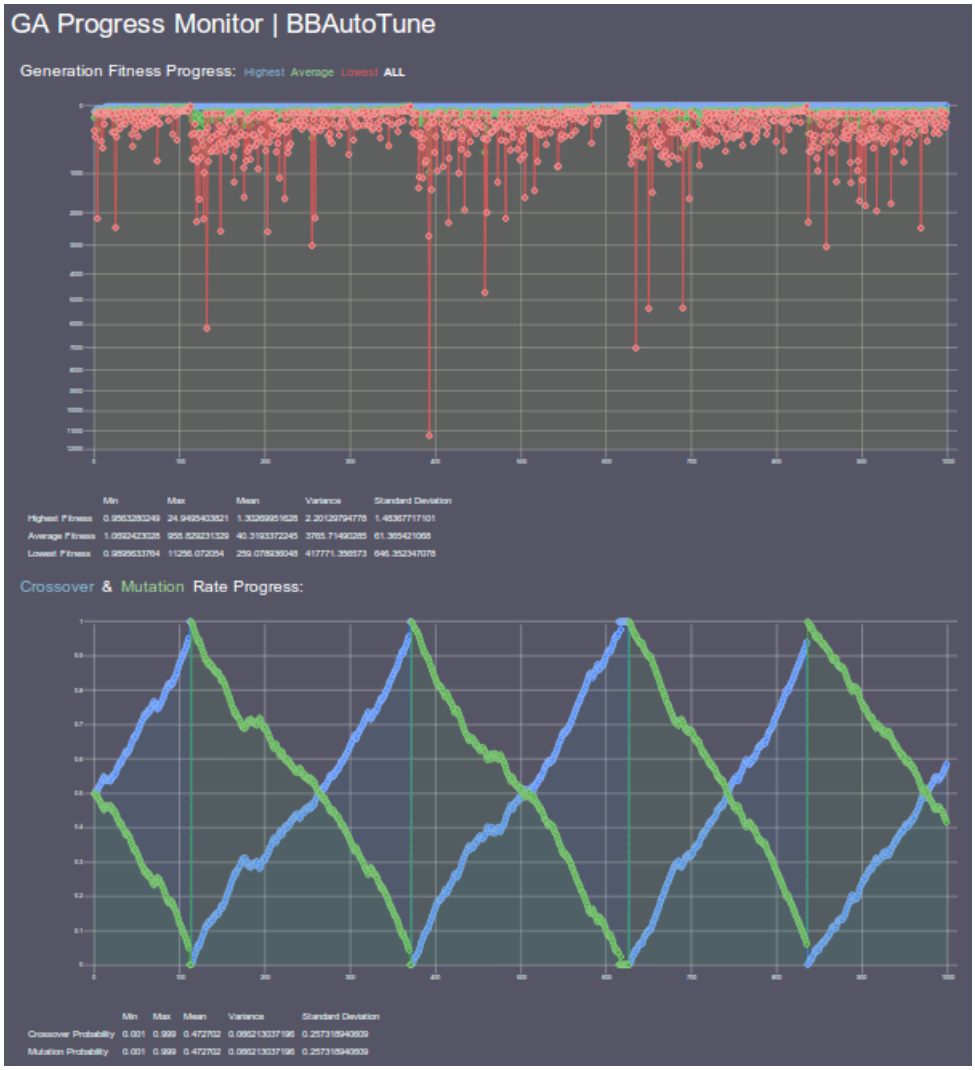

FIGURE 4.2: Here you see external GA progress monitor.

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

4.2.7.1 Encoding Scheme

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

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

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 [8]. 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 Figure 4.3.

```
BEGIN
    Population P with size n has been evaluated
    Sort P in non-increasing order
    For i=1 to n do
        P[i − 1].rankFitness = i
    End for
    For i=0 to n − 1 do
        P[i].prefixSum = 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 4.3: Here you see the rank fitness selection algorithm.

4.2.7.3 Fitness Function

TODO: This section only reflects the data recorded using the old move() function. Change to reflect data collected using the new motion() function.

To construct the fitness function, real robot motion data was collected which included 1040 sample points. X-translation, y-translation, and z-rotation were recorded for the real robot. With a camera overhead,

the real robot was place in the arena and was repeatedly commanded to go forward (relative from its current position and orientation) 25 centimeters. Care was taken to avoid having the robot collide with the arena walls. Once the robot consecutively moved forward three times, the robot rotated in place by 135 degrees and was not recorded during this period of rotation.

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

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

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

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

With the real robot motion data collected and analyzed, a metric was needed for the fitness function.

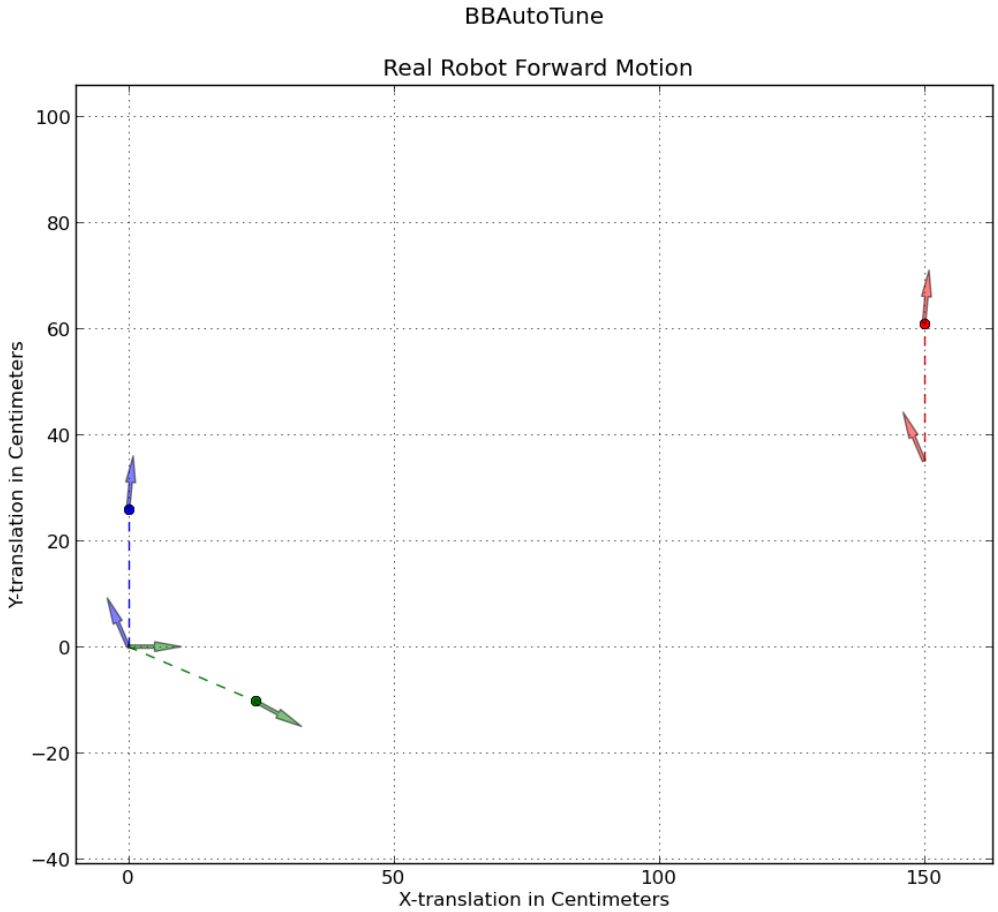

FIGURE 4.4: Here you see a specific instance of how the raw real robot forward motion data was translated and rotated such that each pair of initial and final states have the same reference frame. The red arrows, dot, and line represent the raw initial and final state of the robot recorded from the overhead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

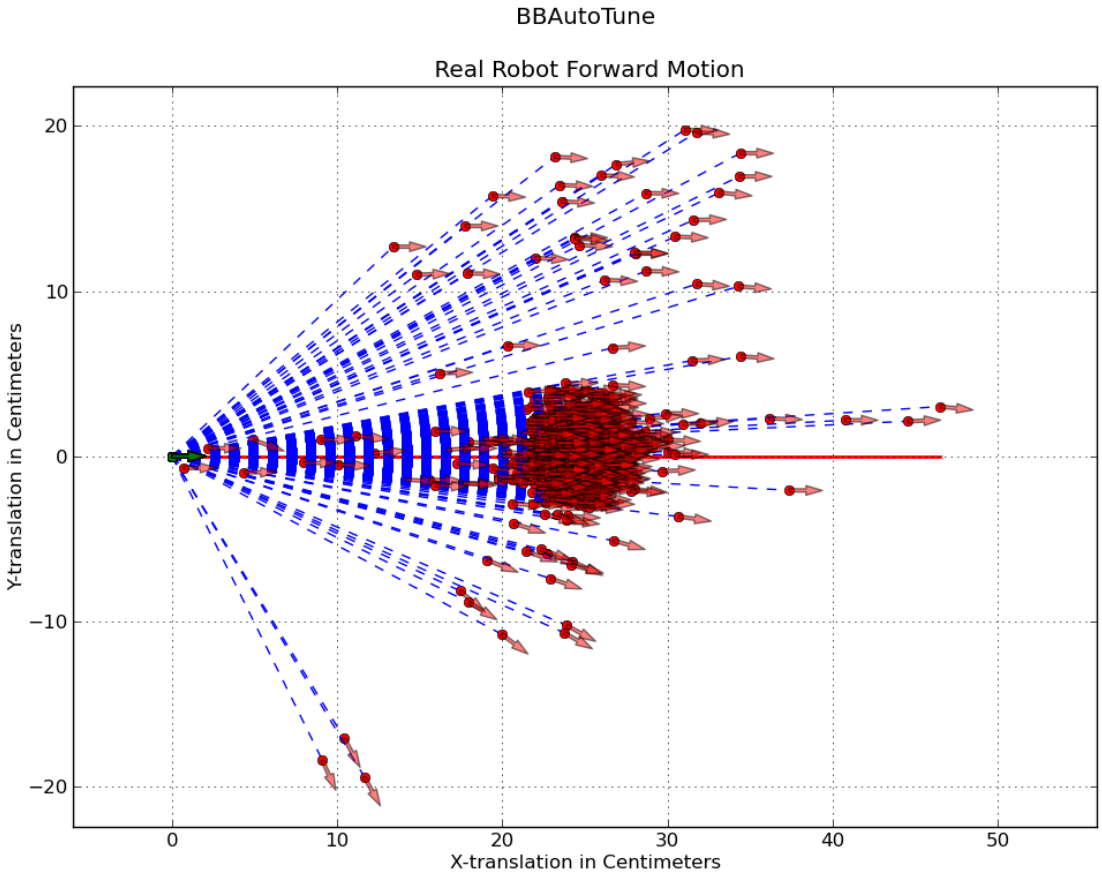

FIGURE 4.5: Here you see the collected real robot forward motion plotted from the same reference point. The green dots and arrows indicate the starting position and orientation of the real robot before the forward command was performed. The red dots and arrows indicate the ending position and orientation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

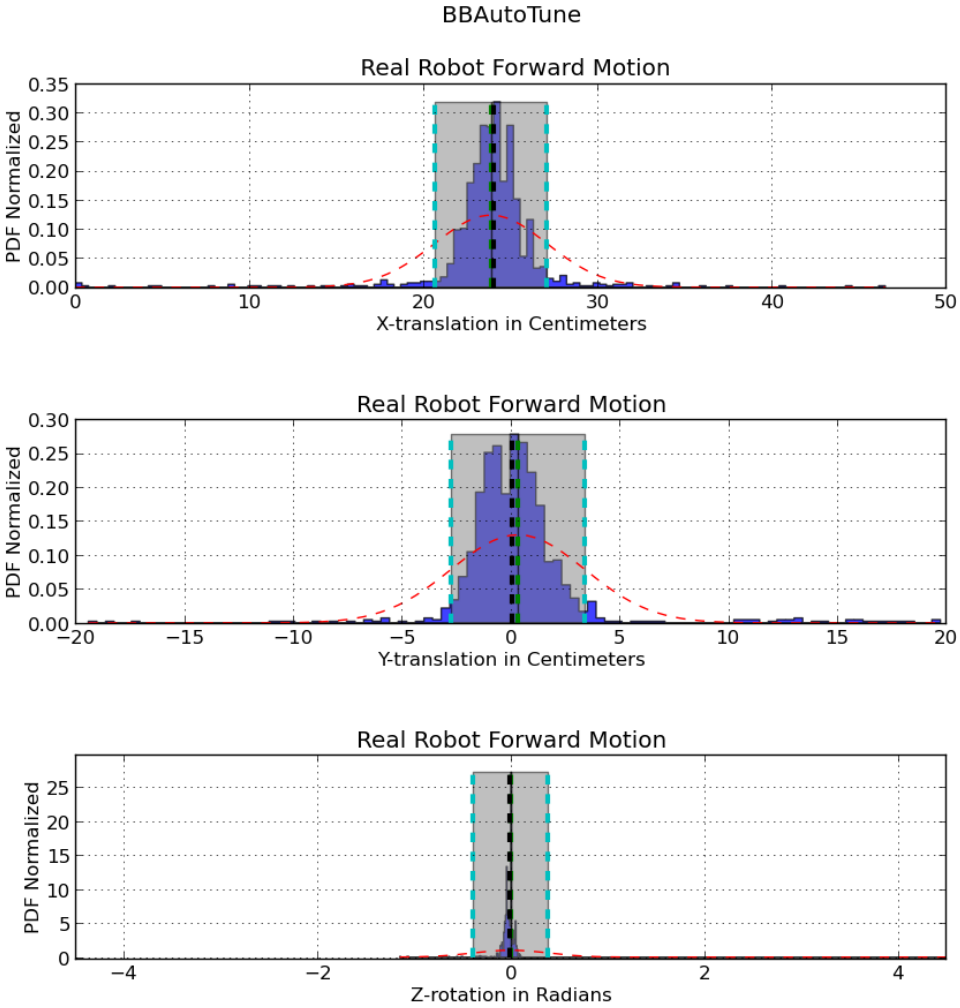

FIGURE 4.6: Here you see the distribution of x-translation, y-translation, and z-rotation dimensions recorded for the real robot forward motion. The broken black bar represents the mode, the green broken bar represents the mean, the cyan broken bars represent one standard deviation from the mean, and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

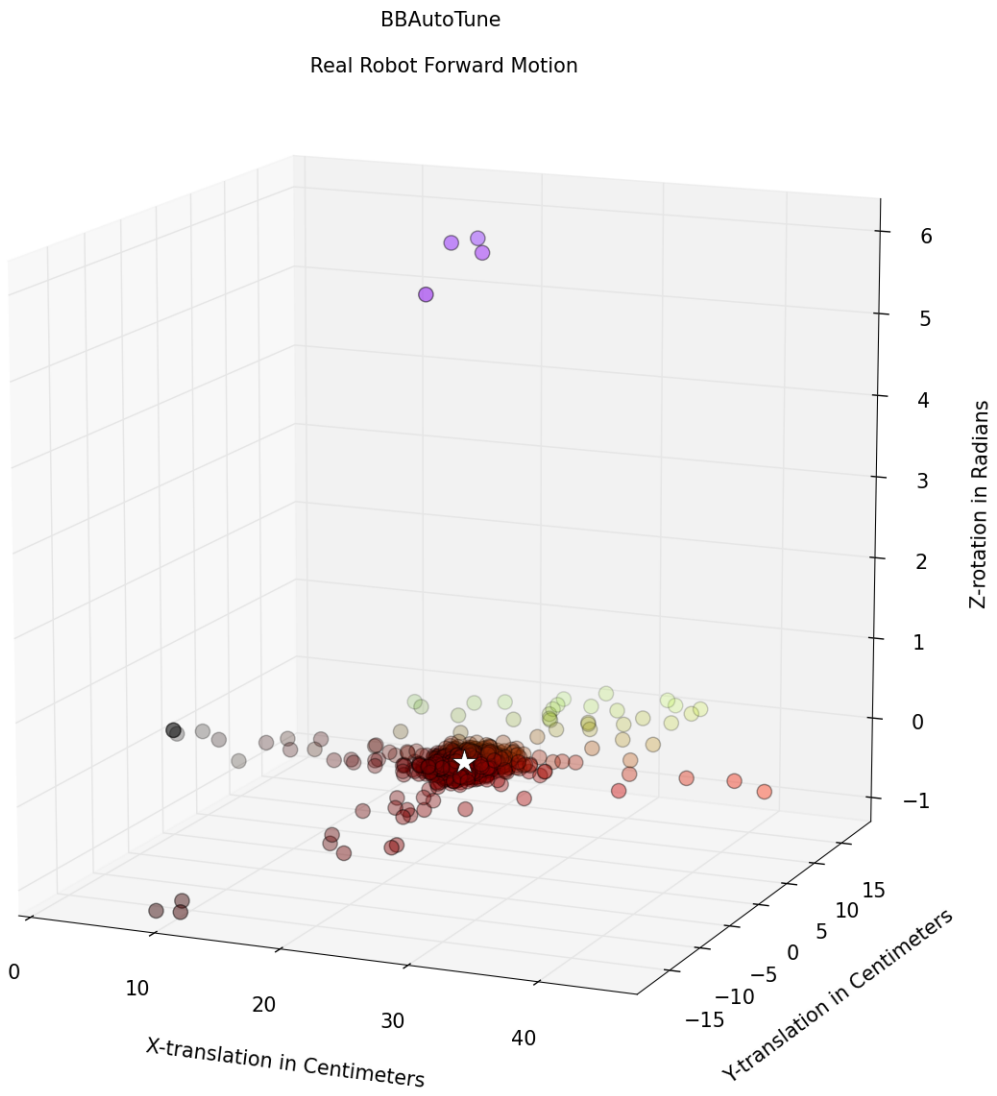

FIGURE 4.7: Here you see a 3D scatter plot of the x-translation, y-translation, and z-rotation values for the real robot after it performed the forward command. Positive values of x-translation, y-translation, and z-rotation contribute a portion of red, green, and blue respectively to each scatter plot.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

This metric would need to indicate how different or dissimilar the movement of the simulated robot was from the real robot while running the GA. The intuition was that as the simulated robot motion observed moved closer to the centroid of the real robot motion, the simulated motion would become increasingly indistinguishable from the real motion. Various distance functions were looked at and the Mahalanobis distance was chosen as the basis for the fitness function. The Mahalanobis distance is the generalized form of the Euclidean distance such that the Mahalanobis distance accounts for the correlation in the data set since it is computed using the inverse of the variance-covariance matrix [9]. For uncorrelated data, the Mahalanobis distance reduces to the Euclidean distance [10].

Inefficiencies with the overhead camera and the timing at which the position and orientation state of the real robot was captured could have potentially skewed the real robot motion model with outliers ultimately resulting in a skewed multivariate mean location and a skewed inverse variance-covariance matrix making the Mahalanobis distance skewed. To account for the potential outliers in the real motion data set, a robust mean location and a robust variance-covariance matrix was computed from the data set using the Fast-MCD algorithm implemented in the Scikit-learn Python module [11][12]. The classical mean location was 23.8644632cm, 0.338269853cm, and -0.00417473025rad for x-translation, y-translation, and z-rotation respectively while the robust mean location was 23.9934044cm, 0.0351240536cm, and -0.0189964938rad for x-translation, y-translation, and z-rotation respectively. The left matrix below was the classical variance-covariance matrix while on the right was the robust variance-covariance matrix returned by the Fast-MCD algorithm.

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

The resulting samples weighted higher than others by the Fast-MCD algorithm are shown in Figure 4.8. These samples where used to calculate the robust mean and the robust variance-covariance matrix returned by the algorithm. By substituting the robust mean and the inverse of the robust variance-covariance matrix into the Mahalanobis distance calculation, the robust distance RD for any sample point can be computed [13]. Comparisons between the Mahalanobis distance and the robust distance for each of the 1040 real robot

motion data points are shown in Figure 4.9. As the data points travel away from the centroid, the robust distance increases more rapidly than the Mahalanobis distance.

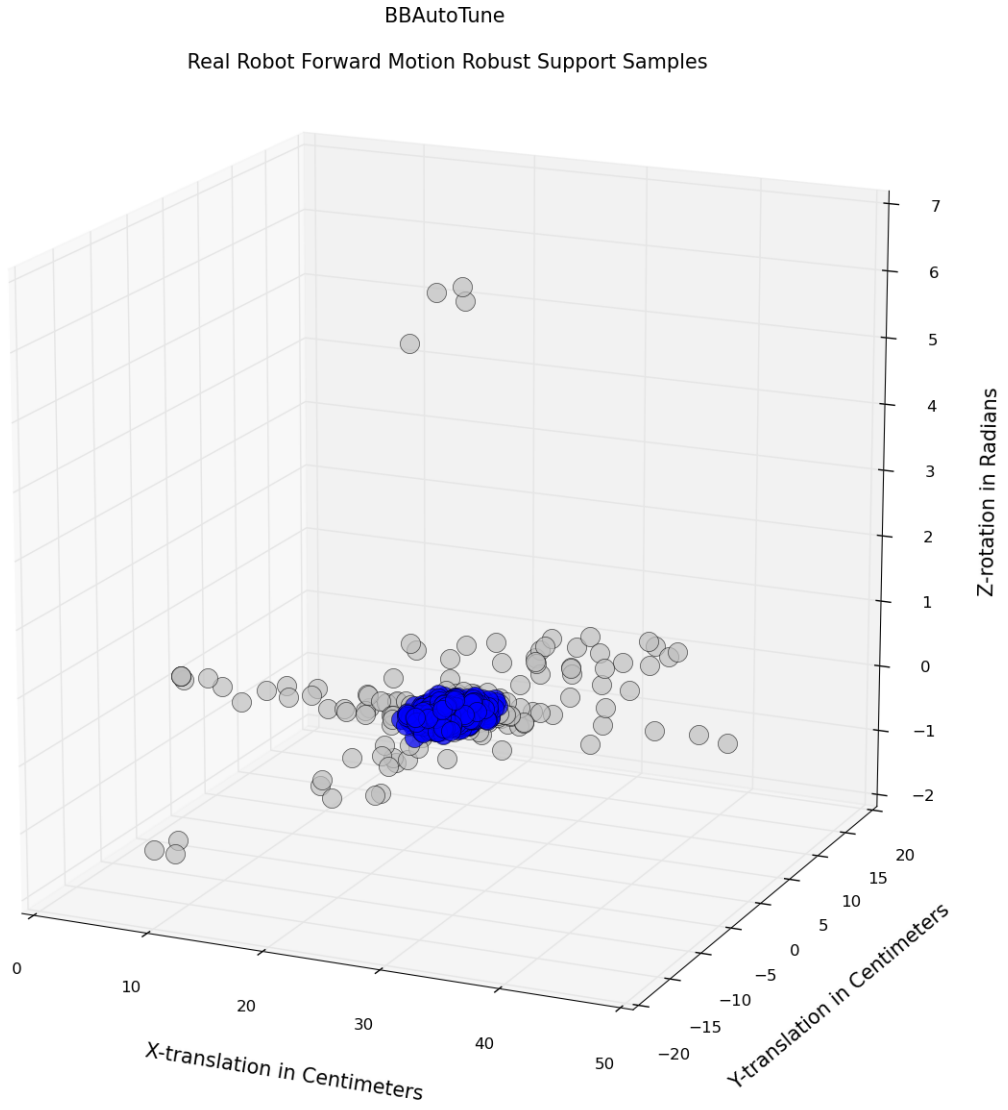

FIGURE 4.8: Here you see a 3D scatter plot of the support samples in blue used to calculate the robust mean location and the robust variance-covariance matrix as returned by the Fast-MCD algorithm.

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 (pitch, roll, and heave) to contend with. Thus, the simulated-robot base's x-rotation, y-rotation, and z-translation were constrained. Additionally, all wheels

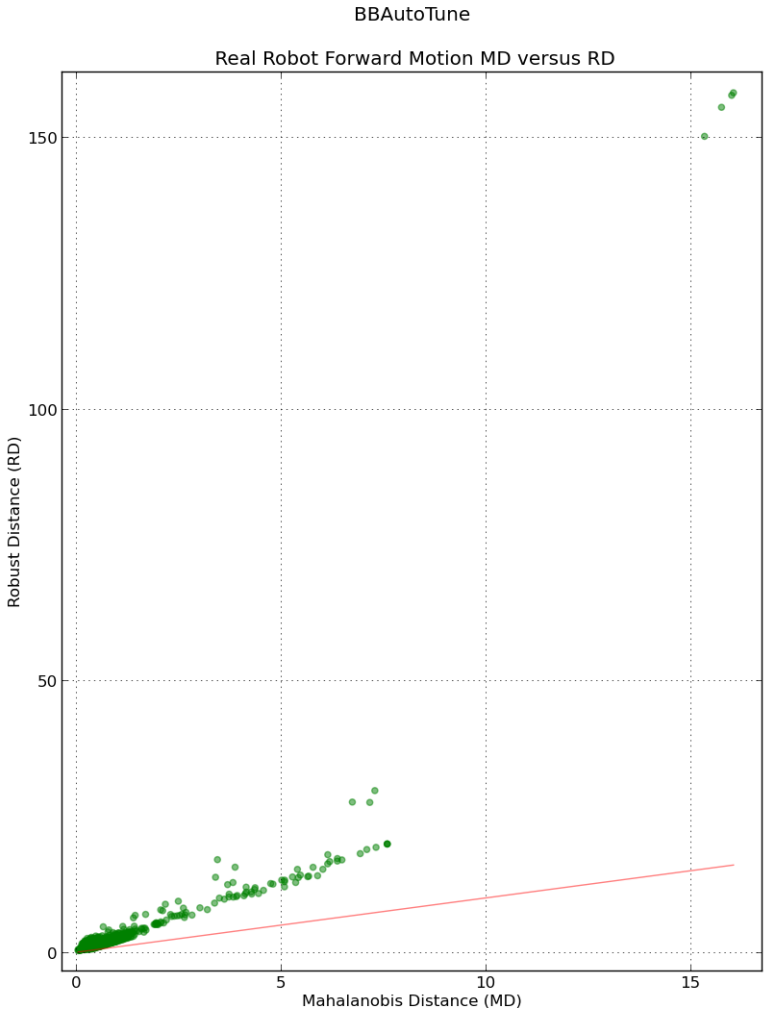

FIGURE 4.9: Here you see a scatter plot comparing the Mahalanobis distance versus the robust distance for each data point collected of the real robot motion.

had their z-translation constrained. As an added precaution, penalties where added onto the robust distance by the absolute amount the simulated robot's base violated x-rotation, y-rotation, and/or z-translation. Additionally, a time penalty was added onto the robust distance by the amount of time in seconds the simulated robot took to evaluate greater than one second with a max penalty of 15 seconds since any given evaluation period only lasted a total of 16 seconds.

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

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

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

4.2.7.4 Evaluation Setup

For each evaluation period (the running of the game engine), the 3D robot model's local coordinate system was axis aligned with the world coordinate system, the robot was faced forward looking down the positive x-axis, and its local origin was placed at the world origin. See Figure 4.10. Before each evaluation period,

the physics engine parameters were set to the values decoded from the genes of the currently being evaluated genome. All evaluation periods lasted no more than 16 seconds. If the robot stopped moving before 16 seconds, then the evaluation period ended immediately. The only applied force to the robot was the wheel torque where each wheel received the same amount of applied torque for the same duration. The duration of applied torque was roughly 16 milliseconds after which no further force was applied to the robot.

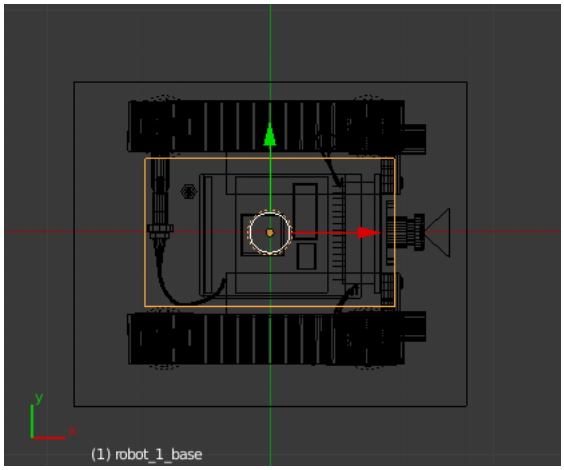

FIGURE 4.10: Here you see the simulated robot's local coordinate system axis aligned with the world coordinate system.

4.3 Platform

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

4.4 Experimental Designs

Stub.

4.4.1 Experiment one: physics engine parameter influence.

The potential Blender physics engine parameter candidates—to be tuned by the GA—were analyzed for their influence over the physics simulation. To accomplish this goal, a racquetball like environment was constructed in Blender which consisted of a ball, an enclosed arena, and an automated racket controlled via

a Python script. See Figure 4.11. Note that the ball was allowed to travel in all three dimensions and was completely physics based.

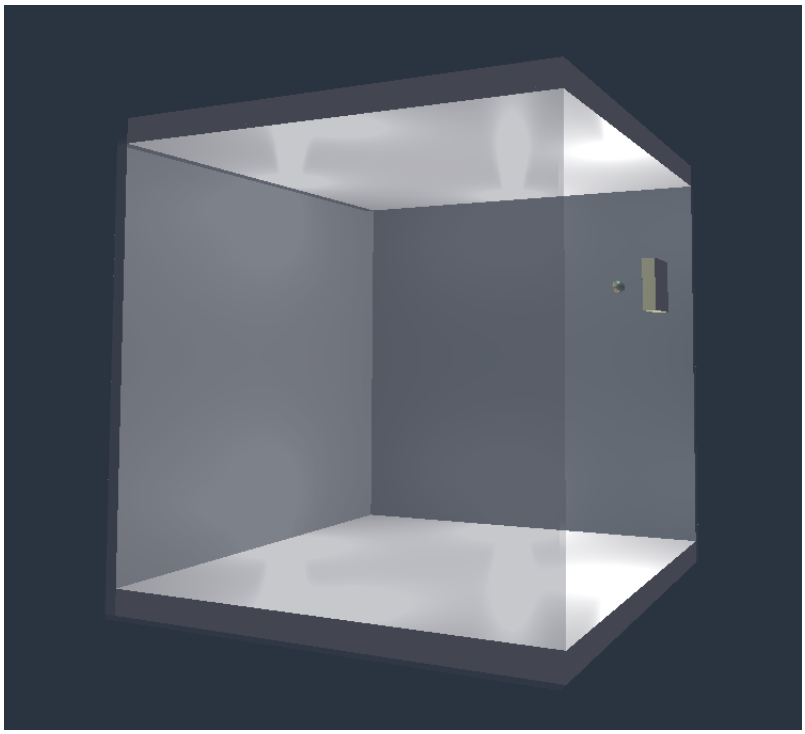

FIGURE 4.11: Here you see the racquetball environment used to test the physics engine parameter candidates' influence over the physics simulation.

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

4.4.2 Experiment two: tournament selection with self-adaptation.

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

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

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

4.5 Experimental Results

Stub.

4.5.1 Experiment one: physics engine parameter influence.

To compare the recorded tweaked-parameter ball paths with the recorded standard, four methods were utilized to give an indication as to how much influence any one candidate parameter had over the simulation. The first method was visual inspection. All 44 tweaked-parameter ball paths were plotted against the standard. See Figure 4.12 and Figure 4.13. The second method was an in-house algorithm, titled the *Lettier distance*, which gives the maximum euclidean distance between two discrete curves P and Q. Informally, imagine holding a rubber band in your hands where the left hand affixes the left end of the rubber band to

the first point in P and the right hand affixes the right end of the rubber band to the first point in Q. During each iteration, you advance the left end of the rubber band to the next point in P and you advance the right end of the rubber band to the next point in Q. If the distance grows between point $p_i \in P$ and point $q_j \in Q$, the rubber band stretches but never shrinks. If $|P| < |Q|$ then you hold the left end of the rubber band to the last point in P and continue advancing to the last point in Q and vice versa. Once you reach the last point in P and the last point in Q, the resulting length of the rubber band is the max euclidean distance between P and Q. See Figure 4.14. The third method was the Frchet distance and the fourth method was the Hausdorff distance [14] [15].

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

4.5.2 Experiment two: tournament selection with self-adaptation.

Stub.

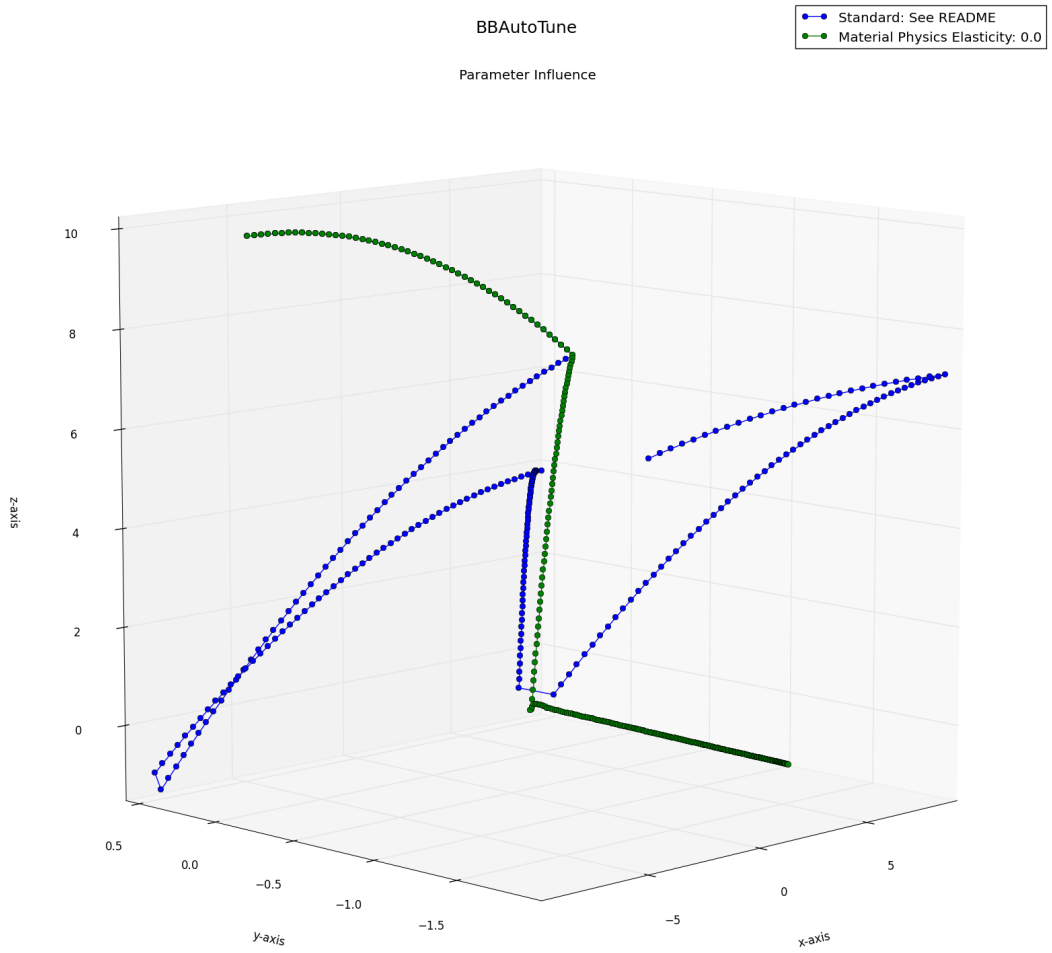

FIGURE 4.12: Here you see the dissimilarity of the ball paths between the tweaked, material-physics elasticity parameter and the standard (where no parameters were tweaked).

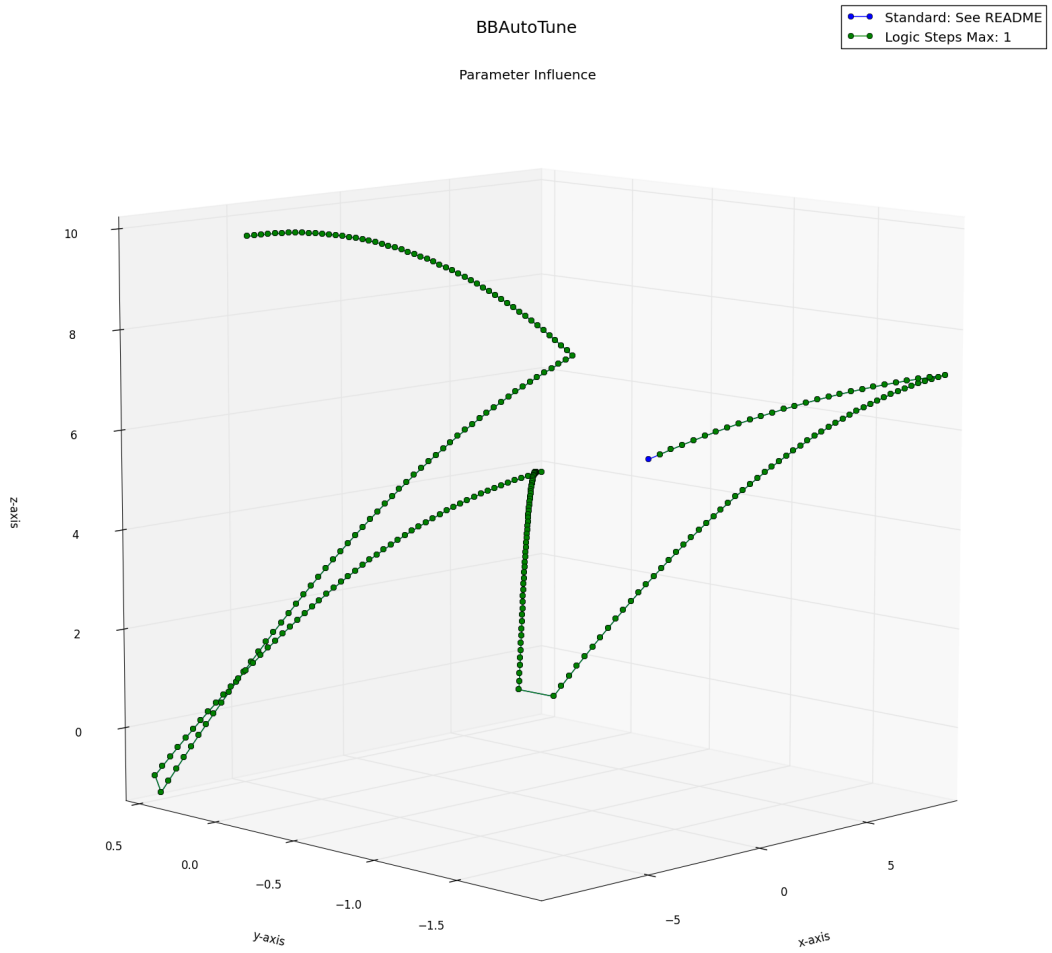

FIGURE 4.13: Here you see the similarity of the ball paths between the tweaked, logic-steps max parameter and the standard (where no parameters were tweaked).

Tweaked Parameter: Value	Lettier Distance	Frchet 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
Actor: False	0.329232644023	0.329232644023	0.329232644023
Ghost: True	138.489494312	138.489494312	132.02387242
Use Material Force Field: True	0.329232644023	0.329232644023	0.329232644023
Rotate From Normal: True	0.329232644023	0.329232644023	0.329232644023
No Sleeping: True	0.329232644023	0.329232644023	0.329232644023
Attributes Radius: 1cm	0.329232644023	0.329232644023	0.329232644023
Attributes Mass: 10000.0	3.57800913743	3.30042073543	3.21534852885
Attributes Form Factor: 0.0	3.55513601614	18.1217630973	3.275450812
Velocity Minimum: 1.0	0.329232644023	0.329232644023	0.329232644023
Anisotropic Friction: True	0.329232644023	0.329232644023	0.329232644023
Velocity Maximum: 1.0	17.1034960114	17.130808295	16.8922077042
Damping Translation: 1.0	17.9729741832	17.9729741832	17.9729741832
Damping Rotation: 1.0	8.09457671285	5.47840838373	5.18331655137
Collision Bounds: False	8.81773036062	17.4044978663	2.80531111777
Collision Bounds Margin: 0m	18.740072391	9.79629223411	3.78119352088
Collision Bounds: False	4.28865271192	4.23797419961	3.72440427516
Launch Dynamic Object Settings Force X: 30.0	5.21169989737	5.21169989737	3.78308993247
Launch Dynamic Object Settings Torque X: 30.0	8.82943537929	8.70403741077	6.58255281735
Launch Dynamic Object Settings AngV X: 30.0	2.93219084728	2.63246279507	1.82341393368
Launch Dynamic Object Settings LinV X: 0.0	19.1625050915	19.1625050915	17.0670582483
Launch Damping Frames: -32768	0.329232644023	0.329232644023	0.329232644023
Collision Dynamic Object Settings Force X: 30.0	5.32618850819	18.9696056987	3.80913153981
Collision Dynamic Object Settings Torque X: 30.0	1.67496526126	1.52842619107	1.52842619107
Collision Dynamic Object Settings LinV X: 0.0	19.2801183238	19.1535557726	3.3918725084
Collision Dynamic Object Settings AngV X: 30.0	4.69097055763	18.7403224372	3.80913153981
Collision Damping Frames: -32768	0.329232644023	0.329232644023	0.329232644023

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

```
BEGIN
    P = ⟨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 4.14: Here you see the Lettier distance algorithm.

Chapter 5

BlenderSim

5.1 Overview

TODO: Add something about comparing the physics based motion to the real robot.

Over the course of three months, a Blender based simulation engine 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. This gave way to the thesis of tuning the physics engine via a genetic algorithm. Once the GA learned the motion of the real robot, BlenderSim was restored to being wholly physics based with the physics engine modeling the locomotion of the real robots.

5.2 HRTeam

Stub.

5.3 Problems Faced

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

Scale was problematic as the Blender/Bullet physics engine has difficulty with collisions of objects that have a size outside of the assumed range of .05 to 10 meters [16]. Objects smaller than .05 (5cm) Blender/Bullet units, in any given dimension, erratically jitter despite having no force acting upon them. As such, since the wheel dimensions of the real SRV-1 are 2.11cm x 2.45cm x 2.52cm, the to-scale 3D model

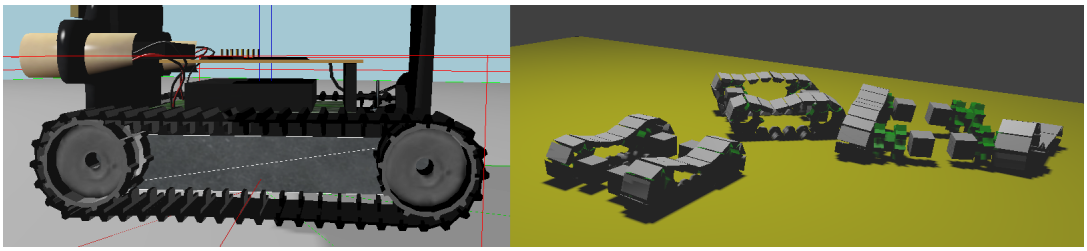

FIGURE 5.1: Here you see the to-scale treads modeled after the physical SRV-1 robot on the left and the physics-based rigid-body tracks in motion on the right.

of the SRV-1 was affected by this scale limitation of the physics engine.

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

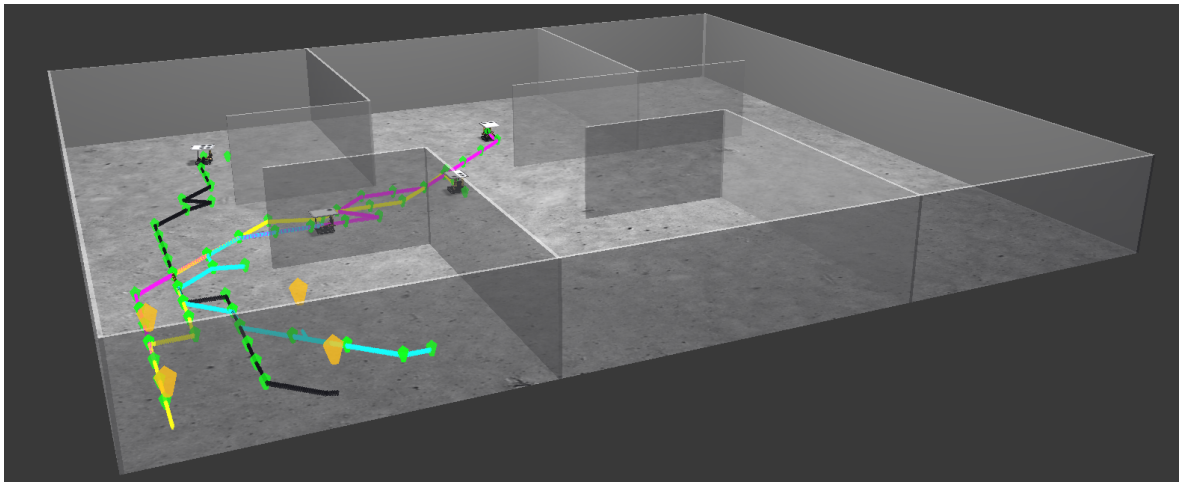

FIGURE 5.2: Here you see the constant velocity motion model at work in BlenderSim with four robots traversing their respective paths of which consist of way-points scrapped from a previously recorded physical lab experiment.

5.4 Implementation

Stub.

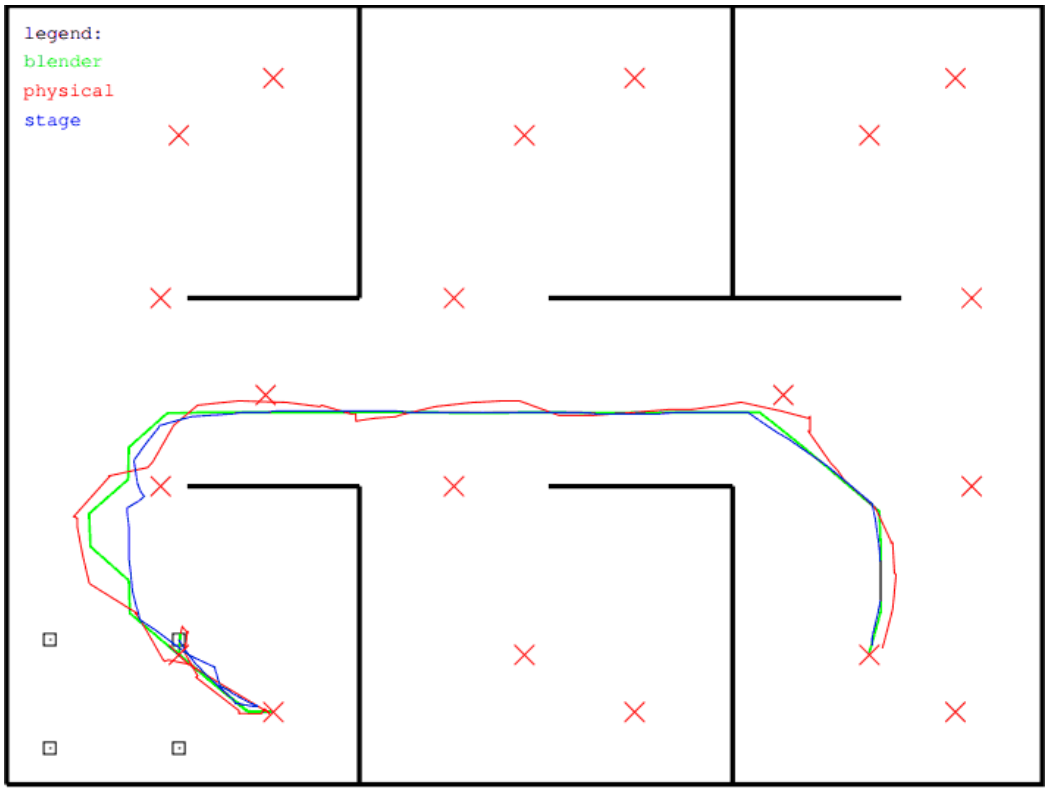

FIGURE 5.3: Here you see the path plots–as plotted by Dr. Elizabeth Sklar–of the BlenderSim robot, the physical robot, and the *Stage* (a 2D robot simulator already in use by HRTeam)[17] robot plotted in comparison.

5.4.1 Arena

The arena is a $602cm \times 538cm$ enclosure with with a main hallway and six compartments. All walls are physics based and respond accordingly should any robot try to pass through them. See Figure 5.4.

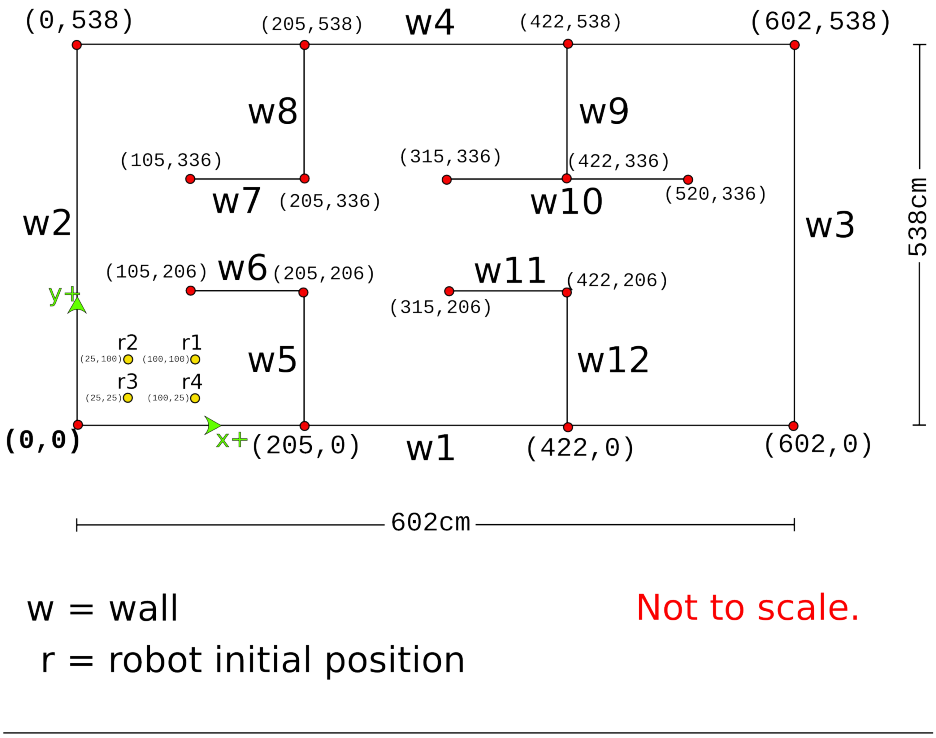

FIGURE 5.4: Here you see the arena map (not to scale) with wall coordinates and robot starting locations.

5.4.2 Surveyor SRV-1 Blackfin 3D Models

The 3D model of the Surveyor SRV-1 Blackfin 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 the rests above the body of the robot. The base and the wheels are the only physics based objects on the model with the wheels being connected to the base via a rigid body hinge joint. See Figure 5.5.

5.4.3 Robot Servers

TODO: Change this to reflect when BlenderSim communicates directly with HRTeam.

The servers are expressed in the simulation as the amber colored *gems* located just above the 3D modeled SRV-1's. See Figure 5.6. The server logic utilizes the *ServerSocket* Python module. Each of the four

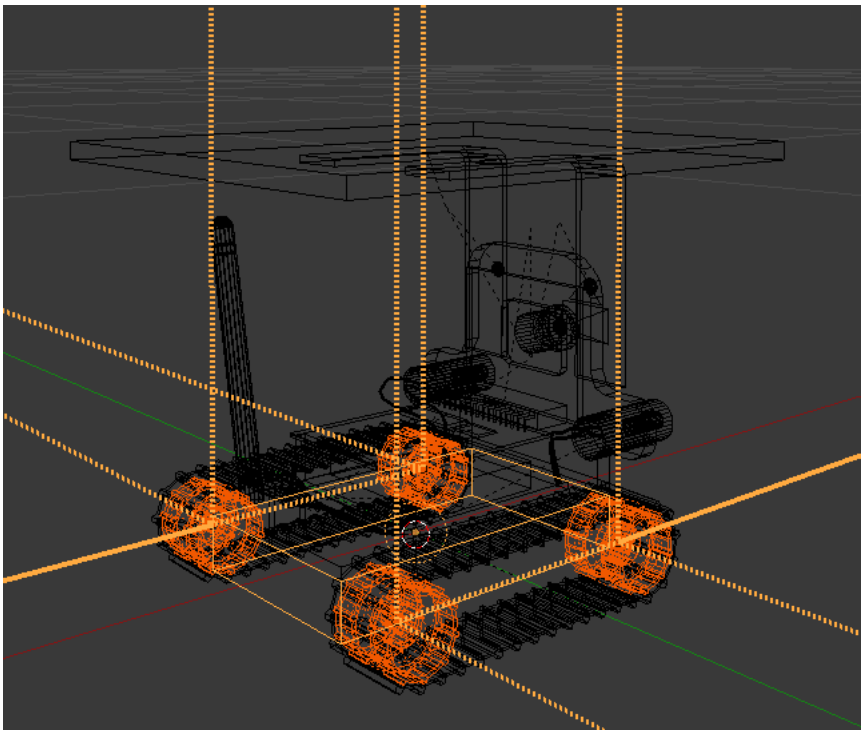

FIGURE 5.5: Here you see the 3D model of the Surveyor SRV-1 Blackfin with the wheels connected to the base via rigid body hinge joints.

servers run in separate threads parented to the Blender process. The four server listen on ports 5001, 5002, 5003 and 5004 respectively. As a connection is made to a client, the server performs a short handshake and begins requesting waypoints from the client. The waypoints are put into a thread safe queue for the robot controller to read from at its leisure. Once the client notifies the server that there are no more waypoints, the server puts *done* in the waypoint queue, sends *done* to the client, shuts down the port, and its thread is terminated.

If the Blender game engine is terminated before the servers receive all of their waypoints, the servers shut down their ports and their threads are terminated. This allows the simulator to be started again fresh (i.e. [*Errno 98*] *Address already in use.* errors) without having to close Blender itself in order to run another simulation.

5.4.4 Robot Clients

TODO: Change this to reflect when BlenderSim communicates directly with HRTeam.

The four robot clients connect to ports 5001 through 5004 respectively. Utilizing a custom log reader script, the clients read in their respective waypoints from the waypoint logs. Once connected to their respective servers, the clients perform a short handshake and then begin transmitting waypoints as each of their respective servers request them. Once the clients run out of waypoints to transmit, they signal the servers that there are *nomore* waypoints. Once they receive *done* from their respective server, they close the connection and terminate. Out of convenience, a master script runs the four robot clients in four separate asynchronous sub-processes utilizing the *Subproccess* Python module.

5.4.5 Robot Controllers

TODO: Change this to reflect when BlenderSim communicates directly with HRTeam.

The robot controllers are expressed in the simulation as the metal box bases of the SRV-1 3D models. See Figure 5.6. The four robot models are controlled via the logic contained in four robot controllers. No robot will move while their respective waypoint queues are empty. Once their waypoint queues begin being populated, by their respective servers, the robots will traverse a linear path to each waypoint first by turning to face the waypoint at a constant velocity and then by moving forward towards the waypoint at a constant

linear velocity. The velocities were calculated from a historical physical lab experiment log. Once the robots detect *done* in their waypoint queue, they will cease movement indicating that they have traversed the complete $A*$ previously-calculated path as read from the historical physical lab experiment master log.

5.4.6 Task Points Manager

TODO: Change this to reflect when BlenderSim communicates directly with HRTeam.

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

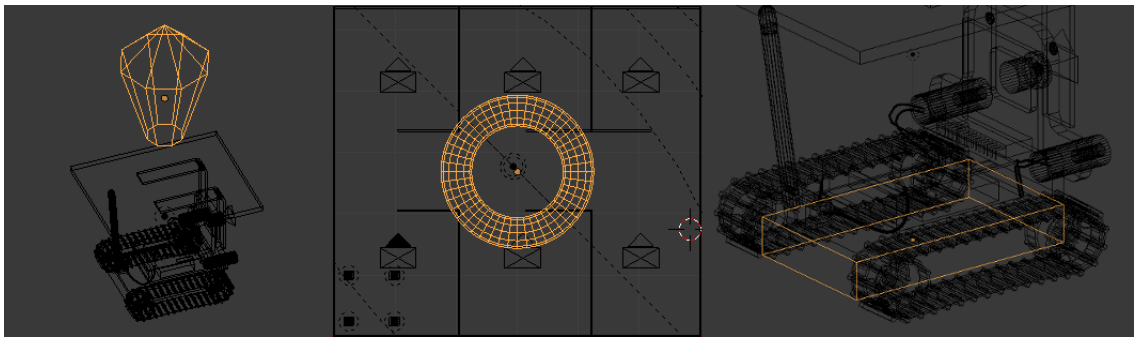

FIGURE 5.6: Here you see from left to right, the server *gem*, the task points manager, and the robot controller manifestations in the simulation.

5.5 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.

5.6 Experimental Designs

Stub.

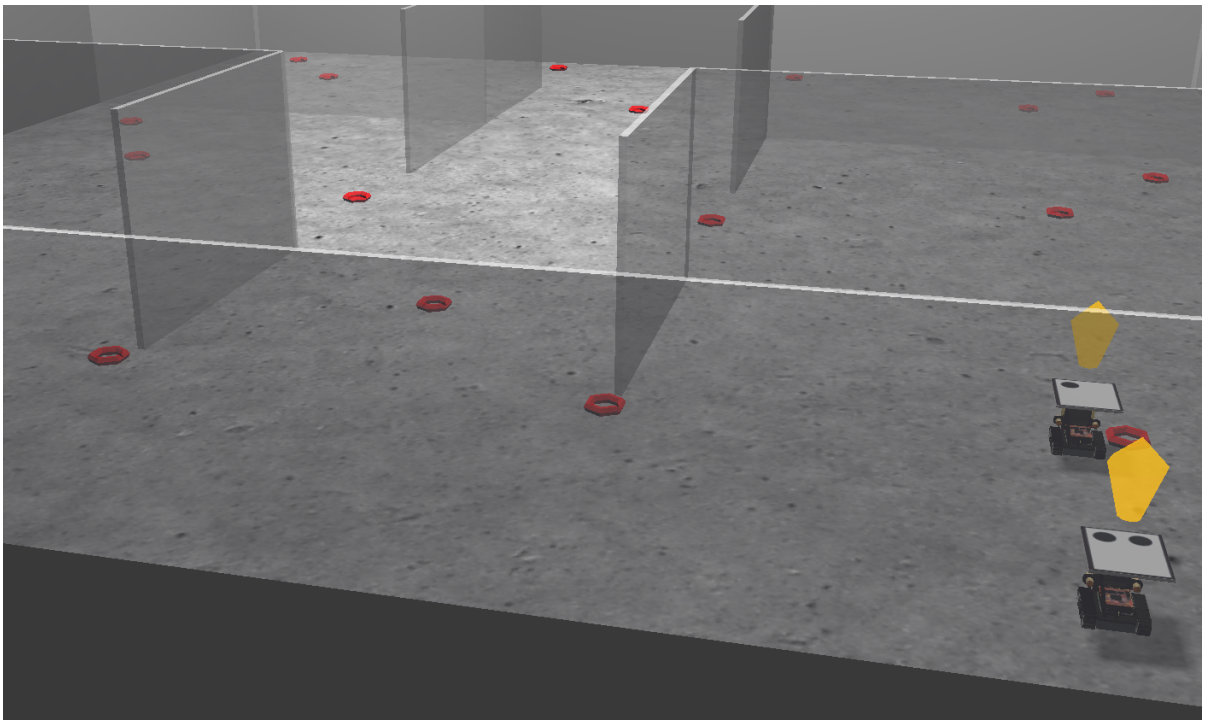

FIGURE 5.7: Here you see the red torus task points laid out in the arena.

5.7 Experimental Results

Stub.

Chapter 6

Conclusion

TODO: Write this last.

6.1 Blender and Bullet

Stub.

6.2 Reality Gap

Stub.

6.3 SimPL

Stub.

6.4 BBAutoTune

Stub.

6.5 BlenderSim

Stub.

Appendix A

Software Code

A.1 SimPL

Stub.

A.2 BBAutoTune

Stub.

A.3 BlenderSim

Stub.

Bibliography

[1] Jhih Ren. Juang, Wei Han Hung, and Shih Chung Kang. Using game engines for physicalbased simulationsa forklift. *Journal of Information Technology in Construction*, 16:3–22, 2011.

[web] Doc:2.6/manual/game engine/physics/object type/dynamic - blenderwiki. `http://wiki.blender.org/index.php/Doc:2.6/Manual/Game_Engine/Physics/Object_Type/Dynamic`, . Accessed August 13, 2013.

[web] Doc:2.6/manual/game engine/physics/object type - blenderwiki. `http://wiki.blender.org/index.php/Doc:2.6/Manual/Game_Engine/Physics/Object_Type`, . Accessed August 13, 2013.

[2] Roland Hess. *The Essential Blender: Guide to 3D Creation with the Open Source Suite Blender*. No Starch Press, San Francisco, CA, USA, 2007. ISBN 1593271662, 9781593271664.

[3] 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`.

[4] 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`.

[5] 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.

[6] Marek Obitko. Selection: Introduction to genetic algorithms. `http://www.obitko.com/tutorials/genetic-algorithms/selection.php`, 1998. URL `http://www.obitko.com/tutorials/genetic-algorithms/selection.php`.

[7] Heinz Mühlenbein and Dirk Schlierkamp-Voosen. Predictive models for the breeder genetic algorithm i. continuous parameter optimization. *Evol. Comput.*, 1(1):25–49, March 1993. ISSN 1063-6560. doi: 10.1162/evco.1993.1.1.25. URL `http://dx.doi.org/10.1162/evco.1993.1.1.25`.

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

[8] 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.

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

[10] 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.

[11] 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`.

[12] 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.

[13] 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`.

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

[15] 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.

[16] 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.

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