SimPL
Simple Pong Learner

David Lettier

November 20, 2013

Note: this write up stub is to be included in the master's thesis proposal, CISC 7902X project proposal, and the CISC 7900X semester write up.

1 Introduction

Simple **P**ong **L**earner or SimPL is an asymmetric autonomous pong clone with one paddle and one ball. SimPL is web based 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.

2 Implementation

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 1.

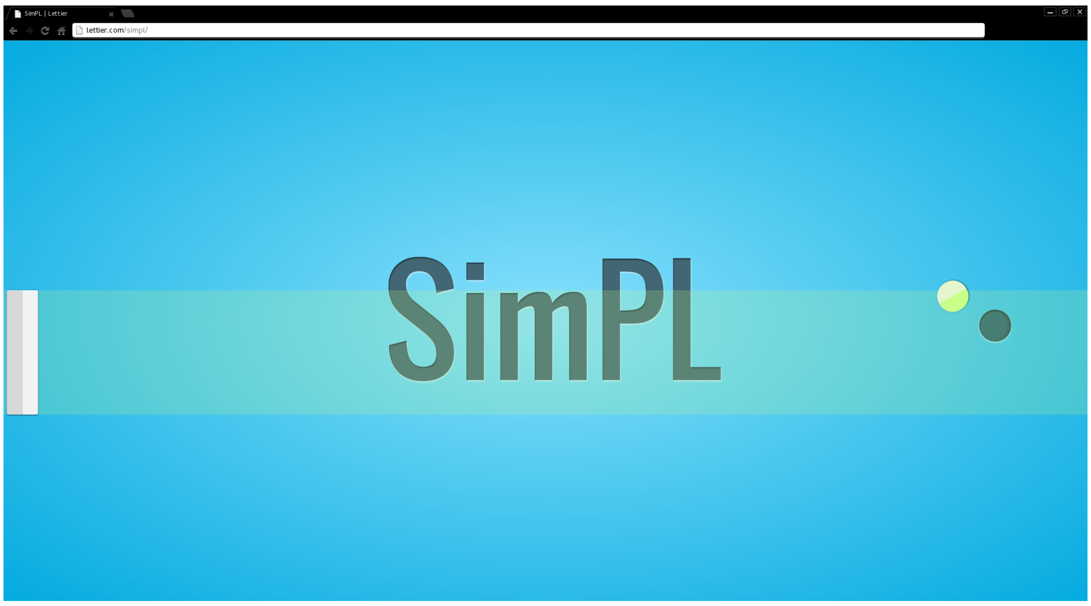


Figure 1: Here you see the SimPL arena containing the paddle and ball.

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 2. The ball's position is managed by the physics engine which responds to any collisions against the arena walls and/or the paddle.

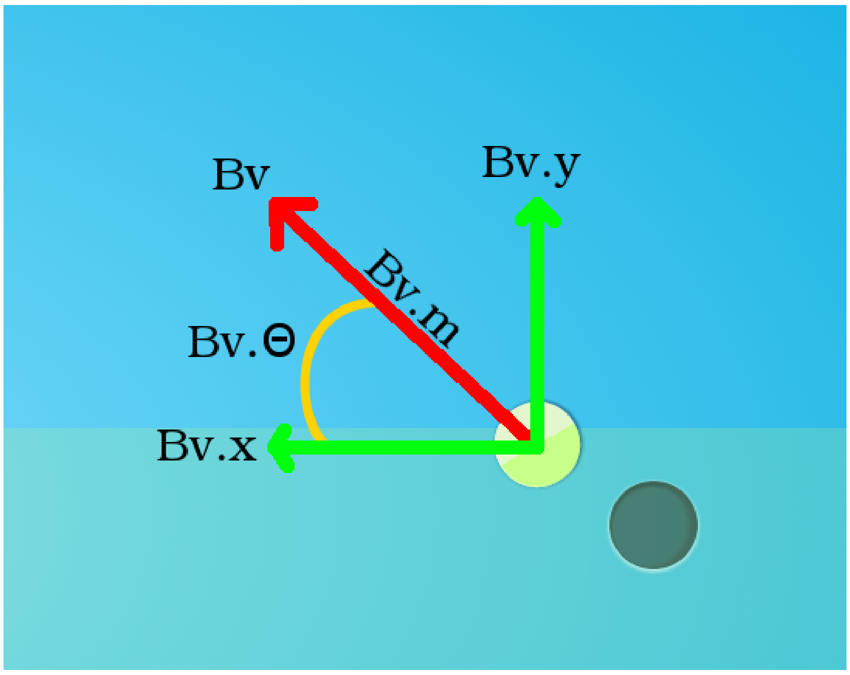


Figure 2: Here you see the ball's dynamic physics properties.

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.

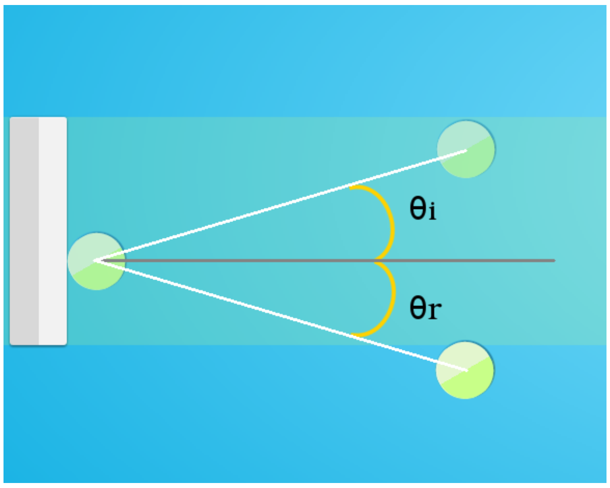


Figure 3: Here you see the ball's collision angle-of-incidence θ_i and its angle-of-reflection θ_r.

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

[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.

2.3 Paddle

The paddle is a physics based dynamic object that has a fixed velocity angle of either 90° or 270°. See Figure 4. Its starting position as well as its starting velocity magnitude are the same at the start of every round.

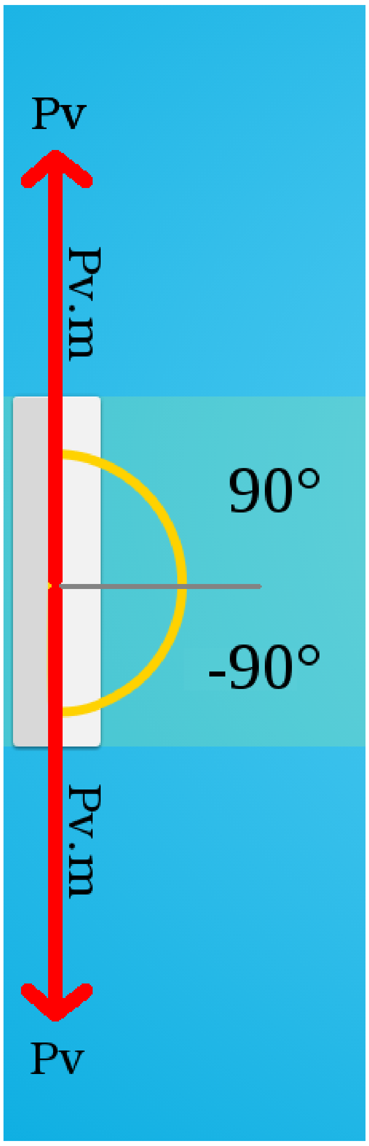


Figure 4: Here you see the paddle's dynamic physics properties.

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 classifies 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.

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 simulation, 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.

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. 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 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 $[-1,1]$.

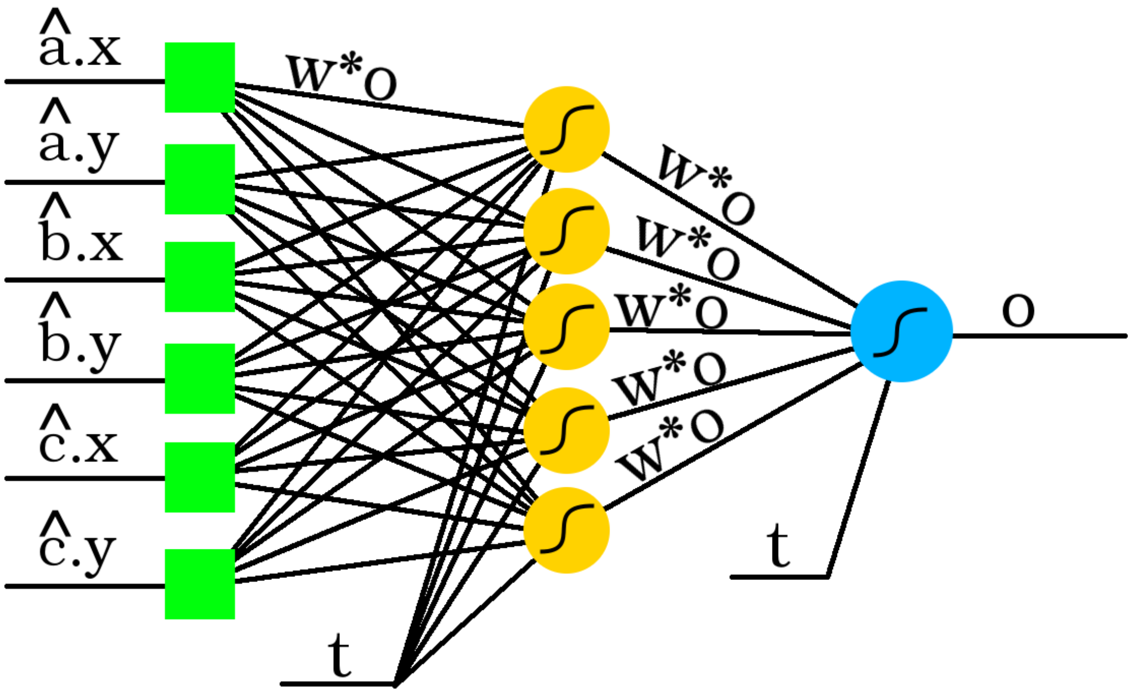


Figure 5: Here you see the neural network as constructed in SimPL.

Input to the neural network is normalized generating three unit vectors: from the balls's center to the paddle's center, the ball's velocity, and from the window's origin to the paddle's center. See Figure 6. These three unit vectors are broken down into their components resulting in six inputs to the neural network. See Figure 7.

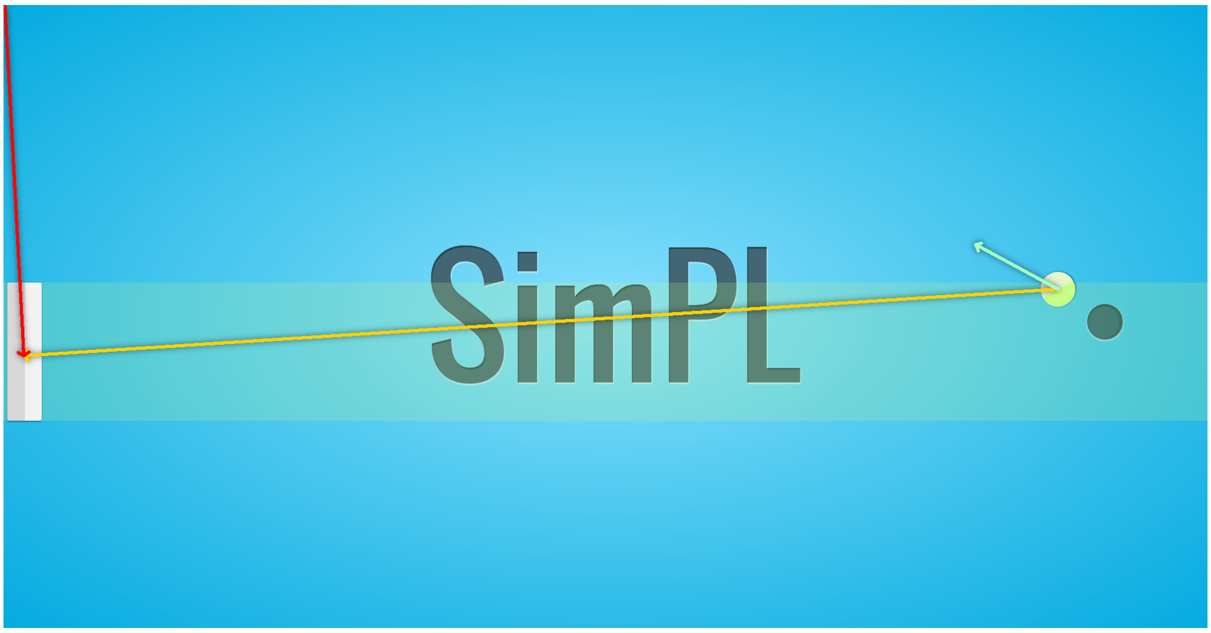
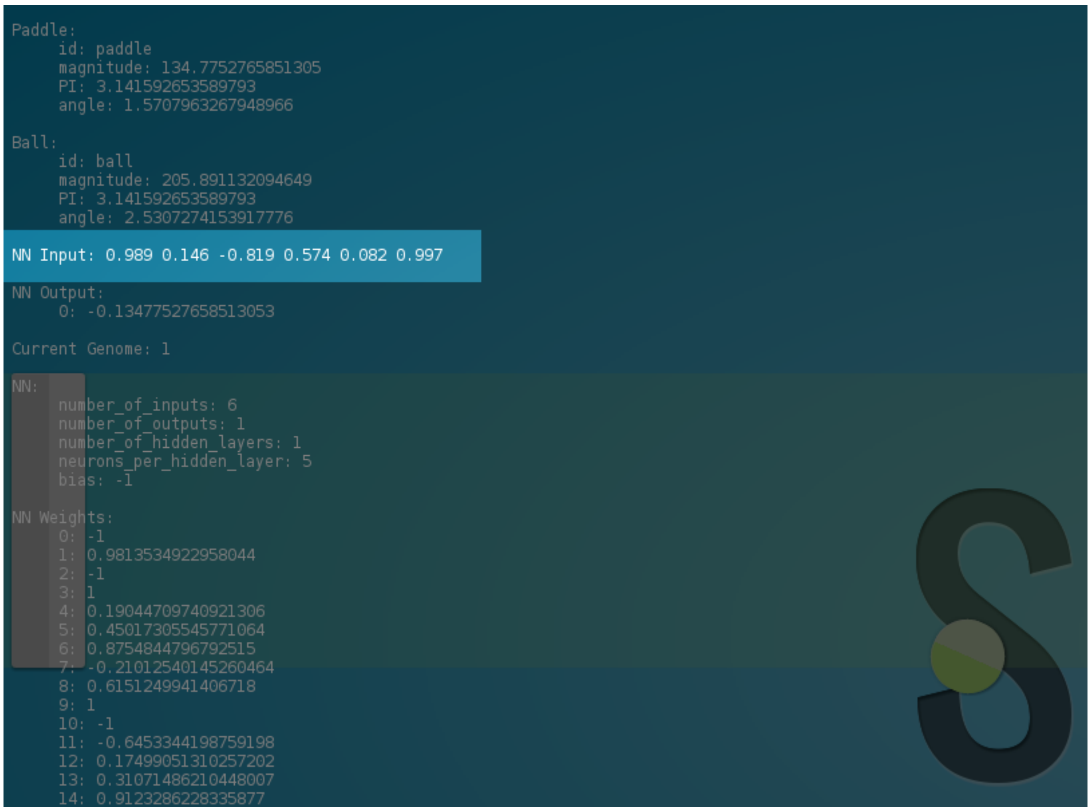


Figure 6: Here you see the three input vectors to the neural network. Note that these vectors are normalized thereby turning them into unit vectors.



```
Paddle:
      id: paddle
      magnitude: 134.7752765851305
      PI: 3.141592653589793
      angle: 1.5707963267948966

Ball:
      id: ball
      magnitude: 205.891132094649
      PI: 3.141592653589793
      angle: 2.5307274153917776

NN Input: 0.989 0.146 -0.819 0.574 0.082 0.997

NN Output:
      0: -0.13477527658513053

Current Genome: 1

NN:
      number_of_inputs: 6
      number_of_outputs: 1
      number_of_hidden_layers: 1
      neurons_per_hidden_layer: 5
      bias: -1

NN Weights:
      0: -1
      1: 0.9813534922958044
      2: -1
      3: 1
      4: 0.19044709740921306
      5: 0.4501730554577106
      6: 0.875484479679251
      7: -0.21012540145260464
      8: 0.6151249941406718
      9: 1
      10: -1
      11: -0.6453344198759198
      12: 0.17499051310257202
      13: 0.31071486210448007
      14: 0.9123286228335877
```

Figure 7: Here you see the normalized input to the neural network.

2.6 Genetic Algorithm

Instead of using back-propagation to train the weights of the neural network, a genetic algorithm is used to optimize or rather tune the weights of the neural network. 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 fitness evaluated by a fitness function. As the genomes evolve over generations to produce fitter genomes, the neural network becomes increasingly accurate at classifying what the paddle should do (move up, stay still, or move down) based on the state of the ball and the paddle.

The genetic algorithm contains four operators that work to produce fitter generations during the creation of a new population. The operators include: the elitism operator, the selection operator, the crossover operator, and the mutation operator. Initially, the genetic algorithm creates a random population of genomes. These initial genomes have zero fitness and a fixed number of genes. Each gene in every genome is given a random value sampled from a uniform distribution coinciding with some valid range. The genes are the input parameters to whatever mechanism the genetic algorithm is working to optimize. In the case of SimPL, the mechanism is the neural network and the genes or parameters—of each genome—represent the 41 weights of the neural network. Each gene/parameter/weight 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/generation from the now old population/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 8.

```
Begin GA :
        Generate population $P_0$.
        While not terminate :
            Evalute population $P_i$.
            Create empty populutation $P_{i+1}$.
            Sort $P_i$ in non−decreasing order of fitness.
            Select n fittest from $P_i$ and append to $P_{i+1}$.
            While $|P_{i+1}| <$ population size :
                If perform crossover and mutation in sequence :
                    $g_1, g_2 \longleftarrow$ Select two genomes from $P_i$ via roulette selection.
                    /∗ Generate two offspring via crossover. ∗/
                    $o_1, o_2 \longleftarrow$ Crossover$(g_1, g_2)$.
                    /∗ Mutate crossover's offspring and
                    append to new population. ∗/
                    $P_{i+1} \longleftarrow$ Mutate$(o_1, o_2)$.
                Else :
                    $g_1, g_2 \longleftarrow$ Select two genomes from $P_i$ via roulette selection.
                    /∗ Generate two offspring via crossover
                    and append to new population. ∗/
                    $P_{i+1} \longleftarrow$ Crossover$(g_1, g_2)$.
                    /∗ Mutate the two selected genomes
                    and append to new population. ∗/
                    $P_{i+1} \longleftarrow$ Mutate$(g_1, g_2)$.
            $P_i \longleftarrow P_{i+1}$
```

Figure 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. 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 where 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. Of course if the crossover and mutation probabilities are both set to 1 then yes they always occur. Before the genetic algorithm is initialized, the crossover and mutation probabilities are set before hand to some value. 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 genomes/solutions to. For SimPL, the parameter space is the weights of the neural network. Each genome represents a solution or point in the weights/parameter space. An optimal solution in the weights space will make the neural network always give a correct classification as to what movement the paddle must make based on the ball's and the paddle's current states. This of course 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.

2.7 Database Manager

The 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 Platform

The screen size or rather 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 on the same platform.

3.1 Laptop

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.

4 Experimental Designs

Each SimPL experimental design (with the exception of experiment 6) was constructed in such a way as to focus on a set of one or more facets to the genetic algorithm. See Table 1. With each progressive experiment, the facets not focused on were kept the same from the previous experiment. The goal of the experiments were 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.

Experiment one and two revolved around the fitness function, the selection operator, and the static crossover and static mutation properties of the genetic algorithm. Experiment 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. Finally, experiment six revolved around constructing a paddle as an optimal-performance standard by which any other paddle could be measured by.

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 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.

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 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 center's 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.

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

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.

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 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 through out any round.

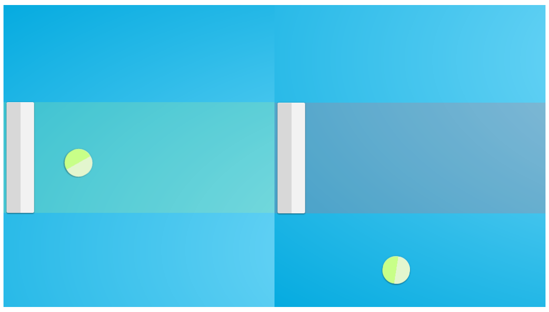


Figure 9: Here you see the simplified fitness function graphically where the paddle on the left is gaining fitness (as indicated by the green-hued bar) while the paddle on the right is gaining no fitness (as indicated by the purple-hued bar).

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. 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 (possibly due more to chance than skill) 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. 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 10. Now all genomes have a better chance of being selected to under go crossover and/or mutation thereby keeping the population diversity high and thus keeping the search scope of the fitness landscape large resulting in better performance of the genetic algorithm.

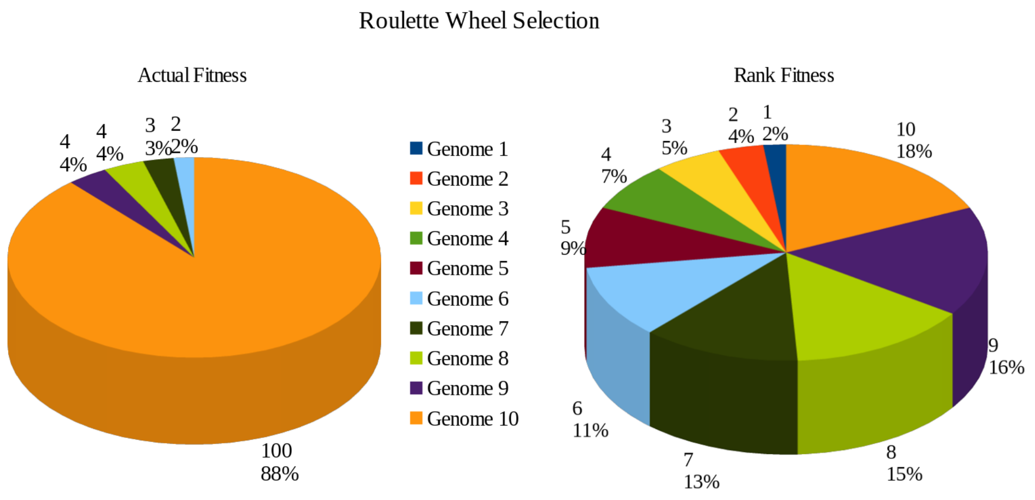


Figure 10: Here you see roulette selection using actual fitness versus rank fitness for a population of 10 genomes. On the left, the integers are the genomes' actual fitness and the percentages are their probabilities of being selected. On the right, the integers are the genomes' rank fitness and the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

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 [1]. The reasoning behind $\frac{1}{n}$ is that per mutation on a gene-by-gene basis, only one gene is mutated on average. In addition to changing the mutation probability, the mutation step was changed from adding and/or subtracting a percentage of a maximum perturbation parameter value to and/or from the gene value to sampling a value from a Gaussian distribution where the distribution $N(\mu, \sigma^2)$ is defined by the mean μ being the gene's current value before mutation and the standard deviation σ being one fourth the valid range of the gene/parameter $\left(\frac{1-(-1)}{4} = 0.5\right)$. Here the standard deviation σ is one fourth the range and thus most of the sampled values will be within two standard deviations of the mean μ 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.

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

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 4.

As outlined in [2], 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 come 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 adjusted by the adjustment parameter. Let the adjustment parameter be denoted as θ. Here θ was self-adjusted as well as outlined in [2]. 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 11. Note that to insure a level playing field, both the crossover and mutation probabilities were set to an initial value of 0.5 before the start of the genetic algorithm.

```
$Begin\ Self-adaptation:$
    $Population\ P_i\ has\ been\ evaluated.$
    $cCount = mCount = 0.$
    $CP_{sum} = MP_{sum} = 0.$
    $\overline{CP} = \overline{MP} = 0.$
    $For\ j = 1\ to\ population\ size:$
        $If\ genome_j\ created\ by\ crossover:$
            $CP_{sum} \longleftarrow CP_{sum} + (genome_j.fitness - genome_j.parentFitness).$
            $cCount \longleftarrow cCount + 1.$
        $Else\ if\ genome_j\ created\ by\ mutation:$
            $MP_{sum} \longleftarrow MP_{sum} + (genome_j.fitness - genome_j.parentFitness).$
            $mCount \longleftarrow mCount + 1.$
    $\overline{CP} = \frac{CP_{sum}}{cCount}.$
    $\overline{MP} = \frac{MP_{sum}}{mCount}.$
    $If\ P_i\ Best\ Fitness > P_i\ Worst\ Fitness:$
        $\theta \longleftarrow 0.01 * \frac{P_i\ Best\ Fitness - P_i\ Mean\ Fitness}{P_i\ Best\ Fitness - P_i\ Worst\ Fitness}.$
    $Else\ if\ P_i\ Best\ Fitness = P_i\ Mean\ Fitness:$
        $\theta \longleftarrow 0.01\ .$
    $If\ \overline{CP} > \overline{MP}:$
        $Crossover\ Prob. \longleftarrow Crossover\ Prob. + \theta.$
        $Mutation\ Prob. \longleftarrow Mutation\ Prob. - \theta.$
    $Else\ if\ \overline{CP} < \overline{MP}:$
        $Crossover\ Prob. \longleftarrow Crossover\ Prob. - \theta.$
        $Mutation\ Prob. \longleftarrow Mutation\ Prob. + \theta.$
    $Clamp\ Crossover\ Prob.\ to\ [0.001, 1.0].$
    $Clamp\ Mutation\ Prob.\ to\ [0.001, 1.0].$
```

Figure 11: Here you see the self-adaptation algorithm.

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

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

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, through out the experiment, to the crossover and mutation probabilities arrived at after the 100 generational run of the genetic algorithm in experiment three. The genetic algorithm parameters used are listed in Table 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.

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

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/solutions created by the crossover operator. The genetic algorithm parameters used are listed in Table 6.

Experiment five as well as experiment four 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.

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

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 keep the genetic algorithm operable. The genetic algorithm parameters used are listed in Table 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 or rather its center y-coordinated 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.

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

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 keep the genetic algorithm operable. The genetic algorithm parameters used are listed in Table 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.

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

5 Experimental Results

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

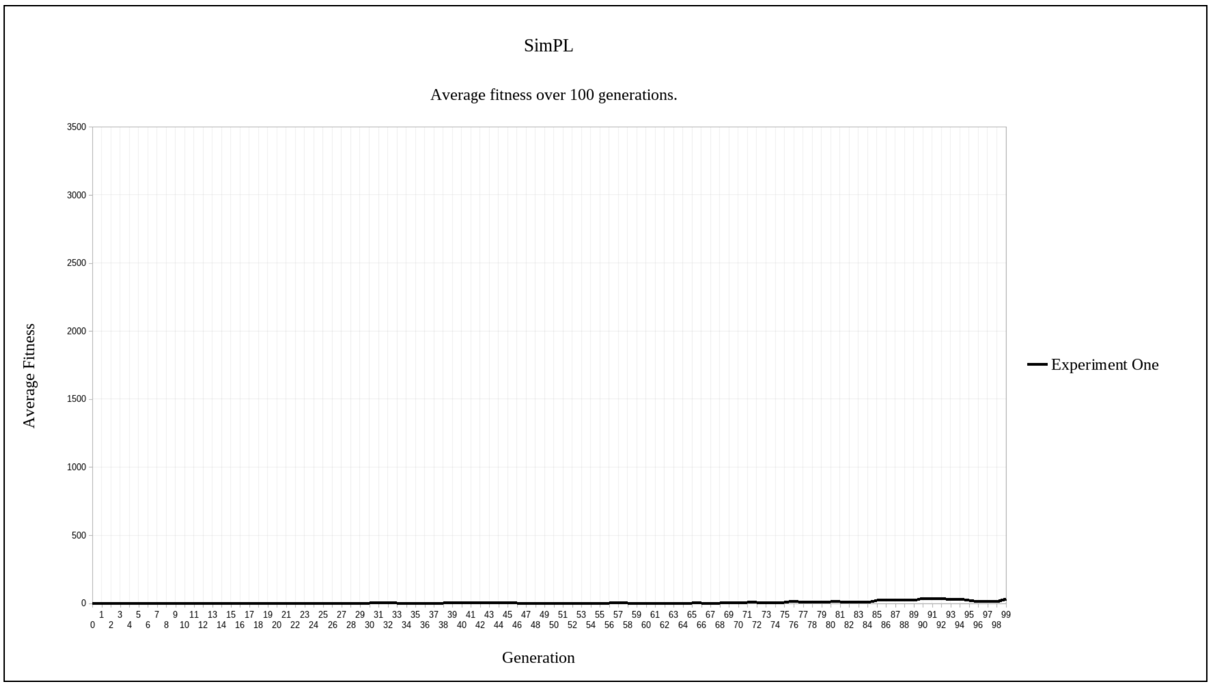


Figure 12: Here you see the average fitness over 100 generations for experiment one.

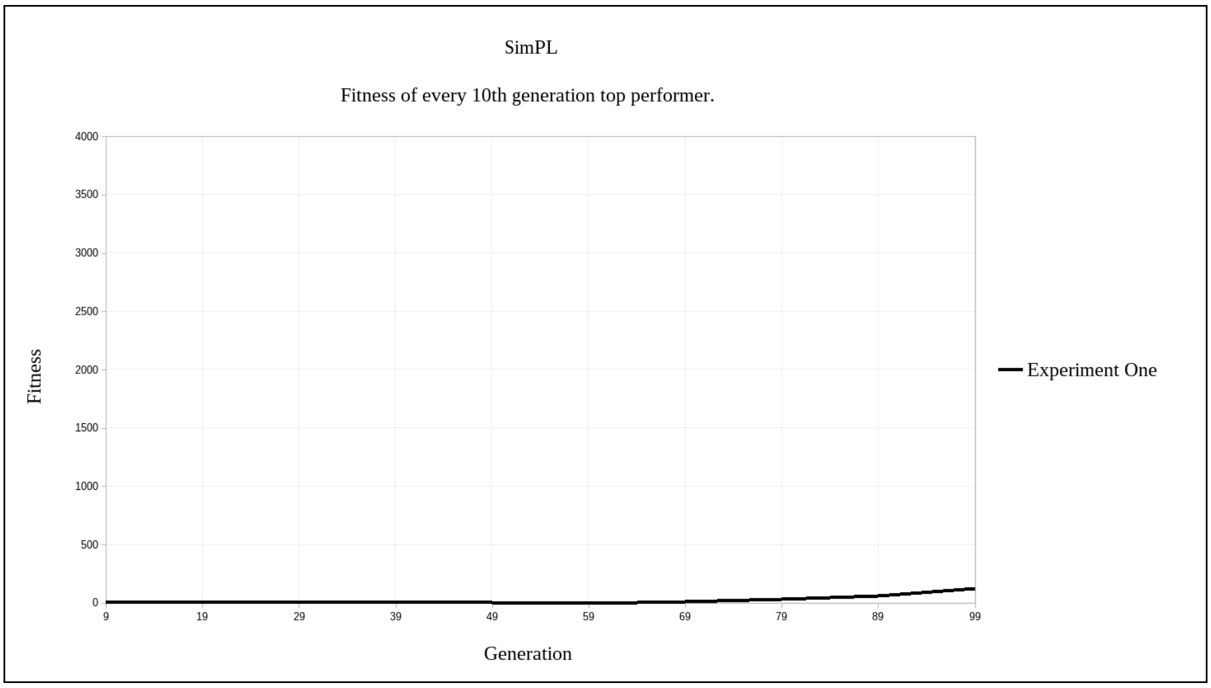


Figure 13: Here you see the fitness of every 10th generation top performer for experiment one.

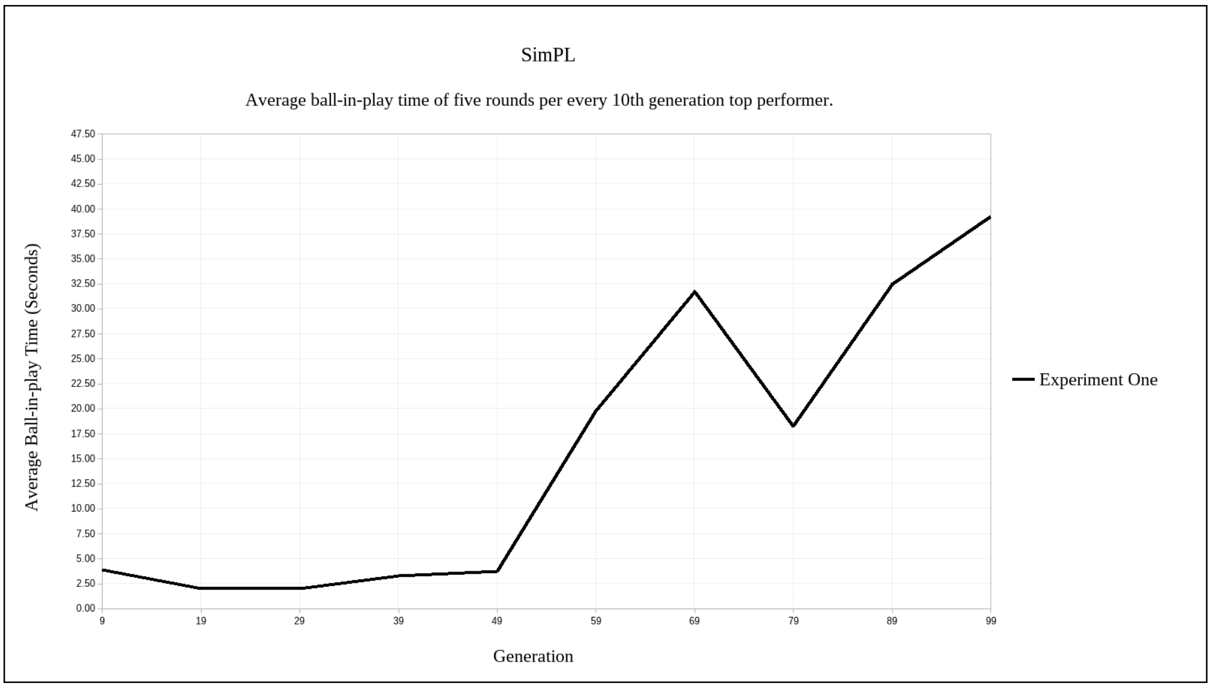


Figure 14: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment one.

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.

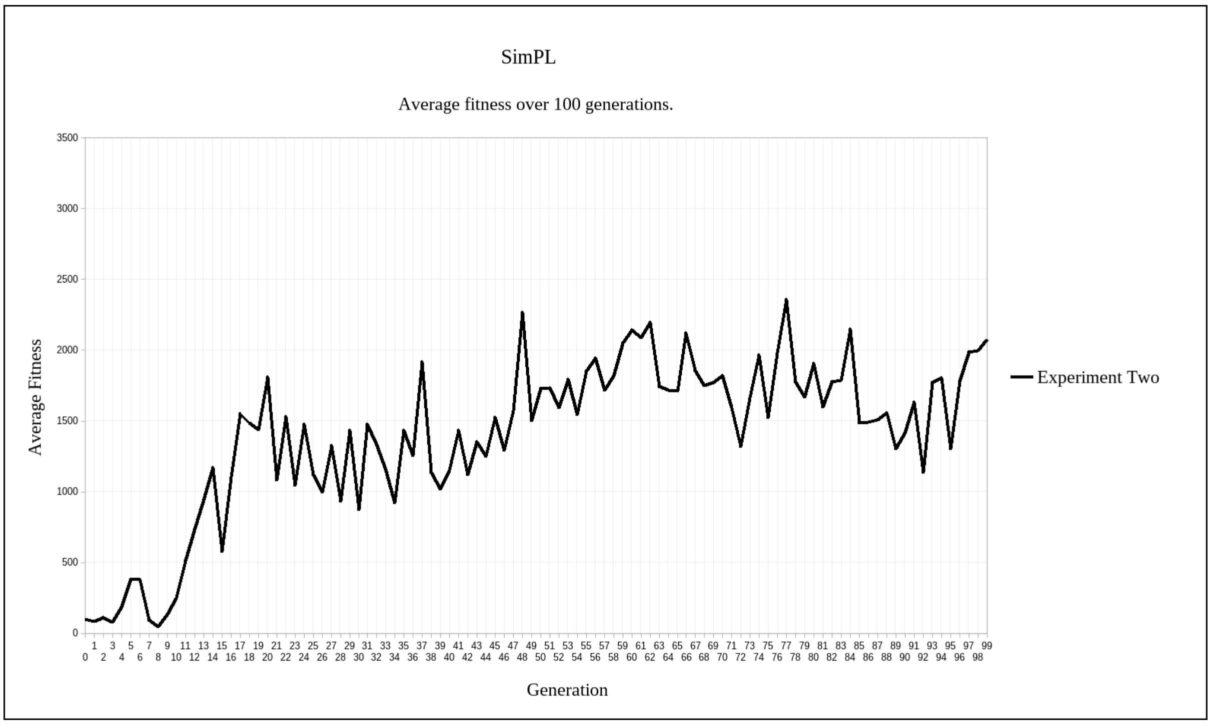


Figure 15: Here you see the average fitness over 100 generations for experiment two.

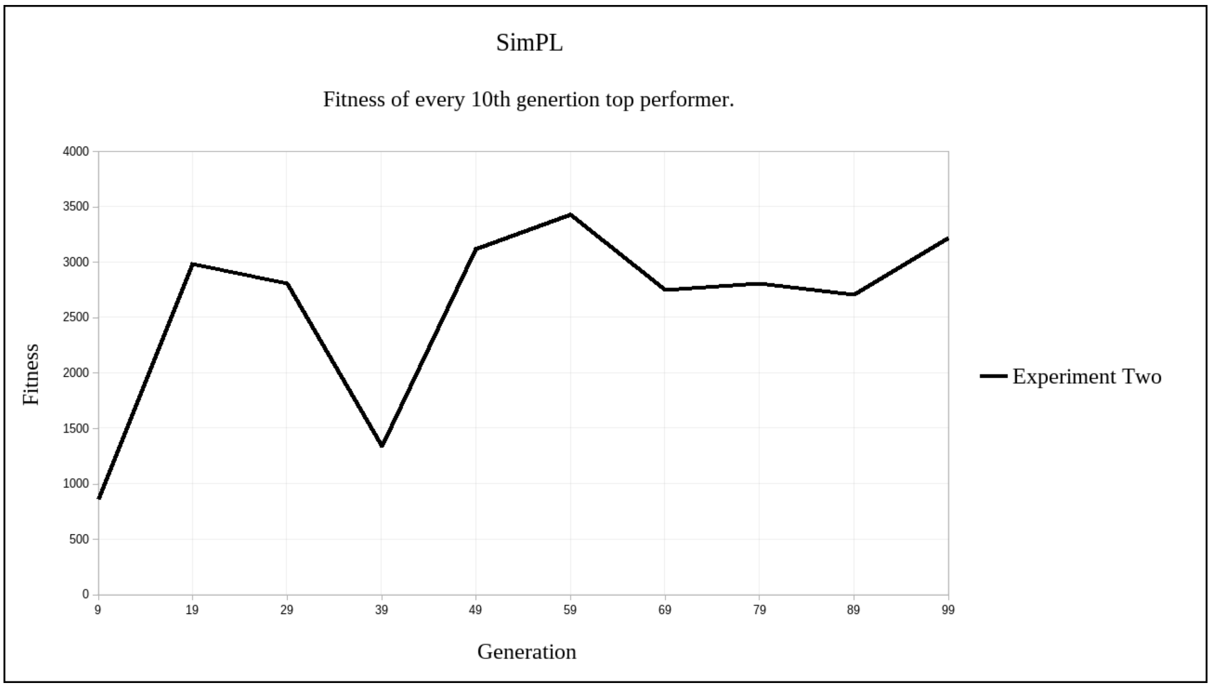


Figure 16: Here you see the fitness of every 10th generation top performer for experiment two.

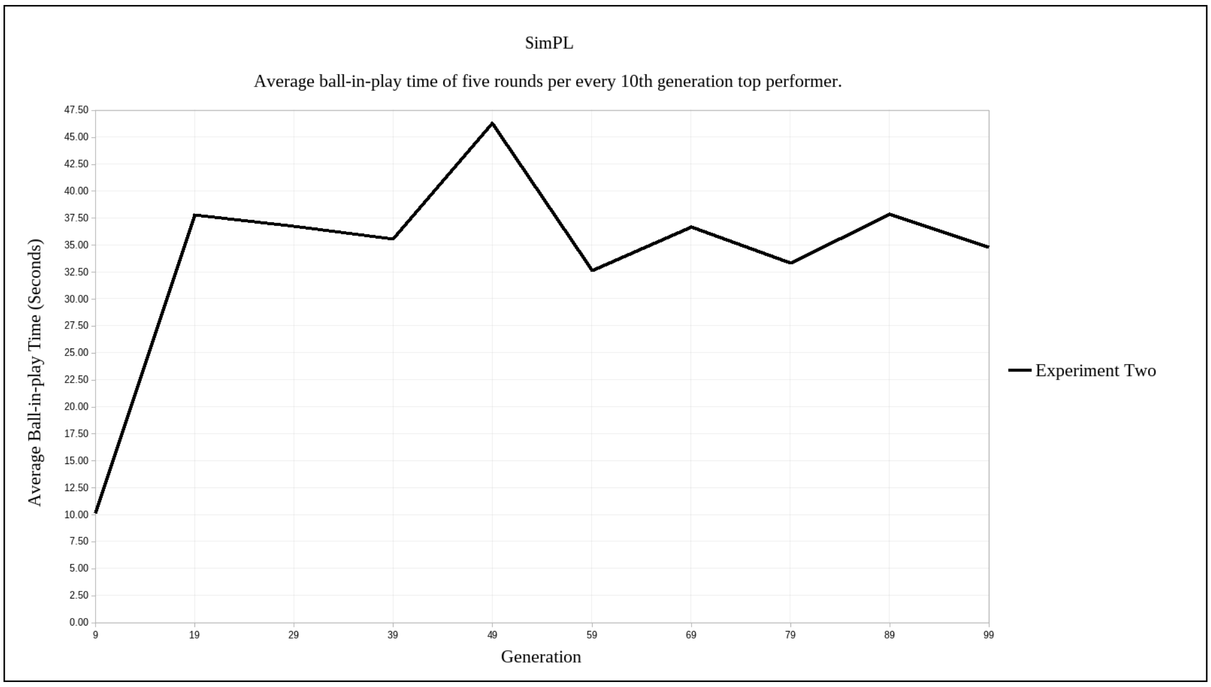


Figure 17: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment two.

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

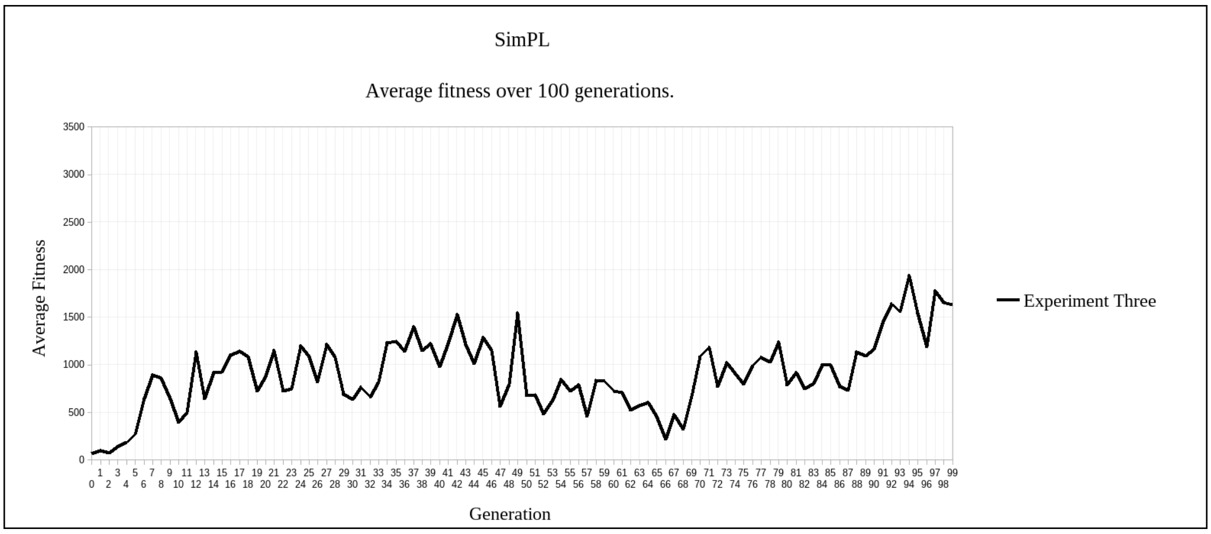


Figure 18: Here you see the average fitness over 100 generations for experiment three.

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 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 [2].

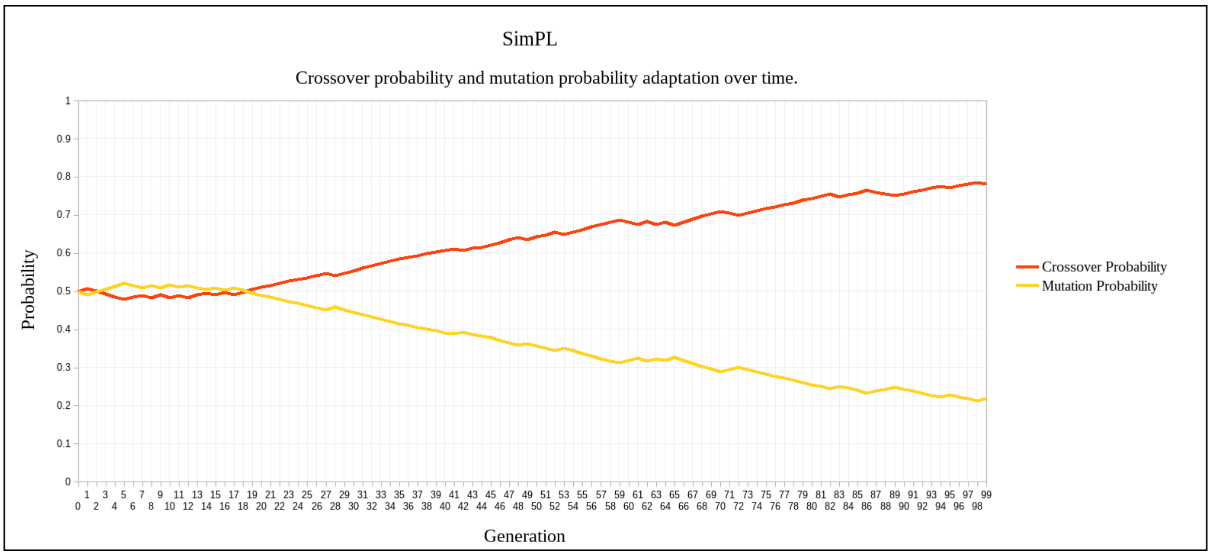


Figure 19: Here you see the self-adaptation of the crossover and mutation probabilities for experiment three.

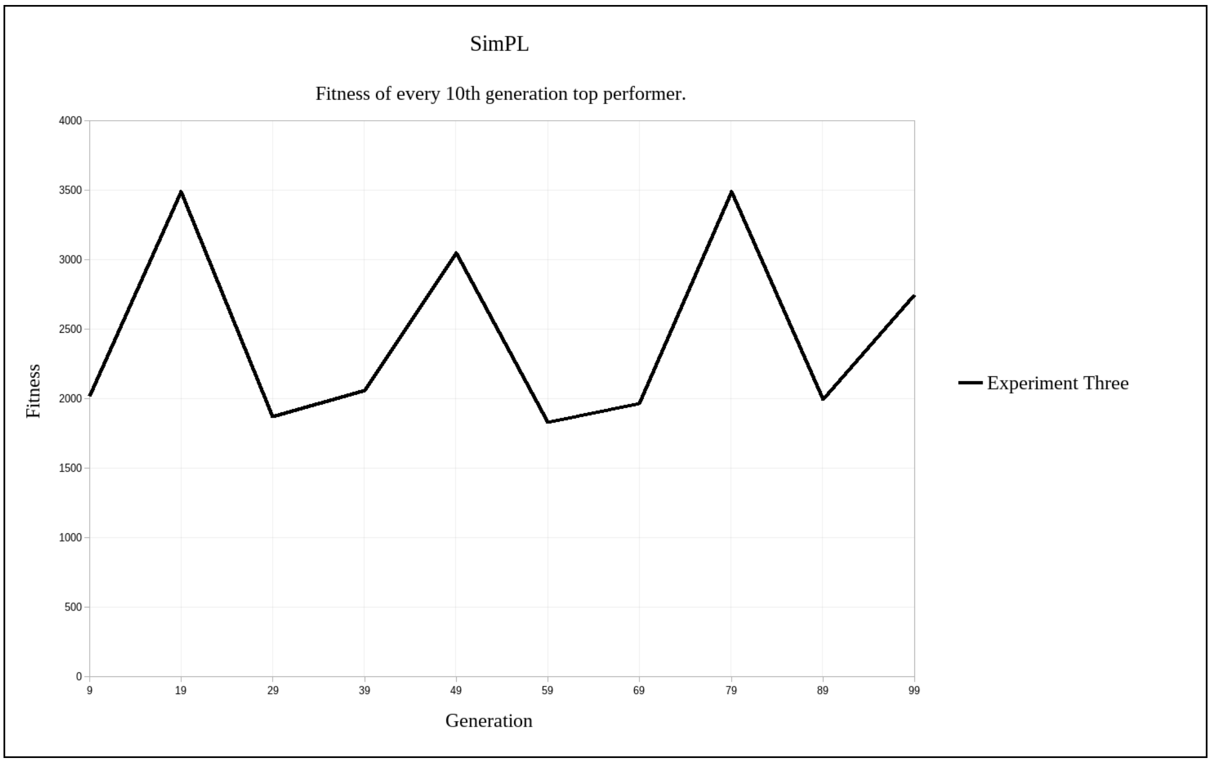


Figure 20: Here you see the fitness of every 10th generation top performer for experiment three.

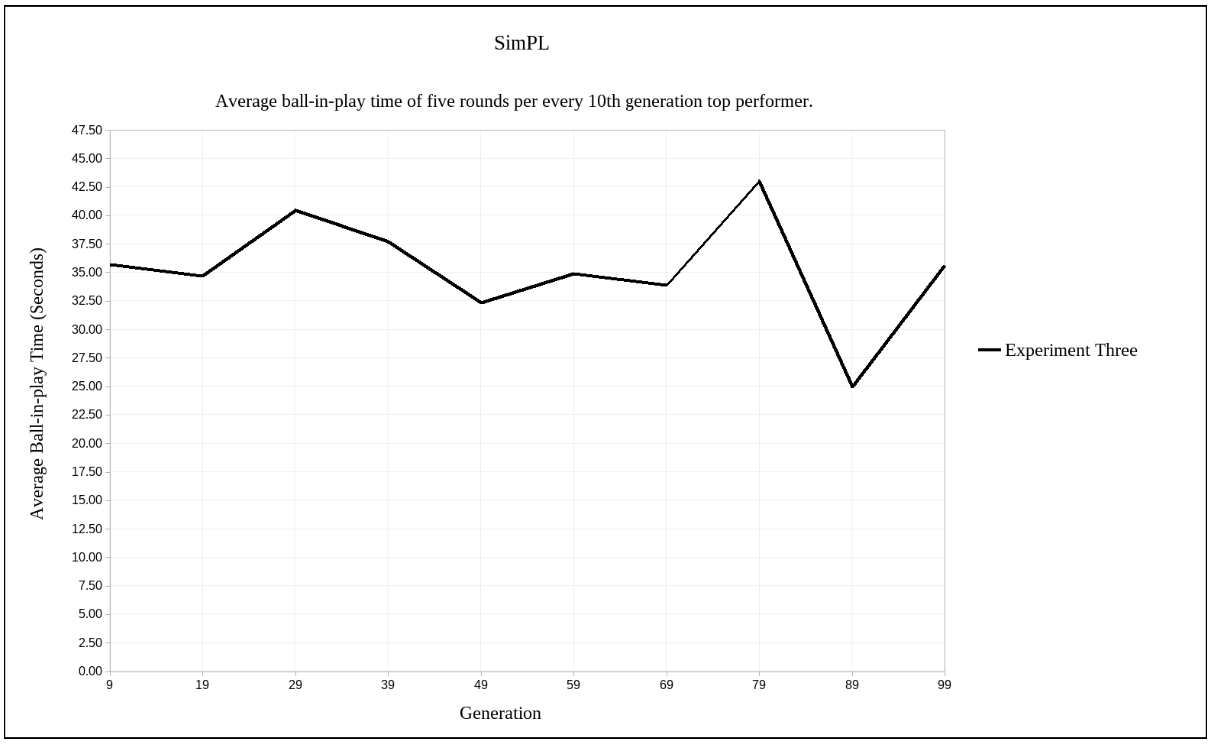


Figure 21: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment three.

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

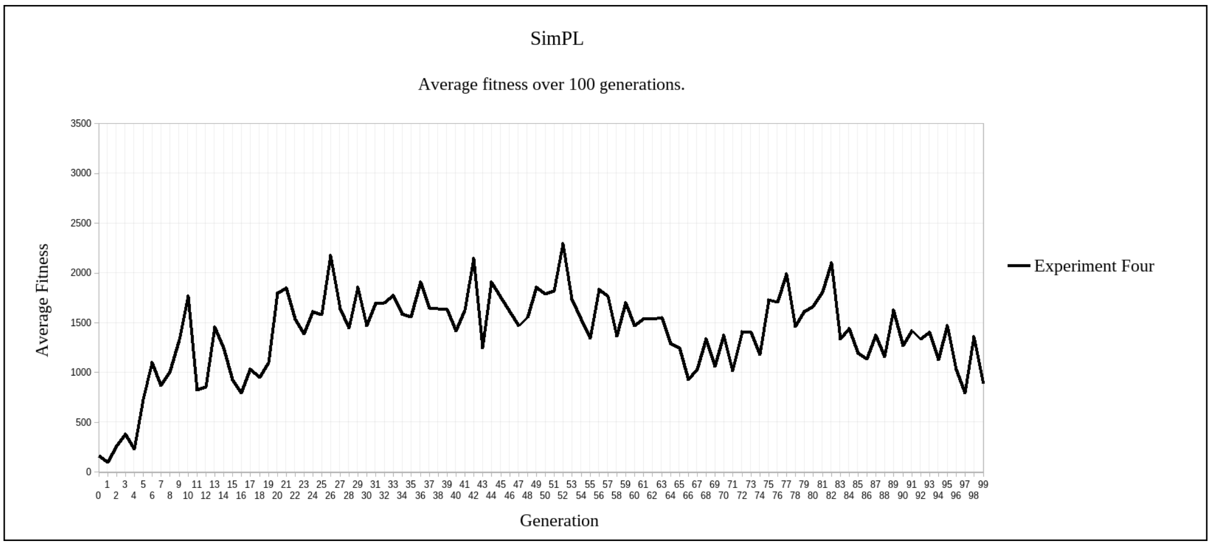


Figure 22: Here you see the average fitness over 100 generations for experiment four.

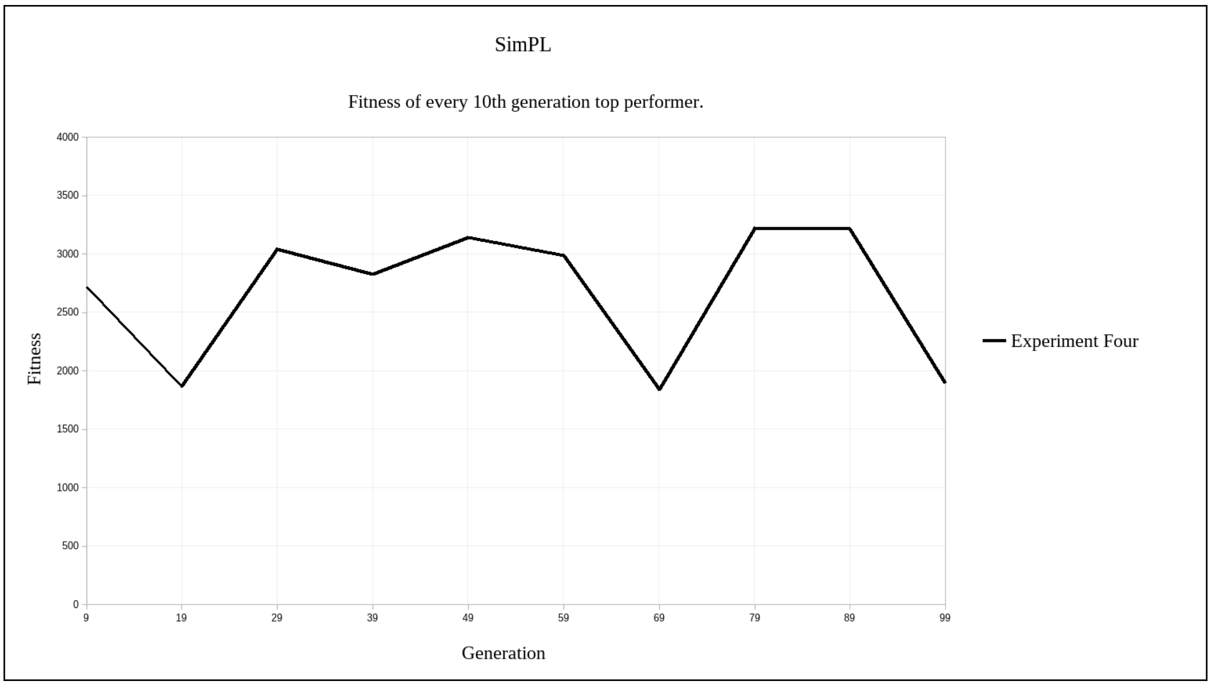


Figure 23: Here you see the fitness of every 10th generation top performer for experiment four.

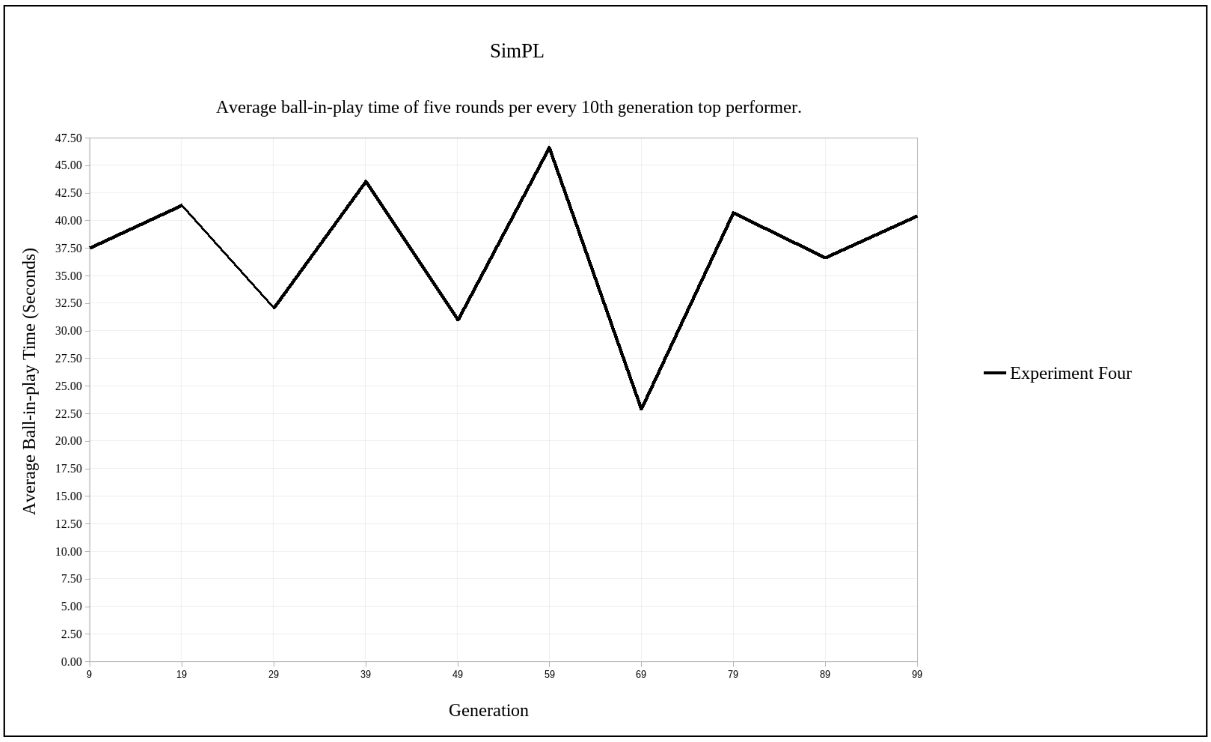


Figure 24: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment four.

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

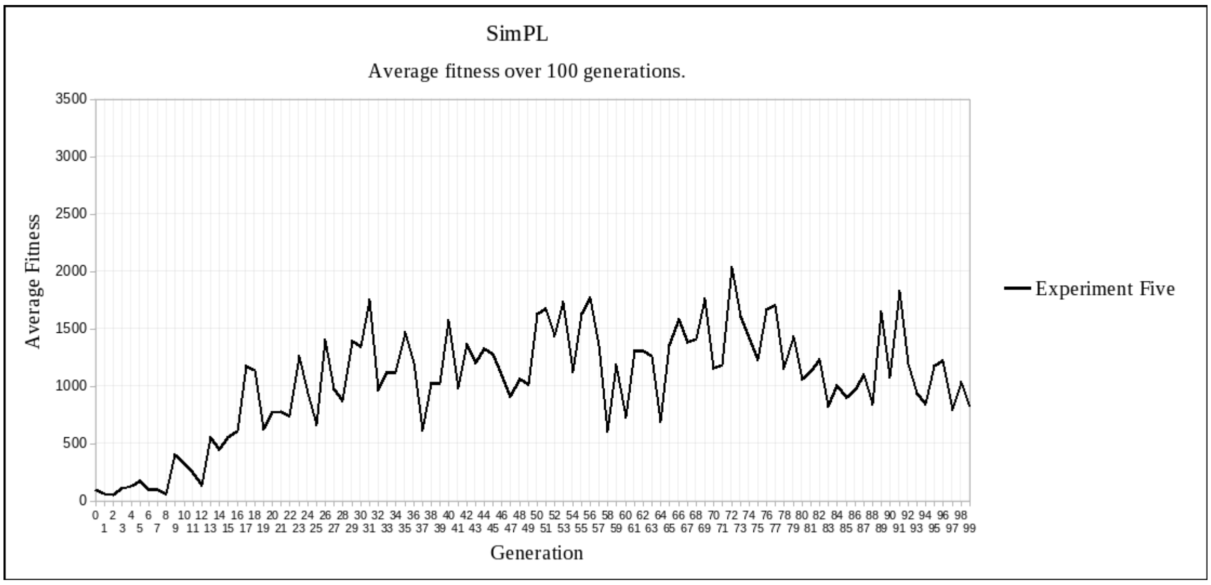


Figure 25: Here you see the average fitness over 100 generations for experiment five.

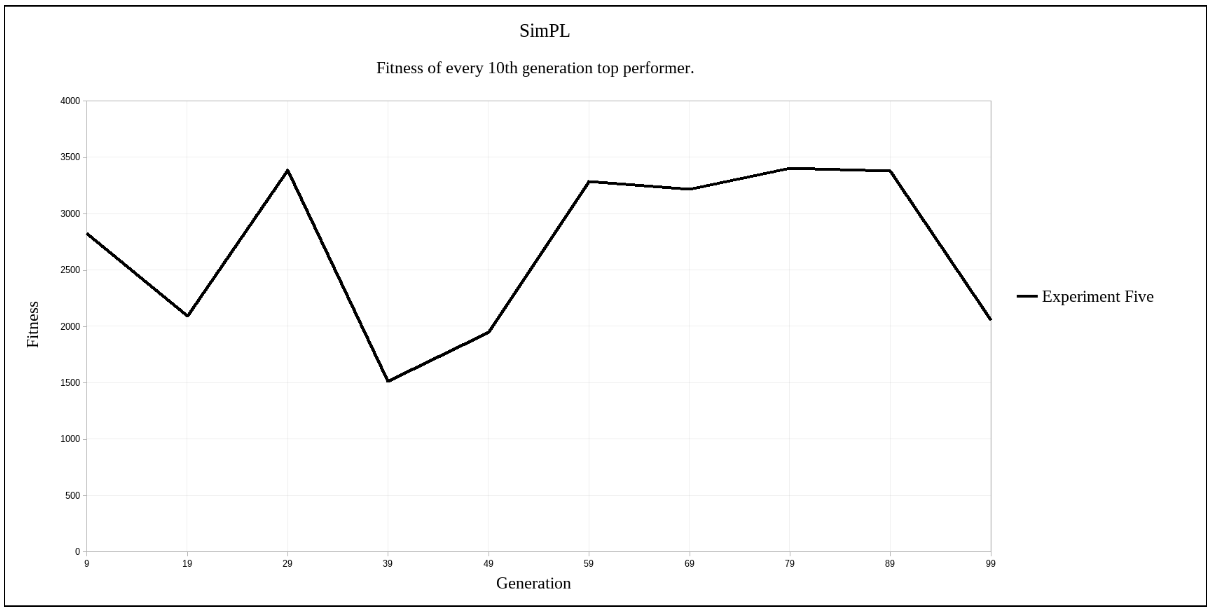


Figure 26: Here you see the fitness of every 10th generation top performer for experiment five.

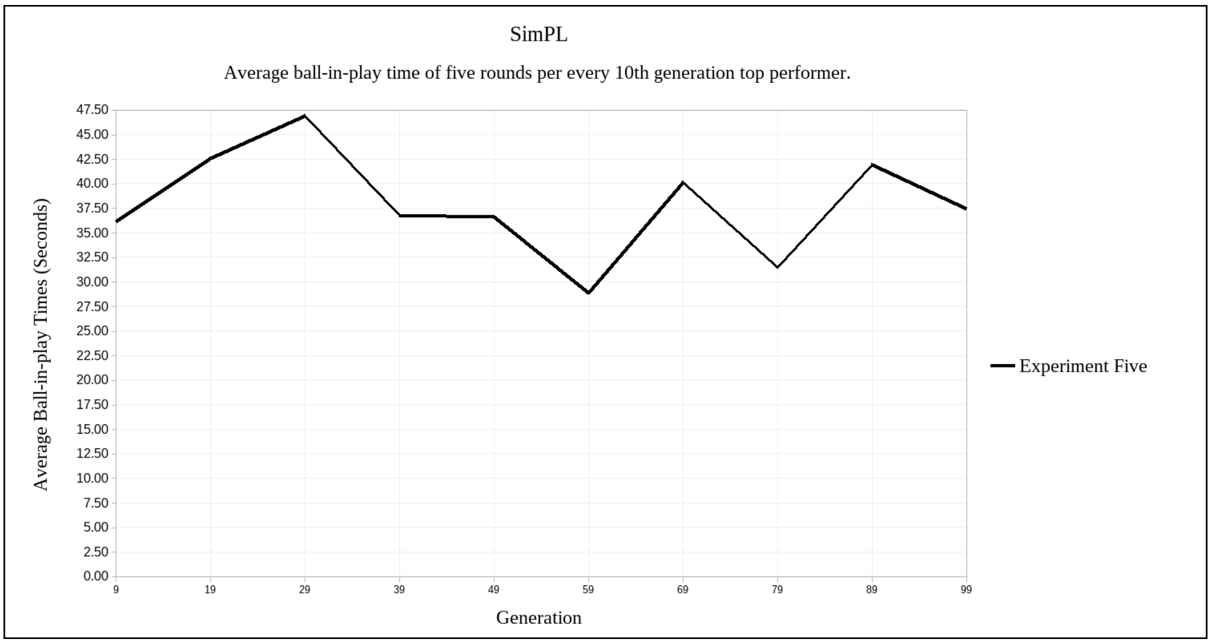


Figure 27: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment five.

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

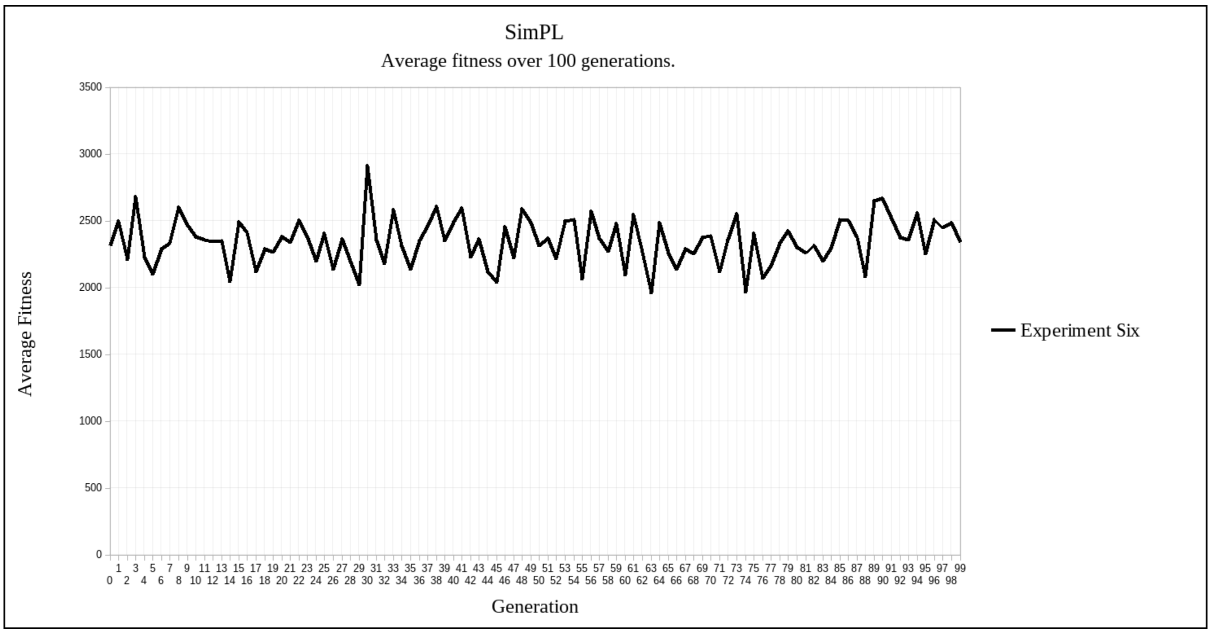


Figure 28: Here you see the average fitness over 100 generations for experiment six.

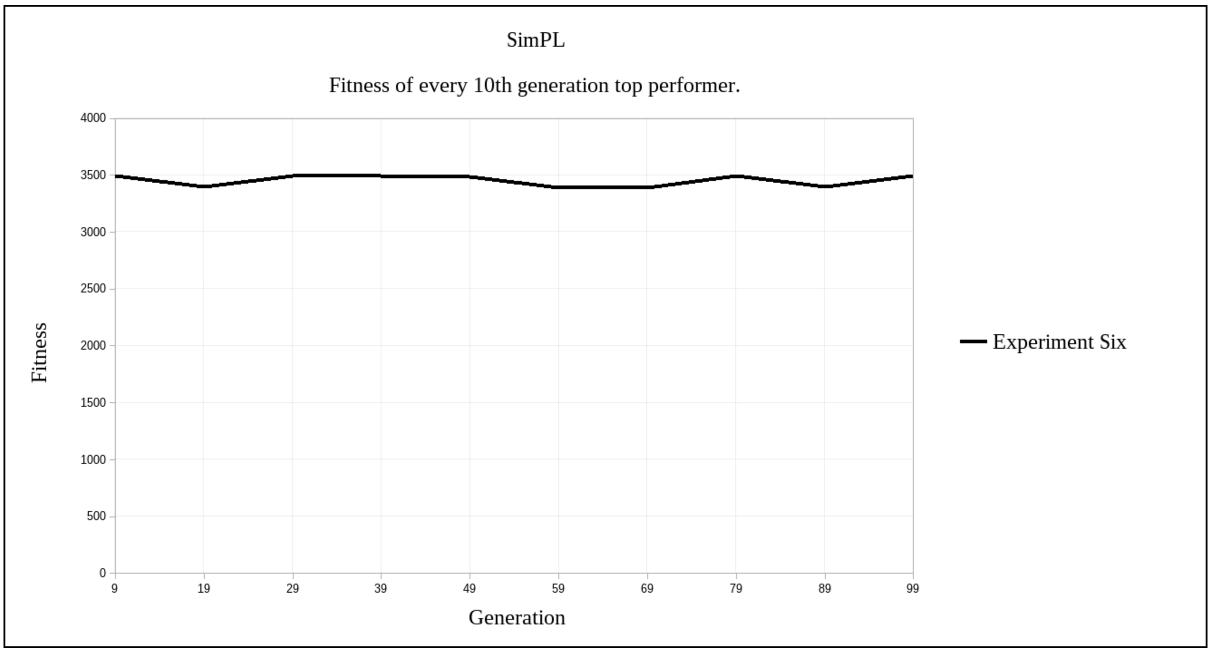


Figure 29: Here you see the fitness of every 10th generation top performer for experiment six.

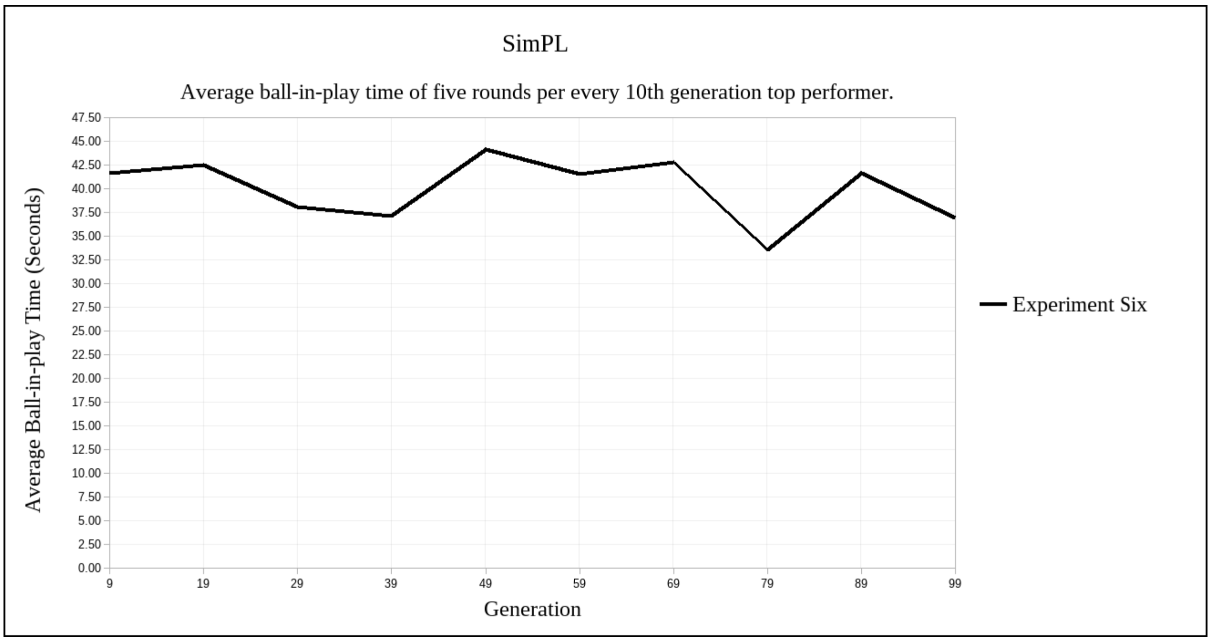


Figure 30: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment six.

5.7 Experiment seven: random paddles.

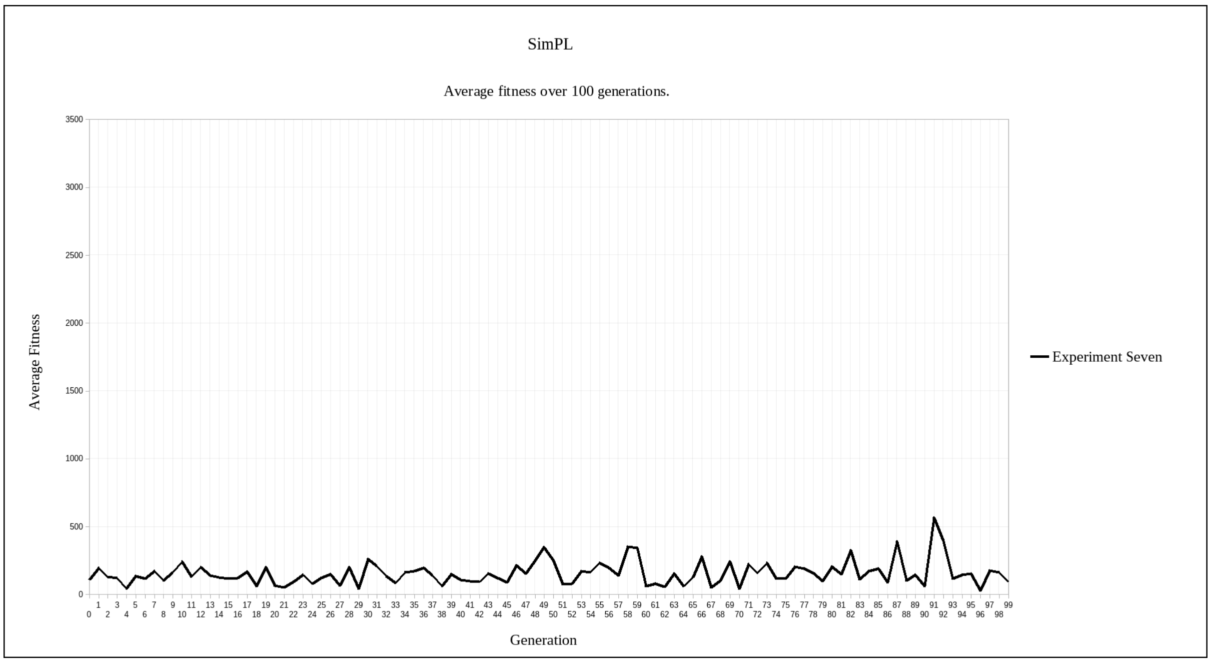


Figure 31: Here you see the average fitness over 100 generations for experiment seven.

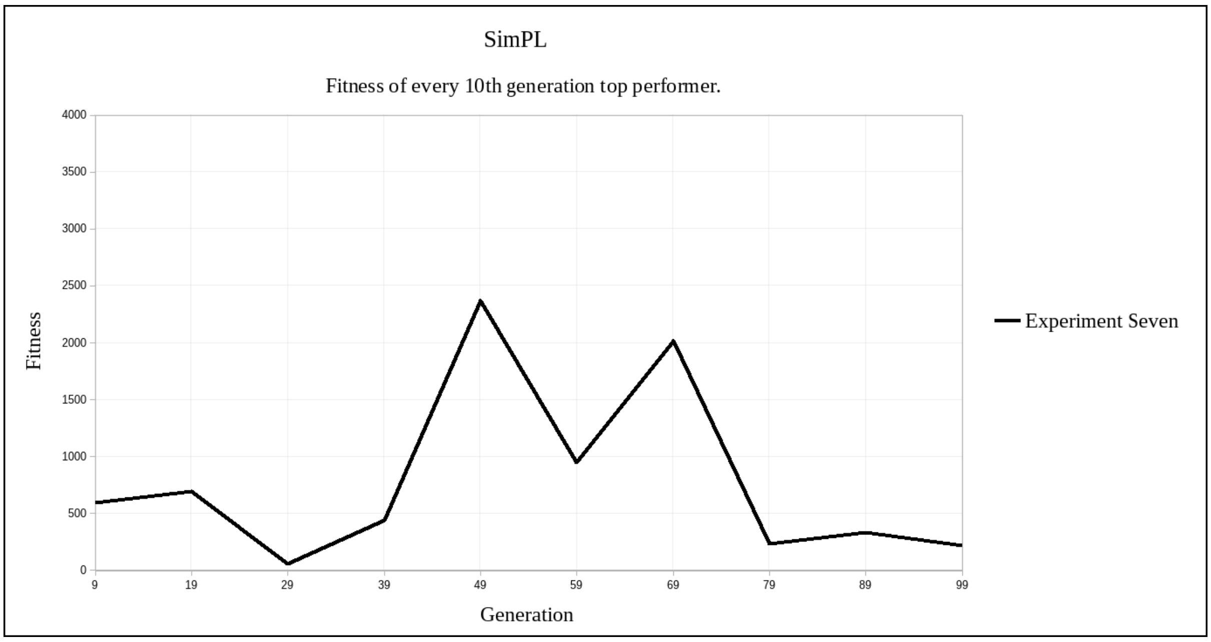


Figure 32: Here you see the fitness of every 10th generation top performer for experiment seven.

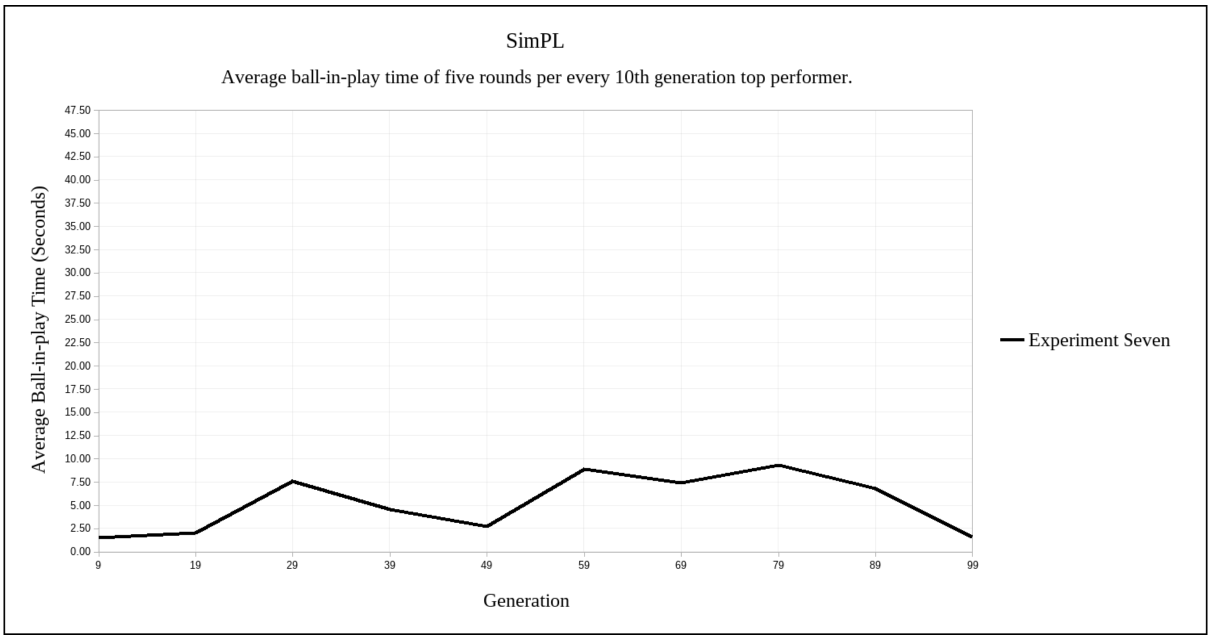


Figure 33: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment seven.

6 Comparative Analysis

6.1 Average fitness over 100 generations.

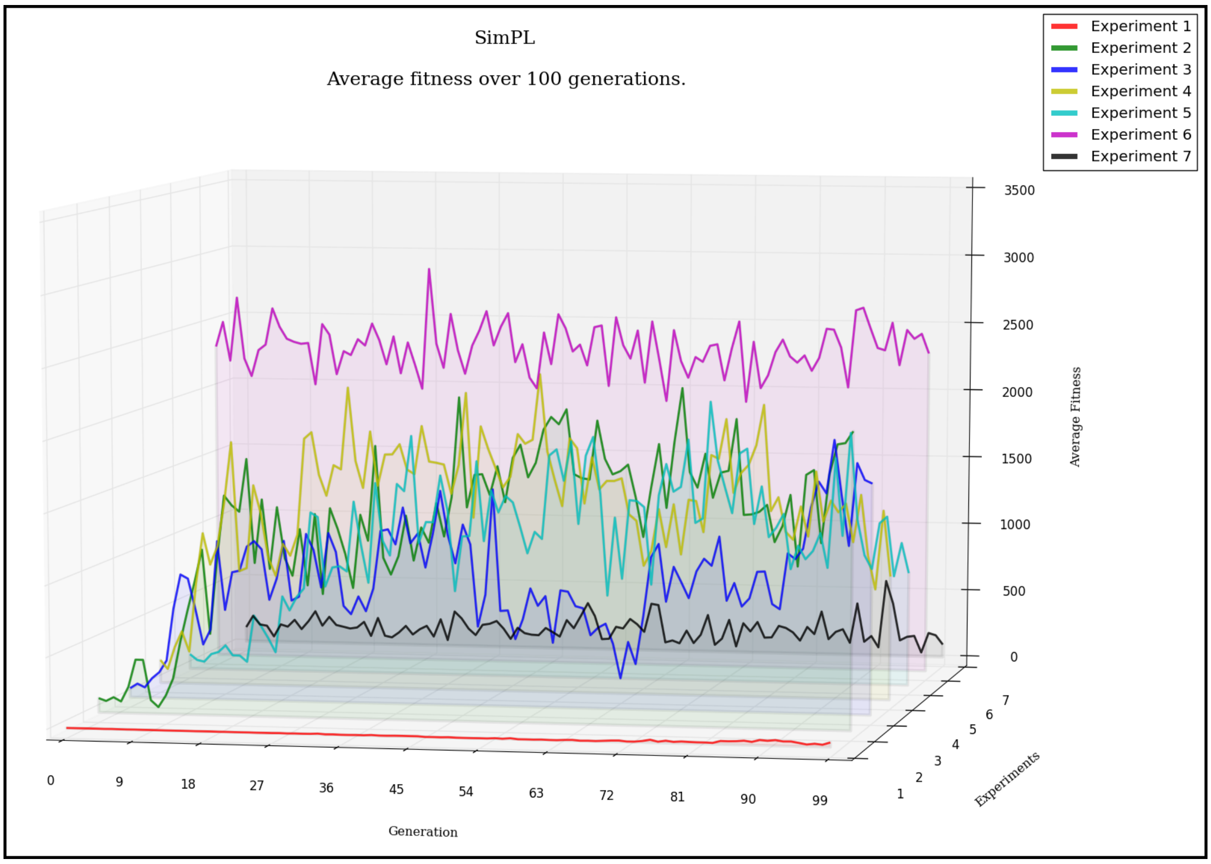


Figure 34: Here you see the average fitness over 100 generations for experiment one through seven.

6.2 Fitness of every 10th generation top performer.

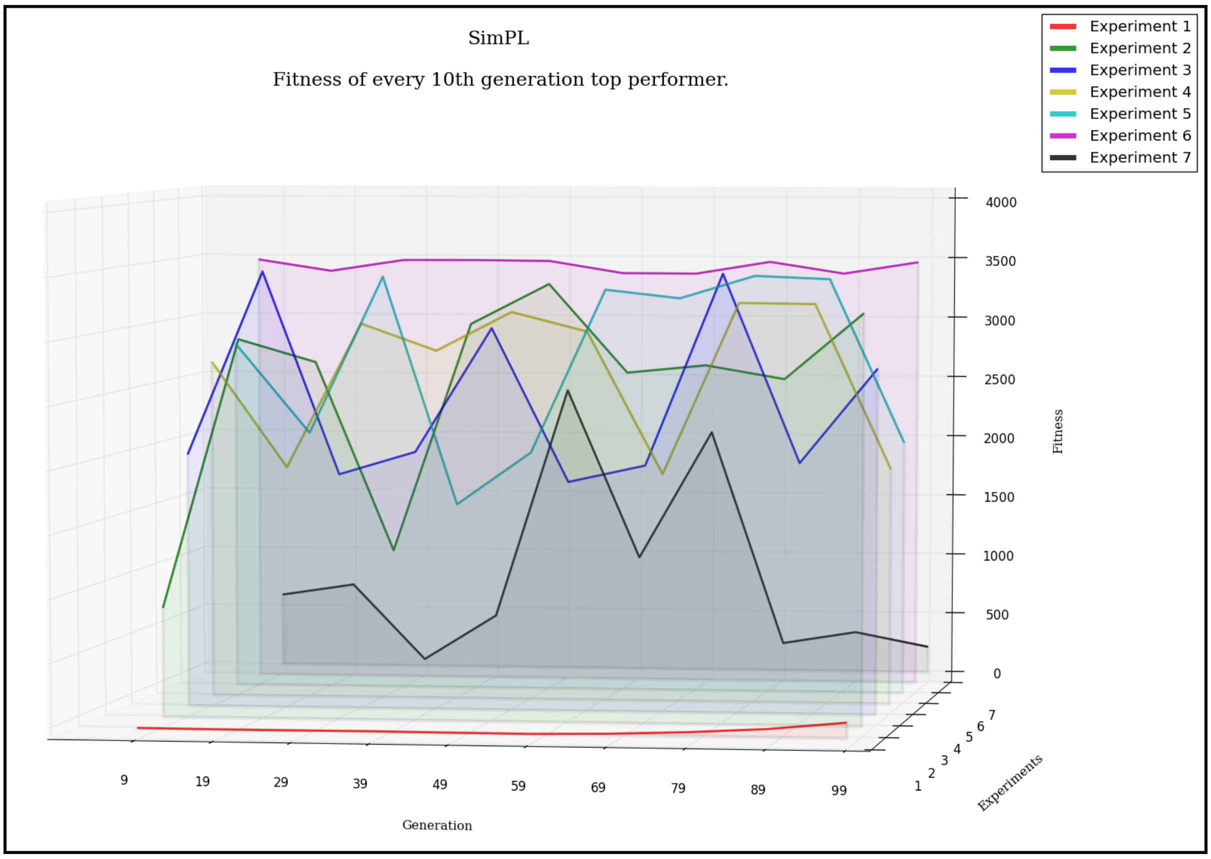


Figure 35: Here you see the fitness of every 10th generation top performer for experiment one through seven.

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

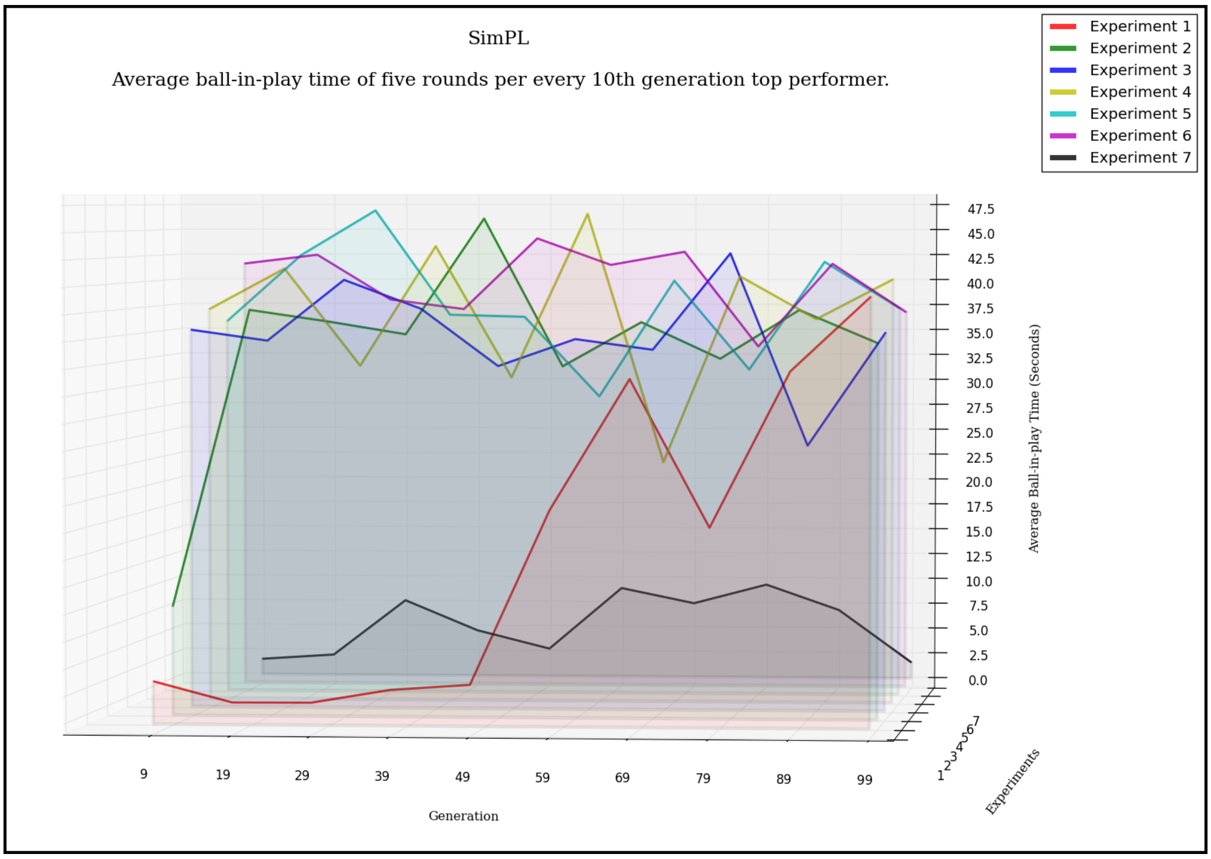


Figure 36: Here you see the average ball-in-play time (in seconds) of five rounds per every 10th generation top performer for experiment one through seven.

7 Conclusion

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 more difficult project, BBAutoTune. And while the problem domain of SimPL and BBAutoTune are quite different, the problems faced and worked-out during the development of SimPL should alleviate the 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 area 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.

References

[1] 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.

[2] 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.