Central Washington University

ScholarWorks@CWU

All Master's Theses | Master's Theses

Winter 2023

Crosshair Optimizer

Jason Torrence
Central Washington University, torrencej@cwu.edu

Follow this and additional works at: https://digitalcommons.cwu.edu/etd

Part of the Other Computer Sciences Commons, and the Theory and Algorithms Commons

Recommended Citation
Torrence, Jason, "Crosshair Optimizer" (2023). *All Master's Theses*. 1835.
https://digitalcommons.cwu.edu/etd/1835

This Thesis is brought to you for free and open access by the Master's Theses at ScholarWorks@CWU. It has been accepted for inclusion in All Master's Theses by an authorized administrator of ScholarWorks@CWU. For more information, please contact scholarworks@cwu.edu.

CROSSHAIR OPTIMIZER

A Thesis

Presented to

The Graduate Faculty

Central Washington University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Computational Science

by

Jason Torrence

February 2023

CENTRAL WASHINGTON UNIVERSITY

Graduate Studies

We hereby approve the thesis of

Jason Torrence

Candidate for the degree of Master of Science

APPROVED FOR THE GRADUATE FACULTY

______________	______________________________
	Dr. Donald Davendra
______________	______________________________
	Dr. Razvan Andonie
______________	______________________________
	Dr. Szilard Vajda
______________	______________________________
	Dean of Graduate Studies

ABSTRACT

CROSSHAIR OPTIMIZER

by

Jason Torrence

February 2023

Metaheuristic optimization algorithms are heuristics that are capable of creating a "good enough" solution to a computationally complex problem. Algorithms in this area of study are focused on the process of exploration and exploitation: exploration of the solution space and exploitation of the results that have been found during that exploration, with most resources going toward the former half of the process. The novel Crosshair optimizer developed in this thesis seeks to take advantage of the latter, exploiting the best possible result as much as possible by directly searching the area around that best result with a stochastic approach.

This research seeks to prove that the Crosshair Optimizer is comparable, if not better in some aspects, to current established metaheuristics optimization algorithms, not only in obtaining optimal results, but usability in high performance computing, and versatility through the use of multiple randomizers and parameter tuning.

ACKNOWLEDGEMENTS

I’d like to acknowledge the Central Washington University Computer Science department’s faculty and staff for all the time and effort they provided and all the department’s resources provided in aid of this research, my family and friends for the support they provided me, and I’d like to acknowledge Dr. Donald Davendra for all his time and effort committed to this research. Without Dr. Davendra’s help, these sentences likely would have never been written.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Optimization algorithms fall within three broad class of heuristics; stochastic algorithms, which includes random-walk [3], Markov chain models [4], etc., mathematical formulations, which encompass integer programming [5], NEH heuristic [6], gradient decent, etc. and metaheuristics, which are algorithms focused on some naturally occurring phenomena such as genetics [7], swarm algorithms [8], etc. These algorithms focus on two aspects, exploration and exploitation. Exploration is generally a *guided* search in the search space, whereas exploitation is generally a *fine grain* search around a local optima such as local search, 2-OPT, 3-OPT [9] being the most common.

Most of the computational resources are devoted to the latter approach, a fine grain search around a fixed point in search space. The tradeoff has always been the exploitation dimension, i.e. how far to map and evaluate. Whereas, random-walk based heuristics, such as Tabu Search [10], went out of favor due to their simplicity, a new wave of stochastic-based algorithms is proving useful, especially with different pseudo-random generators being utilized such as long period oscillators, chaos maps [11], etc.

This research focuses of a novel directed randomized approach named the *Crosshair Optimizer* (CHO) for solving unimodal and multimodal problems. This approach uses multiple generation of random solutions, and only the best solution is used to generate a sequence of new bounds, which are then used to generate new solutions. Three distinct radial sizes are used to generate the new solutions, which aid in both exploration and exploitation. As the population iterates, the average of the population is used as a benched scaling factor in order to increase or decrease these radial sizes.

Three different approaches were developed for this algorithm, the canonical approach using base metrics, which were then tested on standard benchmark functions and compared with other published algorithms. The second approach was to use high performance computing (pthreads) to parallelize the algorithm and speed up its execution time. The final approach taken was to apply different chaos maps as pseudo-random number generators to increase the algorithms effectiveness. All three approaches produced significant improvements to the CHO algorithm.

The thesis is organized as follows: Chapter II introduces the Crosshair Optimizer algorithm, outlines the experiment design, and runs initial experiments to analyze and compare with other published algorithms. Chapter III shows the implementation of High-Performance Computing, and experiments with the use of POSIX Threads and their effect on the algorithm's overall run time. Chapter IV explains different approaches of Chaos Random Number Generation, experiments with the use of these different Randomizers in CHO, and analyzes and compares the randomizers to one another. Chapter V shows how tuning of the user-defined variables may lead to overall better results, and how the tuning of those parameters affects the overall result of the algorithm. This thesis is then concluded in Chapter VI.

CHAPTER II

CROSSHAIR OPTIMIZER

The Crosshair Optimizer (CHO) is an optimization algorithm largely based on random values within changing bounds. The algorithm follows the following steps: Initialization, then iterations of particle movement followed by parameter adjustment. This is given mathematically and explained in further detail in this chapter.

Initialization

The algorithm population P is a matrix of size D (number of solution vectors) by N (dimension of each solution vector), which is randomly initialized between the lower (L) and upper (U) bounds $(rand(L, U))$. Once initialized, the population is evaluated for its fitness $PFit$ and the best solution $bSol$ and its associated fitness $bFit$ is obtained. This process is shown in Algorithm 1.

```
Data: $P = \emptyset, PFit = \emptyset, bSol = \emptyset, bFit = MaxVal$
Population initialization;
for i from 0 to D do
    for j from 0 to N do
        $P_{i,j} = rand(L, U)$;
    end
    $PFit_i = f(P_i)$
    if $PFit_i < bFit$ then
        $bSol \leftarrow PFit_i$
        $bFit = PFit_i$
    end
end
```

Algorithm 1: Population initialization and best solution

Particle Movement

CHO has three unique control parameters for particle adjustment, termed **far** (f_{adj}), **close** (c_{adj}) and **near** (n_{adj}) adjustment. Once the best solution ($bSol$) and its fitness ($bFit$) has been obtained, new adjusted upper and lower bounds are calculated using the adjustment values. Three different bounded values are therefore computed as $L_f, U_f, L_c, U_c, L_n, U_n$, where the subscript represent far, close and near adjustments as given in (2.1).

$$\begin{aligned} L_f &= bSol - (r \times f_{adj}), U_f = bSol + (r \times f_{adj}) \\ L_c &= bSol - (r \times c_{adj}), U_c = bSol + (r \times c_{adj}) \\ L_n &= bSol - (r \times n_{adj}), U_n = bSol + (r \times n_{adj}) \end{aligned} \tag{2.1}$$

where r represents the range of the original problems upper and lower bounds ($r = U - L$).

In addition, another very important input variable is the problem domain variability (var_p), which is a selection parameter designated to every problem function F. The range of this variable is from $0.5 - 1.0$ and its implication is for new particle generation. Once the new bounds are calculated using the best particle, a random number ($rand$) is generated and checked against this variability index. If the random value is less than the index, then the subsequent solution generation takes place within the new bounds, otherwise the original bounds are used for next solution generation.

New Particle Generation

The new particles are generated using a split population approach. The population is divided into three parts based on its size D. The first third of the population particle's are generated using the **far** (f_{adj}) adjusted bounds L_f, U_f given as (2.2):

$$P_i = rand\,(L_f, U_f) \tag{2.2}$$

The second third of the solutions are generated using the **close** (c_{adj}) adjusted bounds L_c, U_c given as (2.3):

$$P_i = rand\,(L_c, U_c) \tag{2.3}$$

The remaining solutions are generated using the **close** (n_{adj}) adjusted bounds L_n, U_n given as (2.4):

$$P_i = rand\,(L_n, U_n) \tag{2.4}$$

where i is the i^{th} dimension index in the specific solution.

In the case where the generated random value is higher than the var_p, the new solution is generated using the original bounds. The new solution is then evaluated for its fitness.

This process can be seen in Algorithm 2.

Population Iteration

Three additional user defined parameters are now introduced, the problem iteration ($iter$), the total experiment run (tr) and the maximum experimentation runs (mer). $iter$ refers to the number of evaluations or new population generations the algorithm undergoes, per each test run tr and the mer represents the total adjustments the algorithm encounters over its iterations.

Once the new population has been generated, it is evaluated for its fitness function and its best solution is obtained. This value is saved in the vector of best iterations results

```
Data: $P, PFit, bSol, U_f, L_f, U_c, L_c, U_n, L_n, r = U - L$
for i from 0 to N do
    $L_f = bSol_i - (r \times f_{adj}), U_f = bSol_i + (r \times f_{adj})$
    $L_c = bSol_i - (r \times c_{adj}), U_c = bSol_i + (r \times c_{adj})$
    $L_n = bSol_i - (r \times n_{adj}), U_n = bSol_i + (r \times n_{adj})$
    for j from 0 to D do
        if $rand < adj$ then
            if $j < D/3$ then
                $P_j = rand\,(L_f, U_f)$
            end
            else if $j < D/(1.5)$ then
                $P_j = rand\,(L_c, U_c)$
            end
            else
                $P_j = rand\,(L_n, U_n)$
            end
        end
        else
            $P_j = rand\,(L, U)$
        end
    end
    $PFit_i = f(P_i)$
end
```
Algorithm 2: New solution vector generation

($bSol_{tr}$). Additionally, the new best solution is compared with the global best solution $bSol$, and updated if it has improved on it.

At this point the population is cleared and a new population is generated as given in Algorithm 1 and 2. These algorithms are used in conjunction in Algorithm 3.

```
Data: $P, bSol, bSol_{iter}, iter$
for i from 0 to $tr$ do
    for j from 0 to $iter$ do
        Algorithm 1
        Algorithm 2
    end
    $bSol_i \leftarrow min(PFit)$
    if $bSol_i < bFit$ then
        $bFit = bSol_i$
        $bSol \leftarrow min(P_i)$
    end
end
```

Algorithm 3: Population iteration

Population Parameter Adjustment

Once the total problem iterations (tr) for a specific experimentation run (mer) has completed, the average (avg) and standard deviation (std) of the saved fitness values in $bSol_{iter}$ is computed. The newly calculated avg is compared and updated with the global best average $gAvg$ if it has improved.

If the current avg value is the same as the $gAvg$, then an increase test bound procedure is done to increase the bounds of the generated solutions. If it is not the same, then a reduction procedure is applied to decrease the bounds. Another two user defined scaling parameters, reduction (red) and enlargement (enl) are required for this adjustment as given in the following equations:

$$\begin{aligned} x_{adj} &= x_{adj} \times \text{red} \\ x_{adj} &= x_{adj} \times \text{enl} \end{aligned} \tag{2.5}$$

where x designates the far (f_{adj}), near (n_{adj}) and close (c_{adj}) adjustment parameters.

These updated parameters are then used in the next experiment run as an adaptive measure for the new population generation.

Termination Criteria

The algorithm termination is based on a number of criteria. The first one is if the maximum experimentation runs (mer) has been reached. Secondly, a comparison is done with the averages split into two components. If the current avg is the $gAvg$ or a user defined minimum bound parameter ($minB$) multiplied with the current avg is less than the the $gAvg$ value, the code continues to be run. Once these criteria have been met, the program terminates, such as is shown in Algorithm 4.

The output of the algorithm is the best obtained solution $bSol$ and its associated fitness $bFit$.

Experimentation

Table 1 gives the operating parameters of the CHO algorithm and Table 2 outlines the functions used in this research. The experimentation was conducted on thirteen functions, of which the first seven are unimodal and the other six are multimodal functions [1]. The last column give the problem domain variability (var_p) of each problem.

```
**Data:** $P, bSol, bSol_{iter}, avg, gAvg, mer, red, enl, j = 0$
**Result:** $gSol, bFit$
**do**
    **Algorithm 3**
    **for** *i from 0 to D* **do**
        $avg = avg + bSol_i$
    **end**
    $avg = avg/D$
    **if** $avg < gAvg$ **then**
        $gAvg = avg$
    **end**
    **if** $avg == gAvg$ **then**
        $x_{adj} = x_{adj} \times \text{red}$
    **end**
    **else**
        $x_{adj} = x_{adj} \times \text{enl}$
    **end**
    $j = j + 1$
**while** $((avg == gAvg \ \vee \ avg \times minB < gAvg) \ \wedge \ (j < mer))$;
```

Algorithm 4: Overall CHO algorithm

TABLE 1: CHO parameters

Para	Value	Para	Value	Para	Value
D	100	f_{adj}	0.5	red	0.5
N	30	c_{adj}	0.25	enl	2.0
$iter$	100	n_{adj}	0.125	$minB$	0.8
tr	100	mer	1000		

TABLE 2: Test Functions [1]

Functions	Dim	Range	$F_{\min}$	var_p
$F_1(x) = \sum_{i=1}^{n} x_{\mathrm{i}}{}^2$	30	[-100,100]	0	0.95
$F_2(x) = \sum_{i=1}^{n} \lvert x_{\mathrm{i}} \rvert + \prod_{i=1}^{n} \lvert x_{\mathrm{i}} \rvert$	30	[-10,10]	0	0.98
$F_3(x) = \sum_{i=1}^{n} (\sum_{j-1}^{i} x_{\mathrm{j}})^2$	30	[-100,100]	0	0.97
$F_4(x) = max_{\mathrm{i}}(\lvert x_i \rvert, 1 <= i <= n)$	30	[-100,100]	0	0.99
$F_5(x) = \sum_{i=1}^{n-1} 100(x_{\mathrm{i+1}} - x_{\mathrm{i}}{}^2)^2 + (x_i - 1)^2$	30	[-30,30]	0	0.92
$F_6(x) = \sum_{i=1}^{n} (x_{\mathrm{i}} + 0.5)^2$	30	[-100,100]	0	0.99
$F_7(x) = \sum_{i=1}^{n} i x_{\mathrm{i}}{}^4 + random[0,1)$	30	[-1.28,1.28]	0	0.99
$F_8(x) = \sum_{i=1}^{n} -x_{\mathrm{i}} sin(\sqrt{\lvert x_{\mathrm{i}} \rvert})$	30	[-500,500]	-418.9829 x 5	0.99
$F_9(x) = \sum_{i=1}^{n} [x_{\mathrm{i}}{}^2 - 10cos(2\pi x_{\mathrm{i}}) + 10]$	30	[-5.12,5.12]	0	0.98
$F_{10}(x) = -20exp(-0.2\sqrt{1/n \sum_{i=1}^{n} x_{\mathrm{i}}{}^2}) - exp(1/n \sum_{i=1}^{n} cos(2\pi x_{\mathrm{i}}))$ $+20 + e$	30	[-32,32]	0	0.98
$F_{11}(x) = 1/4000 \sum_{i=1}^{n} x_{\mathrm{i}}{}^2 - \prod_{i=1}^{n} cos(x_{\mathrm{i}}/\sqrt{i}) + 1$	30	[-600,600]	0	0.99
$F_{12}(x) = \pi/n(10sin(\pi y_1) + \sum_{i=1}^{n-1} (y_{\mathrm{i}} - 1)^2[1 + 10sin^2(\pi y_{\mathrm{i+1}})]$ $+(y_{\mathrm{n}} - 1)^2) + \sum_{i=1}^{n} u(x_{\mathrm{i}}, 5, 100, 4)$ $Y_{\mathrm{i}} = 1 + (x_{\mathrm{i}} + 1)/4$ $u(x_i, a, k, m) = \begin{cases} k(x_i - a)^m & x_i > a \\ 0 & -a < x_i < a \\ k(-x_i - a)^m & x_i < a \end{cases}$	30	[-50,50]	0	0.98
$F_{13}(x) = 0.1(sin^2(3\pi x_1) + \sum_{i=1}^{n} (x_{\mathrm{i}} - 1)^2[1 + sin^2(3\pi x_{\mathrm{i}} + 1)]$ $+(x_{\mathrm{n}} - 1)^2[1 + sin^2(2\pi x_{\mathrm{n}})]) + (\sum_{i=1}^{n} u(x_{\mathrm{i}}, 5, 100, 4))$	30	[-50,50]	0	0.98

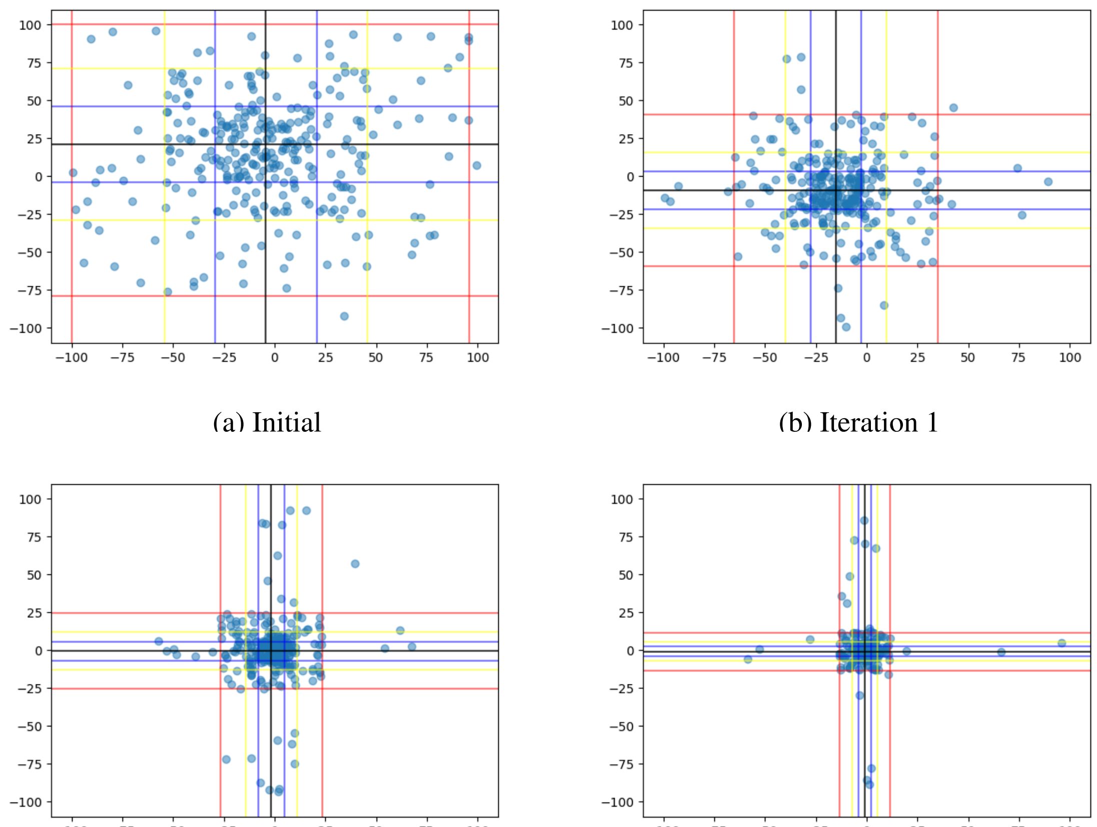
Results and Analysis

The results for the CHO algorithm is given in Table 3 with the average, standard deviation, best obtained result and the total execution time. The crosshair iteration scatter plots is given in Figures 1 and 2 for the different population iteration points. While the initial graph is a true random scatter plot, it is visible that during iteration, the solution converge towards the optima with only two sideways branches extending off the best solution. These values are of the solutions, which are generated further from the minima, and allows for a level of exploration.

Figure 3 shows the different evolution plots for the algorithm in different functions. One of the core attributes is that the CHO algorithm is able to find a good solution within a small iteration period, with less exploration and more exploitation utilized.

TABLE 3: CHO Results

F	Avg	Std	Best	time (s)
1	0	0	0	750.834
2	0	0	0	483.289
3	0	0	0	1381.206
4	0	0	0	442.537
5	19.9064	11.219	0.0873	583.037
6	0	0	0	434.678
7	11.869	0.8339	6.6184	622.303
8	-5706.537	2.2278E-12	-5706.537	850.763
9	0	0	0	717.529
10	7.11E-15	0	3.55E-15	818.621
11	5.42E-20	0	0	893.524
12	0.04976	3.0954E-17	0.0497	1248.399
13	0.00261	1.0324E-18	0.0026	1162.112


(a) Initial

(b) Iteration 1

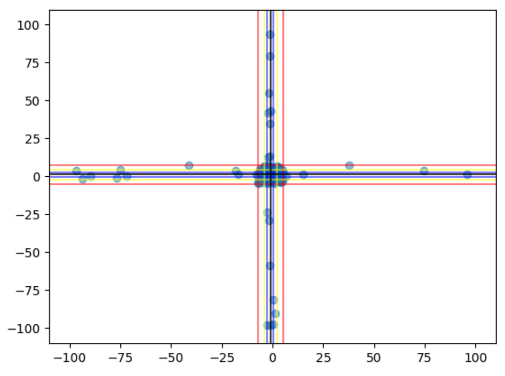
(c) Iteration 2

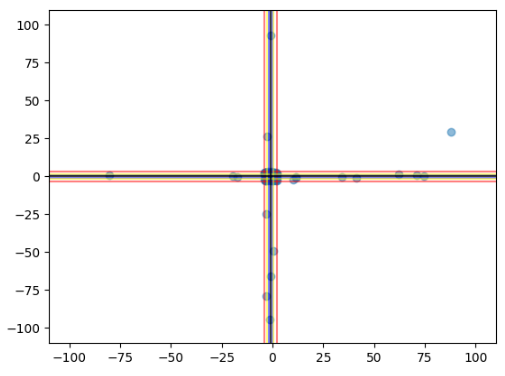
(d) Iteration 3

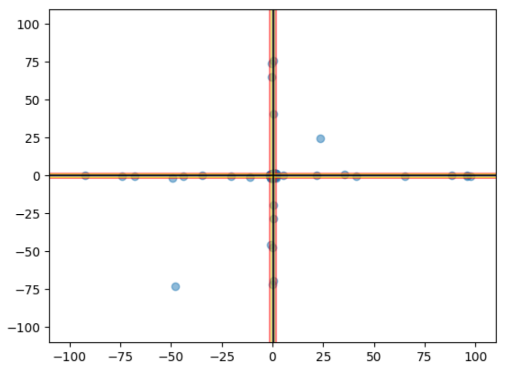
FIGURE 1: Crosshair Iteration Scatter Plots

In order to validate the CHO algorithm, it was compared with the Grey Wolf Optimizer (GWO) algorithm [1], Particle Swarm Optimizer (PSO) algorithm [8], Differential Evolution (DE) algorithm [12], Fast Evolutionary Programming (FEP) algorithm [13] and the Gravitational Search Algorithm (GSA) [14] from literature [1]. The comparison of the algorithms can be seen in Table 4.

From the obtained results, the CHO algorithm obtains the best results in seven of the thirteen problem instances and for the remainder of the instances, and is usually placed with the top three performing algorithms. If the best results obtained over the 30 runs is taken into account, then the CHO algorithm is best performing in eight of the


(a) Iteration 4


(b) Iteration 5


(c) Iteration 6

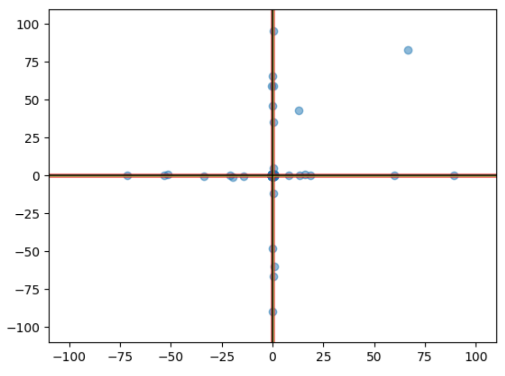
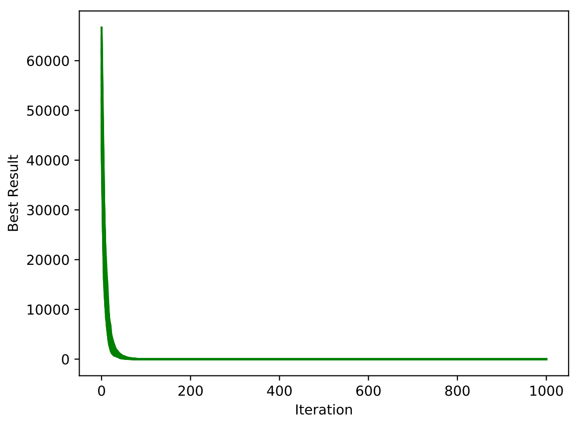

(d) Final iteration

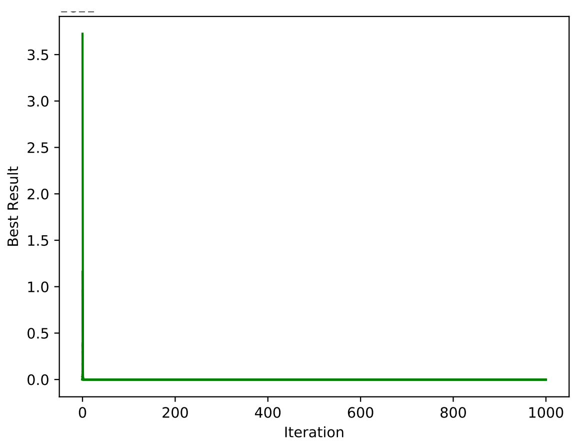
FIGURE 2: Crosshair Iteration Scatter Plots (cont.)

thirteen problem instances. In total, the CHO algorithm is able to find the global best solution on seven of the functions.

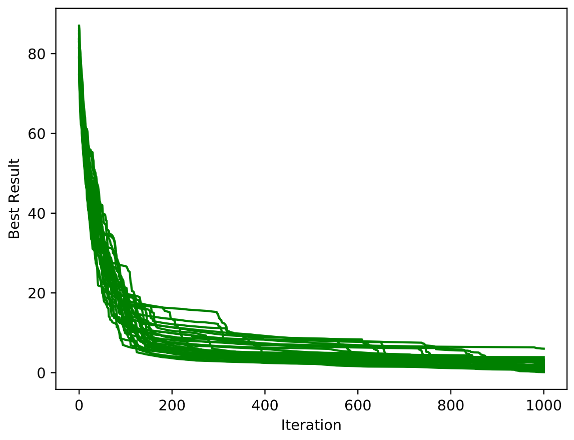
Based on these results, it can be concluded that the developed algorithm using a directed random search is able to compete with established metaheuristics. These results have been published in [2].



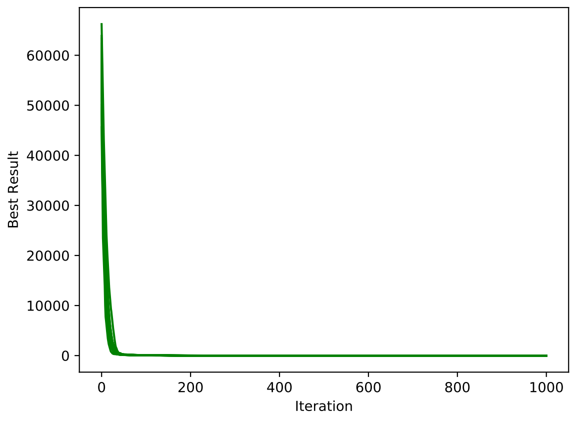
(a) F1 evaluation



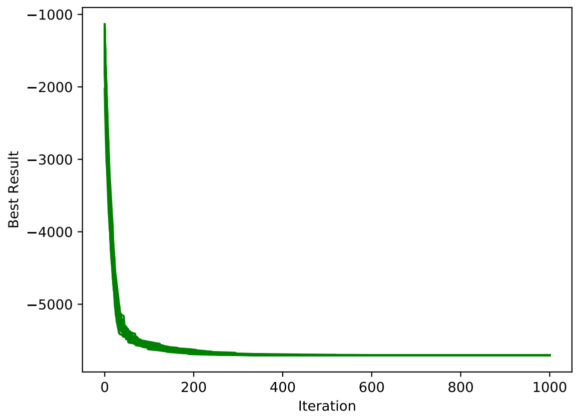
(b) F2 evaluation



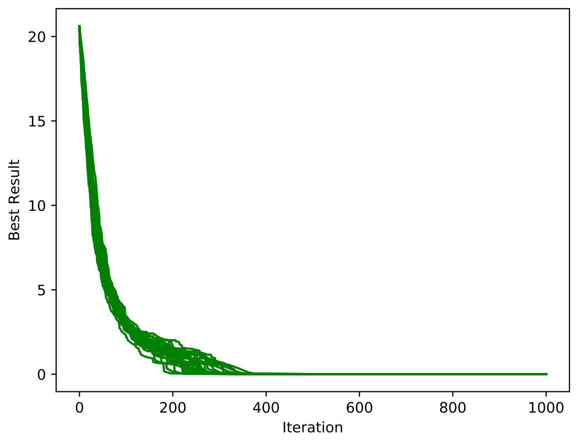
(c) F4 evaluation



(d) F6 evaluation



(e) F8 evaluation



(f) F10 evaluation

FIGURE 3: Crosshair 1000 Iterations for 30 Experiments for Different Functions

TABLE 4: Comparison of Algorithms With Optimal CHO results [2]

F	CHO		GWO		PSO		GSA		DE		FEP	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std	Avg	Std	Avg	Std
F1	**0**	0	6.59E-28	6.34E-05	0.00013	0.0002	2.53E-16	9.67E-17	8.20E-14	5.90E-14	0.00057	0.00013
F2	**0**	0	7.18E-17	0.029	0.04214	0.04542	0.05565	0.19407	1.50E-09	9.90E-10	0.0081	0.00077
F3	**0**	0	3.29E-06	79.1495	70.1256	22.1192	896.534	318.955	6.80E-11	7.4E-11	0.016	0.014
F4	**0**	0	5.61E-07	1.315	1.08648	0.31703	7.35487	1.174145	**0**	0	0.3	0.5
F5	19.9064	11.219	26.8125	69.9049	96.7183	60.1155	67.543	62.2253	**0**	0	5.06	5.87
F6	**0**	0	0.8165	0.0001	0.0001	8.28E-05	2.50E-16	1.74E-16	**0**	0	**0**	0
F7	11.869	0.8339	**0.0022**	0.1002	0.12285	0.04495	0.08944	0.04339	0.00463	0.0012	0.1415	0.3522
F8	-5706.537	2.227E-12	-6123.1	-4087.44	-4841.29	1152.81	-2821.07	493.0375	-11080.1	574.7	**-12554.5**	52.6
F9	**0**	0	0.3105	47.356	46.7042	11.62938	25.9684	7.47006	69.2	383.8	0.046	0.012
F10	**7.11E-15**	0	1.06E-13	0.0778	0.27601	0.50901	0.06208	0.23628	9.70E-08	9.70E-08	0.018	0.0021
F11	5.42E-20	0	0.0044	0.0066	0.00921	0.00772	27.7015	5.04034	**0**	0	0.016	0.022
F12	0.04976	3.095E-17	0.0534	0.0207	0.00691	0.026301	1.79961	0.95114	**7.90E-15**	8.00E-15	9.20E-06	3.60E-06
F13	0.00261	1.032E-18	0.6544	0.0044	0.00667	0.008907	8.89908	7.12624	**5.10E-14**	4.80E-14	0.00016	0.000073

CHAPTER III

HIGH-PERFORMANCE COMPUTING USING POSIX THREADS

In any type of computing, where the amount of time or resources an algorithm or even a single function uses can be scaled up to a significant level, integrating an approach that allows for the use of high-performance computing is an important determining factor in how useful and versatile that particular algorithm or function is. This usually boils down to how well a problem can be split into evenly-sized tasks, how often tasks need to communicate with other tasks, and memory alignment to reduce the amount of unrelated processing time.

Optimization and High-Performance Computing

Optimization problems are no stranger to this same issue. For example, with Particle Swarm Optimization(PSO), the unique part of this function compared to other optimization algorithms is in how it determines each particle's position using a velocity. Every particle has to use a mathematical formula to determine its next position after each iteration based on its current position and a velocity associated with each of its dimensions. This velocity is determined by other particles, but the velocities of one particle do not affect the velocities of other members, and the velocity of one dimension in a particle doesn't affect the velocity of other dimensions within that same particle. Thus, the problem can be split into very small, easily managed tasks, allowing high performance computing approaches to have a significantly better chance at showing a large increase in a high-performance computer's processor utilization, and thus, less time spent processing this part of the algorithm. This is due to a lack of slow-down caused by unevenly sized tasks, which causes larger, more processing-intensive tasks to force other tasks to be delayed until those larger tasks are completed.

As for the communication that needs to occur between the tasks, no communication is necessary, at least during the parts that are unique to PSO. Unfortunately, fitness functions require an entire particle to determine that particle's fitness, meaning these problems are not as easy to split into sub tasks without requiring those tasks to communicate with one another, which can cause a significant amount of wait time between each task's completion. However, this is something that has an effect on all optimization algorithms, and thus what determines whether PSO has an approach that is very capable of using high-performance computing or not is purely based on the parts that are unique to it alone.

As for memory alignment, PSO's velocities rely on each dimension's value, thus, if a high-performance computing approach was also taken with each fitness function, the memory associated with particle's positions (and thus, each dimension value) would be a cause for concern. In this case, the memory alignment required by the fitness function is different from the memory structure required by the velocities, but both are used in determining velocities, thus the number of interrupts when determining a new set of velocities is determined by whichever has more misaligned memory. Due to the fitness functions being less capable of splitting into smaller tasks, the memory structure is likely to have more rigid requirements, and would likely be the cause of more interrupts.

Overall, PSO has an approach that only has very minor issues when it comes to its requirements in high-performance computing, and this makes it very viable as an algorithm that takes full advantage of high-performance computing as well, due to how easily the problem is split into smaller tasks, the lack of communication between tasks, and how variable the memory structure for those tasks can be.

Implementing High-Performance Computing

CHO is similar to PSO when it comes to its requirements for a memory structure. The part that is unique to how CHO determines its particle's dimensions is based on a random number within a set of bounds determined by a part of the best given particle in a population. This means that the dimensions are not dependent on one another, and don't need to communicate with one another. Unfortunately, it also means that it has the same problem as PSO. The only memory structure it uses is the population itself, meaning the memory alignment requirements are based solely on the fitness function's requirements.

Thus, a set of tests was performed, where the problems were split into tasks using `POSIX` Threads (PThreads) in `C`. PThreads were used in place of other methods such as OpenMP and CUDA due to its ease of implementation based on the fact that the original program was written in C, and how much manual control a user has over how many threads are created and when they are destroyed and recreated. In particular, this means that the threads are manually assigned an even distribution of work, and a thread pool is being used that can be created and destroyed at will, allowing for testing of multiple different-sized thread pools in a single run of the executable. GPU type High-Performance Computing methods, such as CUDA, were also considered, but due to this algorithm's heavy reliance on the memory structure of the population, and that population needing to have row-wise memory alignment, CUDA was less likely to see as much speed up as a high-end processor.

ThreadPool Creation

At the time of creation, a thread pool simply needs to allocate space for the number of threads it's going to use (PT), an atomic integer for the task count (tc), an atomic integer for the number of currently running tasks (rt), a task queue (TQ), a mutex lock

(M) and a queue condition (C). Once the memory has been allocated, threads can be created and start running the thread pool's queue function.

Tasks are defined in Algorithm 5.

```
Struct task contains
    int low;
    int high;
    Function f;
end
```

Algorithm 5: Task

Once the pool has been created, the thread pool's queue function is shown in Algorithm 6

```
Data: $TQ, tc, rt, M, C$
while true do
    $task = none$
    MutexLock(M)
    while tc is 0 do
        CondWait(C)
    end
    $task = TQ_0$
    for i from 1 to tc do
        $TQ_{i-1} = TQ_i$
    end
    $tc = tc - 1$
    $rt = rt + 1$
    MutexUnlock(M)
    $task \rightarrow f()$
    $rt = rt - 1$
end
```

Algorithm 6: Thread Pool Queue Function

In Algorithm **??**, the function uses a `while(true)` loop to continue running constantly, a combination of both a mutex lock and mutex condition to avoid using unnecessary processor time, and tc and rt are used to tell the main thread when the thread pool has completed all tasks. Through this uses a `while(true)` loop, a simple way to break a function out of this when using pthreads is to simply send it a task that tells the thread to exit, then use the `join()` function once all the threads have been passed an exit task.

Task Creation

In order to add the tasks to the pool, another relatively simple function is required to add the tasks to the pool and signal the condition, this can be seen in Algorithm 7.

```
Data: $TQ, tc, M, C, task$
MutexLock(M)
$TQ_{tc} = task$
$tc = tc + 1$
MutexUnlock(M)
CondSignal(C)
```

Algorithm 7: Add Tasks To Thread Pool Queue

Tasks need to be created in such a way that each task is given nearly the same amount of data, thus the size of the thread pool ($PSize$) is used to evenly split up the data. In order to align the memory in such a way to avoid interrupts as often as possible, the memory is aligned using rows of the population, that is, each function is given an even number of particles. This process is can be seen in Algorithm 8.

Any function already defined simply needs to be slightly edited to use the low and high of a task, rather than iterating from 0 to D.

Experimentation

During experimentation, it was found that incorporating random number generation into a threaded function while using Mersenne-Twister led to increased completion time over single threaded versions of the same code. Thus, these tests were run allowing CHO's random movement of particles to be single threaded while the fitness functions were passed to the thread pool. However, the amount of time taken by CHO's movement

```
**Data:** $task$
$low = 0$
$rem = DmodPSize$
for i from 0 to PSize do
    $high = low + (D/PSize) - 1$
    if $i < rem$ then
        $high = high + 1$
    end
    $task \rightarrow f = task$
    $task \rightarrow low = low$
    $task \rightarrow high = high$
    Add task to pool
    $low = high + 1$
end
while $tc > 0$ and $rt > 0$ do
    Wait
end
```

Algorithm 8: Split Up Tasks

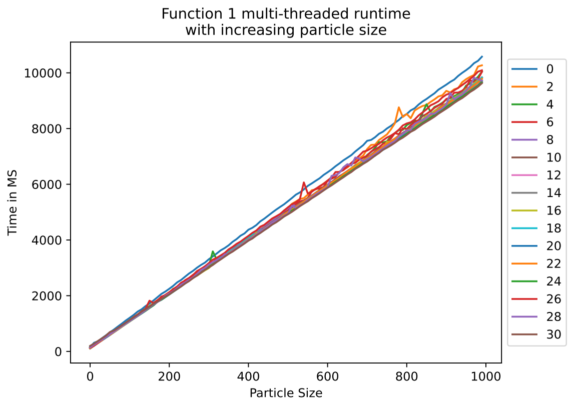
of particles is insignificant enough that the experiments that took the longest were experiments run on complex fitness functions.

The tests in Figures 4 and 5 were run using an Intel i9 10900k at base clock speeds, and 32GB of DDR4 RAM @ 2133 MHz and the following parameters (Table 5):

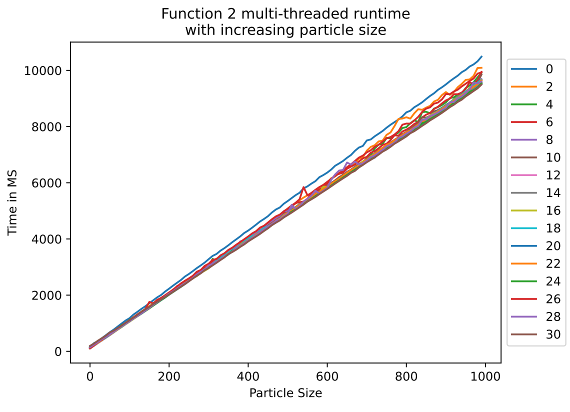
TABLE 5: High-Performance Computing Test Parameters

Parameter	Value
D	300
$iter$	100
tr	30
mer	10

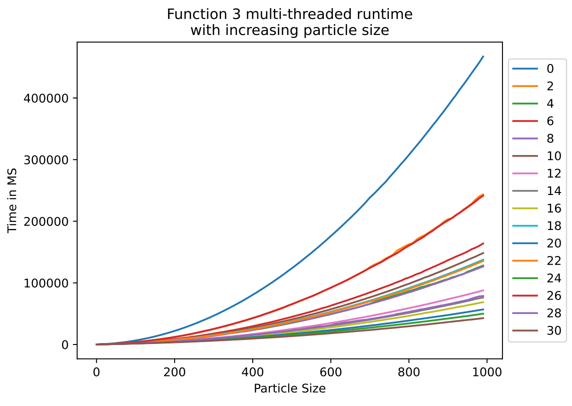
A larger population is used to allow for a larger number of threads, but as this test is looking at run time over results, less iterations are used. These tests used an increasing N value in the range (10-1000) as this causes the algorithm to require increasing computation resources but keeps individual tests from having differences in memory alignment.



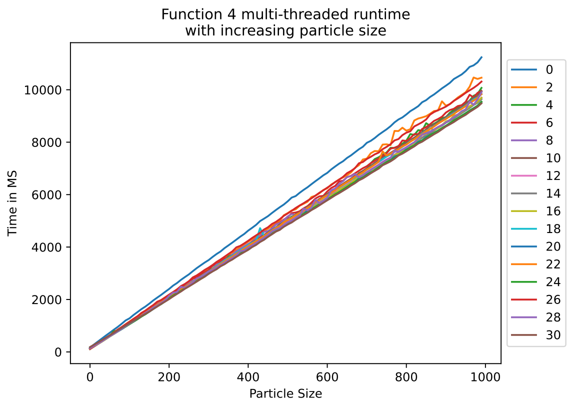
(a) Function 1



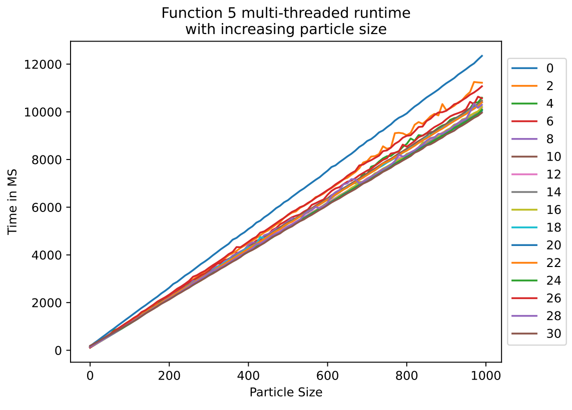
(b) Function 2



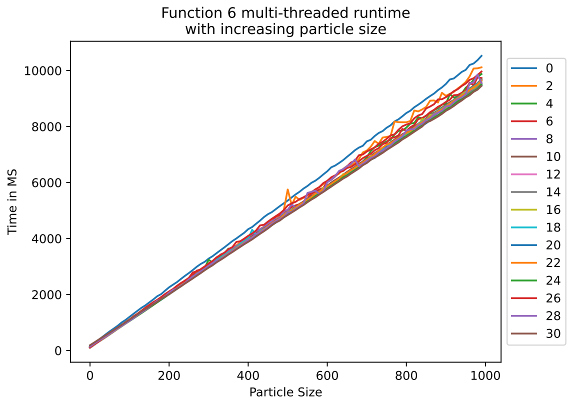
(c) Function 3



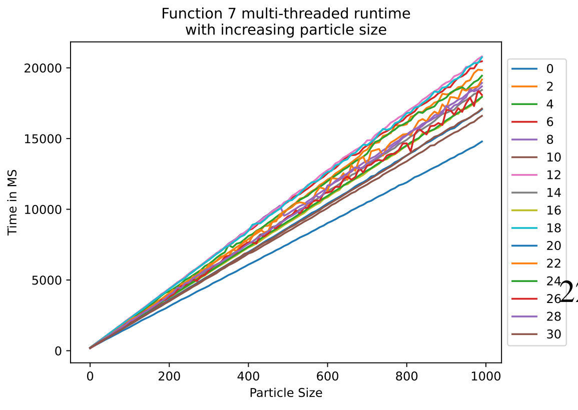
(d) Function 4



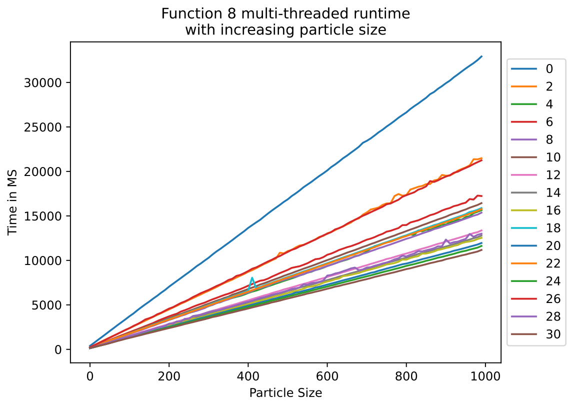
(e) Function 5



(f) Function 6

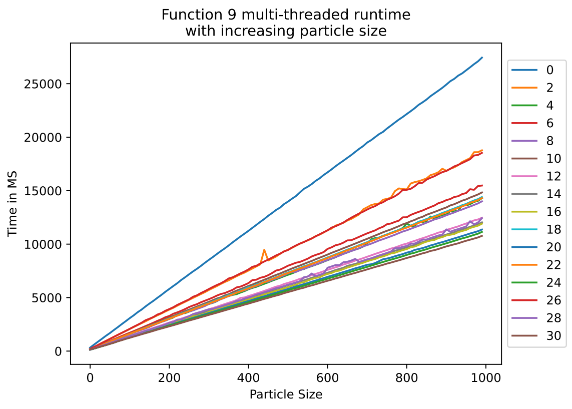


(g) Function 7

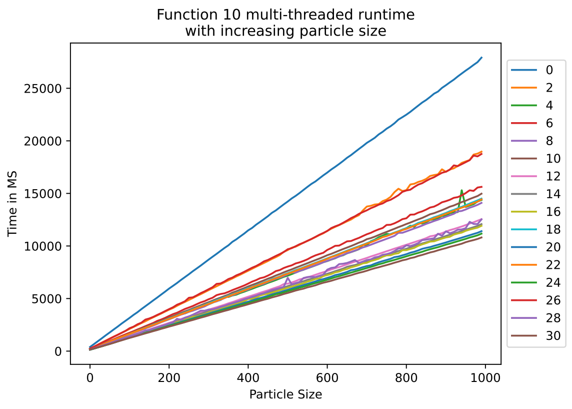


(h) Function 8

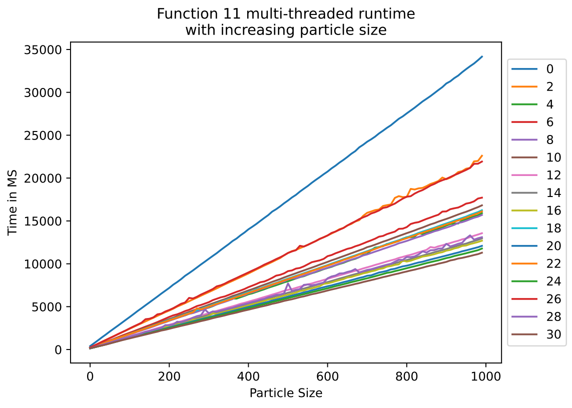
FIGURE 4: Crosshair Iteration Scatter Plots for Function 1 - 8



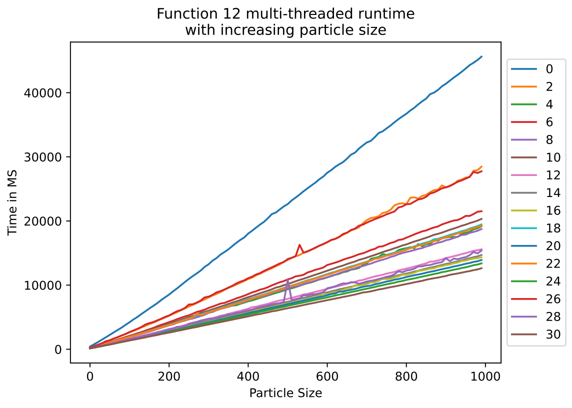
(a) Function 9



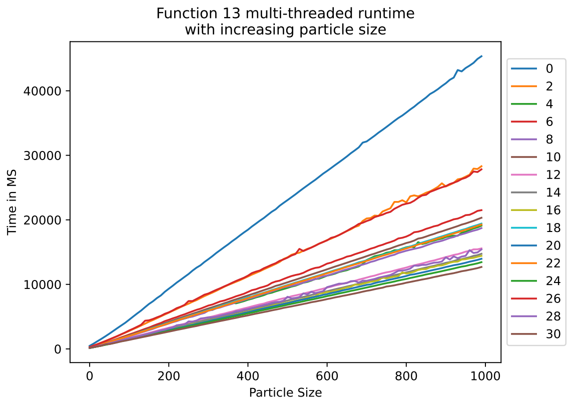
(b) Function 10



(c) Function 11



(d) Function 12



(e) Function 13

FIGURE 5: Crosshair Iteration Scatter Plots for Function 9 - 13 (cont.)

Results and Analysis

Based on comparing the run times of these functions to one another, the results show that the amount of time taken and speedup achieved is directly correlated with the fitness function. With a simple fitness function, such as F1, F2, or F6, very little speedup is seen as the amount of time that particle movement takes is fairly linear. However, with fitness functions that require more computing power such as trigonometric operations, such as F3 and F12, there is significant speedup when using threads.

Despite the threading functions being limited to the fitness function, the algorithm sees a significant speedup when using threads on complex fitness functions. The pseudo-random number generation and small amount of mathematical operations that is associated with the generation of bounds for each dimension appear to take very little time. However, with a different random number generator perhaps different results will be seen in either the run time or fitness of each function.

CHAPTER IV

STOCHASTICITY USING ENSEMBLE CHAOS MAPS

The Crosshair optimizer is heavily reliant on pseudo-random number generators, due to its extensive use of random walks for exploitation. Random number generators (RNG's) have a widespread usage in many different scientific fields to provide basic stochasticity to an underlying system such as cryptography, economic models, statistical simulations, etc. RNG's should address the following key attributes: 1) The generated sequence should not exhibit any statistical weakness. 2) Sub-sequence information should not allow prediction or estimation of predecessors or successor numbers. 3) Knowledge of internal state value (equation), should allow successor sequences to be generated.

Based on these, there are two general classes of RNG's, namely True Random Number Generators (TRNG's) and Pseudo-Random Number Generators (PRNG's). True number generators are generally based on real-world occurring phenomena of physical processes including radioactive decay, various thermal and atmospheric noise etc. TRNG suffer from slowness of implementation, high cost and dependency on hardware [15].

Pseudo-Random Number Generators on the other hand produce sequences which are deterministic, generally based on mathematical formation with long cycle times and initialized via a random initial (seed) value. In theory, these deterministic sequences are predictable with a certain time complexity, but by increasing this complexity, the oscillating sequence period can be extended.

Recently a new class of RNG's has appeared based on Chaos theory. Chaos theory was first introduced by Edward Lorenz in 1963 [16] and has gained a number of applications over the past 50 years especially in dynamical systems [17]. Chaos maps exhibit three important facets; ergodicity, sensitivity to initial conditions and random behavior. These features make them an interesting system for generating random sequences.

This section discusses the application of different chaos maps as Chaos Random Number Generators (CRNG's) embedded in the Crosshair optimizer to generate stochasticity.

Chaos Systems

The term chaos is used to describe the complex behavior of simple dynamical systems. When observed overall, this behavior appears to be erratic and somewhat random, however, these systems are inherently deterministic with the precise behavior carefully mapped. The aperiodic non-repeating behavior of chaotic systems makes chaotic sequences a prime candidate for generating pseudo-random sequences. Overall, four general branches of chaotic systems exist; 1) dissipative systems, 2) fractals, 3) dissipative and high-dimensional systems, 4) conservative systems [15]. The systems utilized in this thesis are the discrete dissipative systems.

Chaos as Pseudo-Random Number Generators

The development of a CRNG system is based on the application of the dual nature of chaos, deterministic in microscopic space formulated through its mathematical equations, and random in macroscopic space. A mathematical foundation exists which shows how such chaos system overcome the significant issues with traditional random number systems, such as its reliance on the assumed randomness of a physical process, inability to analyze and optimize the random number generator, inability to compute probabilities and entropy of the random number generator, and inconclusiveness of statistical tests. Subsequently, comprehensive research has been done on applying chaos maps as random number generators. The connection between chaotic systems and random number generators has been given by [18]. A strong linkage has been shown between the Lehmer generator [19] and the simple chaos dynamical system of Bernoulli shift [20]. Furthermore, [21] showed the hidden periodicity of chaos system and its dependence on numerical systems.

A family of enhanced CPRNG's has been developed by [22], where a very long series of pseudo-random numbers have been generated, accomplished through the ultra weak coupling of chaotic systems, such as the Tent Map, which is enhanced in order to conceal the chaotic genuine function [23]. Recently, the very notion of using CPRNG's in EA's has been explored by [24]. Some other recent examples of chaos used as random number generators include [17], [25], [26], [27], [28], [29], [30], [31] and [32] amongst others.

Discrete Dissipative Chaos Maps

Discrete dissipative chaotic maps are considered the most interesting chaotic systems, which are based on a linear set of equations. By encompassing a fine grain approach over the solution space, these set of equations can be easily formulated. All these attributes allows a unique parsing period of chaotic oscillation. A total of eight unique chaotic systems were utilized in this research. The equations and operating parameters can be obtained from [33].

Arnold's Cat Chaos Map

The Arnold's Cat Map is a two dimensional torus discrete chaotic map, whose equations are given in (4.1). The parameter of $k = 2.0$.

$$X_{n+1} = X_n + Y_n \cdot (\mathrm{mod}1)$$
$$Y_{n+1} = X_n + k \cdot Y_n \cdot (\mathrm{mod}1) \tag{4.1}$$

<u>Burgers Chaos Map</u>

The Burgers Mapping (Figure 6) is a discretization of a pair of coupled differential equations, used by Burgers [34] to illustrate the relevance of the concept of bifurcation to the study of hydrodynamic flows. Equations (4.2) and (4.3) give the system equations.

$$X_{n+1} = aX_n - Y_n^2 \tag{4.2}$$

$$Y_{n+1} = bY_n + X_nY_n \tag{4.3}$$

Where, a = 0.75 and b = 1.75 and the initial conditions being X_0 = -0.1 and Y_0 = 0.1. The different plots including frequency of integer and real values can be seen in Figure 7, whereas Figure 8 gives the plots of *x* and *y* values.

<u>Delayed Logistic Chaos Map</u>

The Delayed Logistic Map (Figure 9) is a dissipative map with a smooth invariant circle having a strange attractor due to interspersed parameter intervals [35]. Its equations are given in Equations (4.4) and (4.5).

$$X_{n+1} = AX_n\left(1 - Y_n\right) \tag{4.4}$$

$$Y_{n+1} = X_n \tag{4.5}$$

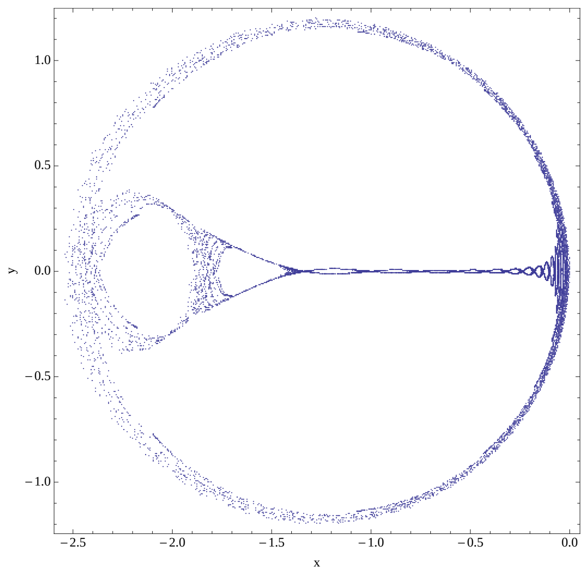


FIGURE 6: Burgers Chaos Map

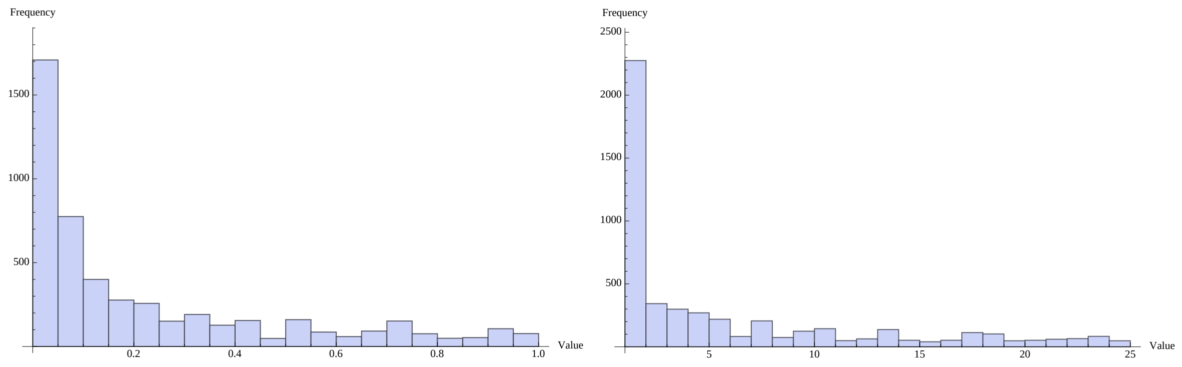


FIGURE 7: Real and Integer Number Histograms by Burgers Chaos Map

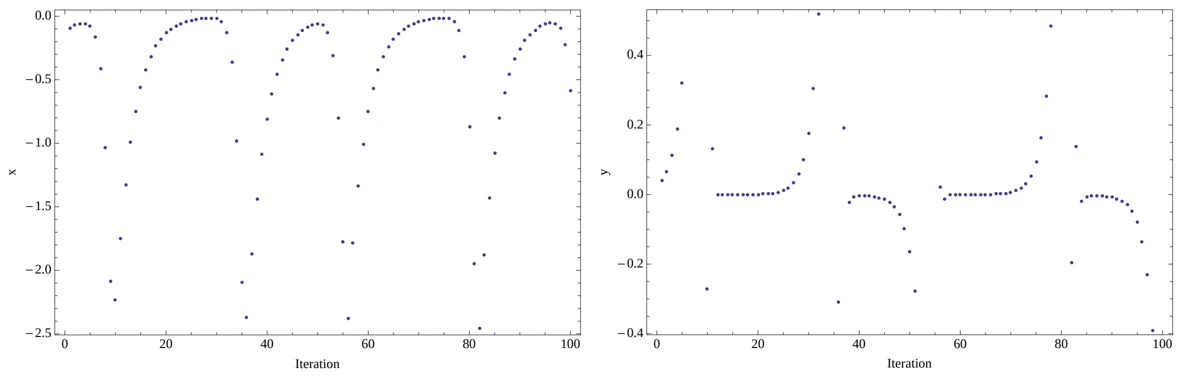


FIGURE 8: x and y Values Iteration Plots for Burgers Chaos Map

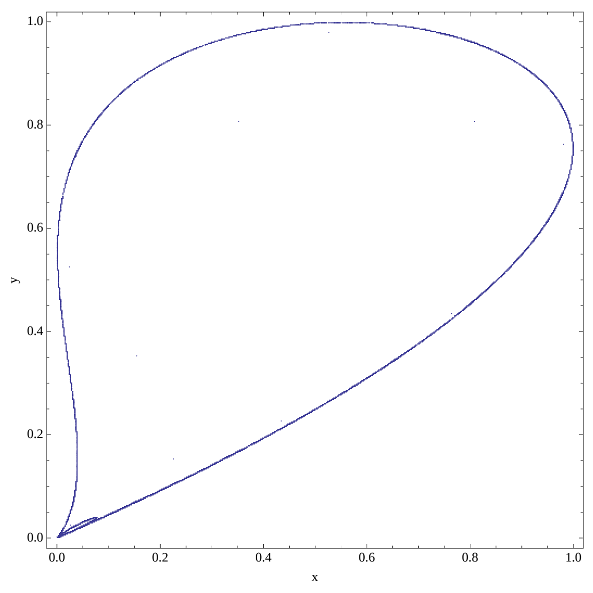


FIGURE 9: Delayed Logistic Chaos Map

Where $A = 2.27$ with initial conditions being $X_0 = 0.001$ and $Y_0 = 0.001$. The different plots including frequency of integer and real values can be seen in Figure 10, whereas Figure 11 gives the plots of *x* and *y* values.

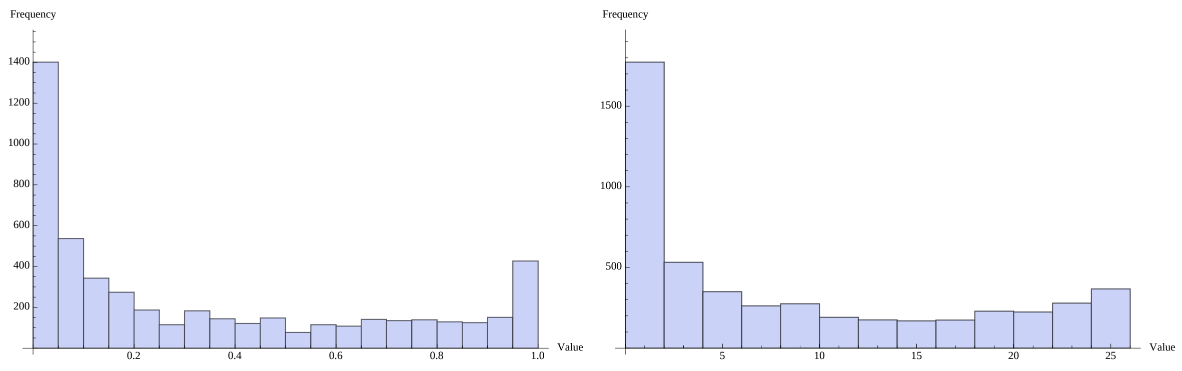


FIGURE 10: Real and Integer Number Histograms by Delayed Logistic Chaos Map

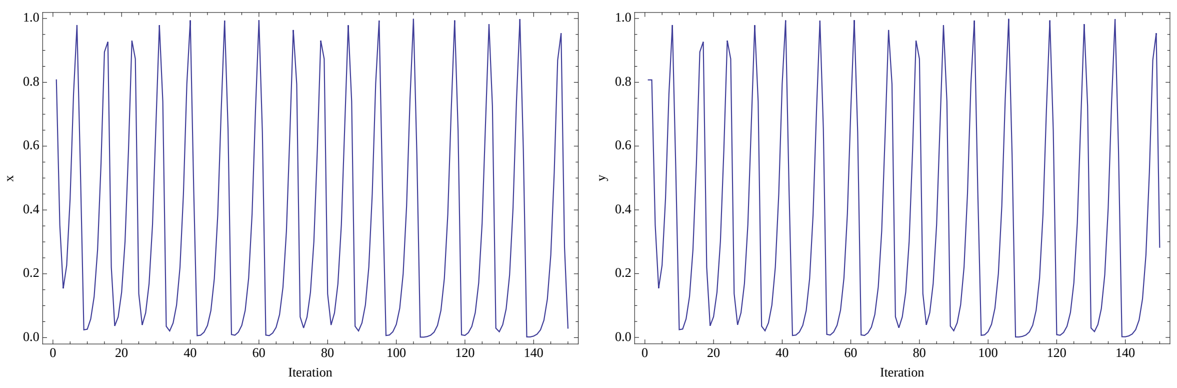


FIGURE 11: *x* and *y* Values Iteration Plots for Delayed Logistic Chaos Map

<u>Dissipative Standard Chaos Map</u>

The Dissipative Standard Map is a two-dimensional chaotic system, with its equations given in (4.6).

$$\begin{aligned} X_{n+1} &= X_n + Y_{n-1} \cdot (\mathrm{mod} 2\pi) \\ Y_{n+1} &= (\beta \cdot Y_n) + (k \cdot \sin X_n \, (\mathrm{mod} 2\pi)) \end{aligned} \tag{4.6}$$

Where β = 0.1 and k = 8.8.

<u>Henon Chaos Map</u>

The Henon Map is a discrete-time simplified Poincare Map dynamical system. The equation is given in (4.7).

$$\begin{aligned} X_{n+1} &= \alpha - X_n^2 + (\beta \cdot Y_n) \\ Y_{n+1} &= X_n \end{aligned} \tag{4.7}$$

Where α = 1.4 and β = 0.3.

Ikeda Chaos Map

The Ikeda map is a discrete-time dynamical system formulated on a light model going around across a nonlinear optical resonator. The 2D equation is given in Equation (4.8).

$$\begin{aligned} X_{n+1} &= \gamma + \mu \cdot ((X_n \cdot \cos\phi) - (Y_n \cdot \sin\phi)) \\ Y_{n+1} &= \mu \cdot ((X_n \cdot \sin\phi) + (Y_n \cdot \cos\phi)) \\ \phi &= \beta - \tfrac{\alpha}{(1+X_n^2+Y_n^2)} \end{aligned} \tag{4.8}$$

Where α = 0.75, β = 1.75, γ = 1 and μ = 0.9.

Lozi Chaos Map

The Lozi Map (Figure 12) is a two-dimensional piece-wise linear map closely related to the Henon Map.

The equations of this map are given in Equations (4.9) and (4.10).

$$X_{n+1} = 1 - a\,|X_n| + bY_n \tag{4.9}$$

$$Y_{n+1} = X_n \tag{4.10}$$

Where $a = 1.7$ and $b = 0.5$ [33] with X_0 = -0.1 and Y_0 = 0.1 as the initial conditions.

The different plots including frequency of integer and real values can be seen in Figure 13, whereas Figure 14 gives the plots of *x* and *y* values. The presented figures of each chaotic map are referenced from [36].

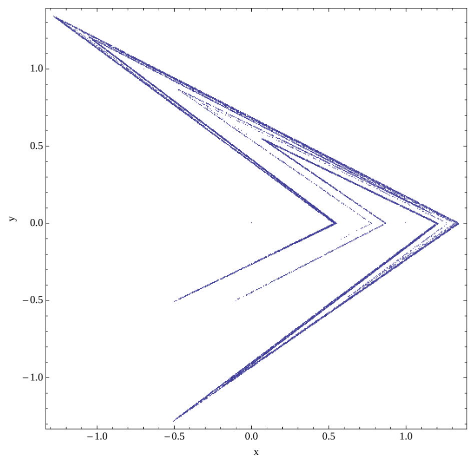


FIGURE 12: Lozi Chaos Map

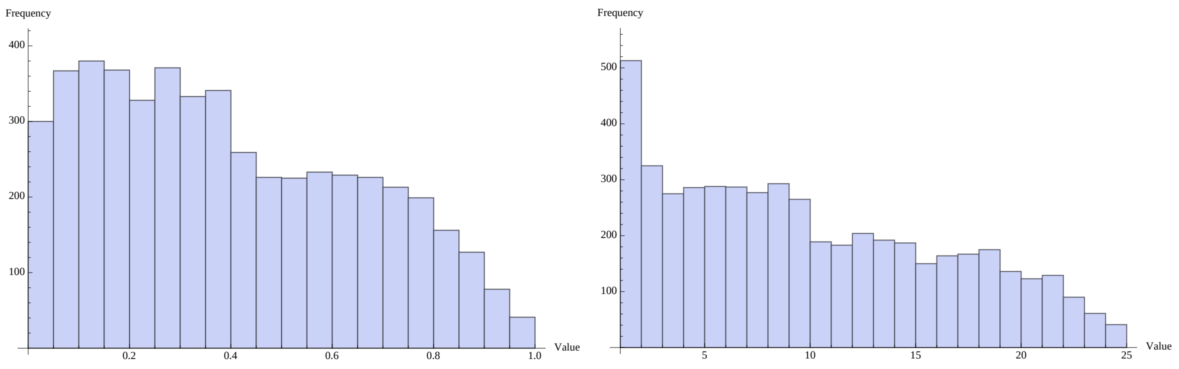


FIGURE 13: Real and Integer Number Histograms by Lozi Chaos Map

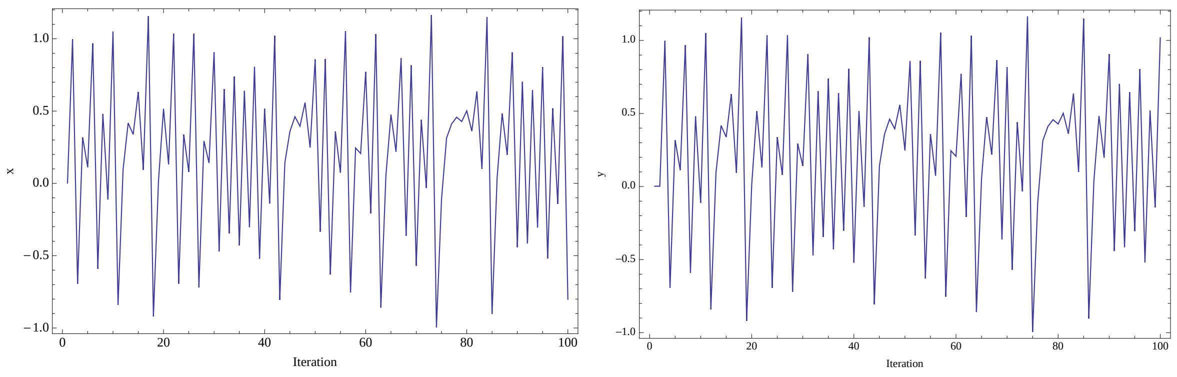


FIGURE 14: x and y Values Iteration Plots for Lozi Chaos Map

Sinai Chaos Map

The Sinai Map is similar to the Arnold's Cat Map in having a simple two-dimensional discrete system. The equation of the map is given in (4.11).

$$\begin{aligned} X_{n+1} &= X_n + Y_n + (\delta \cdot \cos 2\pi \cdot Y_n \cdot (\mathrm{mod} 1)) \\ Y_{n+1} &= X_n + 2 \cdot Y_n \cdot (\mathrm{mod} 1) \end{aligned} \tag{4.11}$$

Where δ = 0.1.

Experimentation

Using chaos maps, tests were run to find an average result, standard deviation, best result, and time in milliseconds (msec) for each of the 13 objective functions. The goal of these tests is to see if using a different form of pseudo-random number generation will lead to either better results or similar results being found in less time. To test for each randomizer's effectiveness on both factors, two tests were run: **Experiment A** with a smaller population size and less overall iterations (Parameters shown in Table 6), and **Experiment B** with a larger population size and more iterations (Parameters shown in Table 7). The short test would prioritize randomizers that are capable of finding optimal results quicker than others, while the larger test allows for the possibility of a randomizer that may produce better results that others given additional usage.

These tests were run using the following changes to the population and iteration parameters:

TABLE 6: Experiment A Parameters

Parameter	Value
D	100
$iter$	30
tr	30
N	30
mer	100

TABLE 7: Experiment B Parameters

Parameter	Value
D	100
$iter$	100
tr	30
N	30
mer	500

Table 8 shows the map name abbreviations.

TABLE 8: Map Name Abbreviations

ARN	BUR	DEL	DIS	HEN	IKE	LOZ	MER	SIN
Arnold's Cat	Burger	Delayed Logistic	Dissapative	Henon	Ikeda	Lozi	Mersenee Twister	Sinai

Results and Analysis

In analyzing the results, the most impactful differences between the randomizers came in two different forms: how optimal results were, and how the amount of time each randomizer added to the overall runtime of the algorithm. Thus, the results will be broken into two sections: *Comparison of Results*, and *Comparison of Time*.

Comparison of Results

Tables 9 and 10 show all results found by Experiment A, while Table 11 and 12 shows all results found by Experiment B. Additionally, Tables 13 and 14 show a direct comparison of each randomizer's Experiment A results, and Tables 15 and 16 show a direct comparison of the randomizer's Experiment B results.

The experiments appear to show some of the same results, where Arnold's Cat Map, Henon Map, Mersenne Twister, and Sinai Map are capable of finding the best results the fastest, but Dissapative Map manages to generate significantly better results in Experiment B than it did in Experiment A. However, given the additional resources, Mersenne Twister has a greater number of fitness functions with the best found averages, and Arnold's Cat Map finds more Best values.

However, overall, it appears that each map is capable of finding optimal results with different fitness functions, such as Delayed Logistic Map, which finds the most optimal averages out of the randomizers for Function 7. This suggests that if a different fitness function were given to this algorithm, an approach that uses more than one random number generator would likely lead to the most optimal outcome.

Overall, Arnold's Cat Map, Henon Map, Mersenne Twister, Sinai Map, and Dissipative Map are the most capable randomizers between the tests, but other randomizers should be considered when using CHO.

TABLE 9: Experiment A Results By Randomizer

F	Avg	Std	Best
F1	0	0	0
F2	3.5333E-29	3.34E-30	3.00E-29
F3	0	0	0
F4	3.72163E-29	4.65551E-30	2.8321E-29
F5	16.28589017	0.002346576	15.82159547
F6	2.77334E-32	0	2.77334E-32
F7	11.62489593	0.591599922	7.013377106
F8	-5706.53701	1.81899E-12	-5706.53701
F9	0	0	0
F10	7.10543E-15	0	7.10543E-15
F11	0	0	0
F12	0.049765827	0	0.049765827
F13	0.002614683	8.67362E-19	0.002614683

(a) Experiment A: ARN Results

F	Avg	Std	Best
F1	0.043661512	0.000557799	0.043296347
F2	13.90106659	0.015181459	13.88247318
F3	255488.505	1.1642E-10	255488.505
F4	7.504859433	4.44089E-15	6.58653868
F5	26.46321272	1.06581E-14	26.46321272
F6	1213.198361	3.709341626	1208.008648
F7	369.9892887	0.850859571	366.8989633
F8	-2043.53336	9.09495E-13	-2043.53336
F9	30.80030413	0.013363865	30.78861776
F10	6.79569402	2.66454E-15	6.79569402
F11	12.47480977	3.55271E-15	12.47480977
F12	0.930433512	0.002342741	0.928776944
F13	6.424783374	1.77636E-15	6.424783374

(b) Experiment A: BUR Results

F	Avg	Std	Best
F1	0.00754897	0.002470168	0.004085801
F2	0.00160229	0	0.00160229
F3	0.013275417	6.93889E-18	0.013275417
F4	0.00962206	3.46945E-18	0.00962206
F5	1199.901017	947.4738992	226.8987997
F6	6.79345E-06	5.0822E-21	6.79345E-06
F7	8.289090212	0.610570948	6.953521788
F8	-2676.05903	4.54747E-13	-2771.17783
F9	1.57469E-05	1.01634E-06	1.39053E-05
F10	0.001195601	2.87535E-05	0.001178262
F11	1.32215E-05	0	1.32215E-05
F12	0.049765864	2.08167E-17	0.049765864
F13	0.002616446	1.03397E-07	0.002615889

(c) Experiment A: DEL Results

F	Avg	Std	Best
F1	0	0	0
F2	3.75627E-29	2.938E-30	3.16248E-29
F3	7.41071E-10	0	7.41071E-10
F4	4.24864E-29	5.98721E-30	3.09068E-29
F5	20.33336368	7.10543E-15	20.33336368
F6	3.08149E-32	0	3.08149E-32
F7	12.66311041	1.008185086	8.998286264
F8	-5706.53701	2.72848E-12	-5706.53701
F9	0	0	0
F10	7.10543E-15	0	7.10543E-15
F11	0.00739604	0	0.00739604
F12	0.049765827	1.38778E-17	0.049765827
F13	0.002614683	0	0.002614683

(d) Experiment A: DIS Results

F	Avg	Std	Best
F1	0	0	0
F2	3.26377E-29	3.19786E-30	2.87307E-29
F3	5.56406E-27	2.5E-39	5.56406E-27
F4	3.10762E-29	4.21918E-30	2.27044E-29
F5	10.61670083	0.002862199	10.61593587
F6	2.15704E-32	0	2.15704E-32
F7	16.86029191	0.327342002	12.38971283
F8	-5706.53701	1.81899E-12	-5706.53701
F9	13.9294268	8.88178E-15	13.9294268
F10	7.10543E-15	0	7.10543E-15
F11	0.029552184	0	0.029552184
F12	0.049765827	1.38778E-17	0.049765827
F13	0.002614683	0	0.002614683

(e) Experiment A: HEN Results

F	Avg	Std	Best
F1	0	0	0
F2	1.87269E-28	4.44732E-29	1.42147E-28
F3	1.50968E-06	1.00008E-07	1.35691E-06
F4	0.040345991	5.80888E-05	0.040216101
F5	27.25384659	1.06581E-14	27.25384659
F6	6.47112E-32	0	6.47112E-32
F7	9.727677739	0.521279142	7.538286859
F8	-5706.53701	9.09495E-13	-5706.53701
F9	2.4567E-12	0	6.75016E-14
F10	3.2685E-13	0	5.32907E-14
F11	0.012316073	0	0.012316073
F12	0.049765827	2.77556E-17	0.049765827
F13	0.002614683	0	0.002614683

(f) Experiment A: IKE Results

TABLE 10: Experiment A Results By Randomizer (cont.)

F	Avg	Std	Best
F1	0	0	0
F2	1.72416E-28	3.17706E-29	1.24773E-28
F3	2.992123707	1.33227E-15	2.992123707
F4	15.24070809	5.32907E-15	12.4239081
F5	26.46207232	0.00178413	26.45954918
F6	9.86076E-32	0	9.86076E-32
F7	1164.186661	0.255294442	29.14785382
F8	-5706.53701	1.81899E-12	-5706.53701
F9	29.84877171	1.42109E-14	29.84877171
F10	6.903162044	3.55271E-15	6.903162044
F11	1.47993E-13	0	6.32827E-15
F12	0.049765827	1.38778E-17	0.049765827
F13	0.002614683	1.30104E-18	0.002614683

(a) Experiment A: LOZ Results

F	Avg	Std	Best
F1	0	0	0
F2	3.60209E-29	3.36074E-30	2.80967E-29
F3	9.76369E-35	0	9.76369E-35
F4	3.72061E-29	4.42044E-30	2.95805E-29
F5	18.14559995	7.10543E-15	15.94578266
F6	5.23853E-32	0	5.23853E-32
F7	12.45609533	0.61919679	8.984489629
F8	-5706.53701	1.81899E-12	-5706.53701
F9	0	0	0
F10	7.10543E-15	0	7.10543E-15
F11	0	0	0
F12	0.049765827	3.46945E-17	0.049765827
F13	0.002614683	4.33681E-19	0.002614683

(b) Experiment A: MER Results

F	Avg	Std	Best
F1	0	0	0
F2	3.73065E-29	3.10779E-30	3.00926E-29
F3	5.68042E-25	3.88E-38	5.68042E-25
F4	3.54613E-29	3.82918E-30	2.59017E-29
F5	82.98181026	2.242710657	77.67640634
F6	4.31408E-32	0	4.31408E-32
F7	13.41413424	0.634312442	9.121128541
F8	-5706.53701	1.81899E-12	-5706.53701
F9	0	0	0
F10	7.10543E-15	0	7.10543E-15
F11	0	0	0
F12	0.049765827	3.46945E-17	0.049765827
F13	0.002614683	1.30104E-18	0.002614683

(c) Experiment A: SIN Results

TABLE 11: Experiment B Results By Randomizer

F	Avg	Std	Best
F1	0	0	0
F2	0	0	0
F3	0	0	0
F4	0	0	0
F5	16.31245035	12.76711193	0.418957607
F6	3.69779E-32	0	3.69779E-32
F7	11.51422017	1.484698159	5.931027461
F8	-5706.53701	1.81899E-12	-5706.53701
F9	0	0	0
F10	7.10543E-15	0	3.55271E-15
F11	0.014779777	0	0.014779777
F12	0.049765827	0	0.049765827
F13	0.002614683	2.1684E-18	0.002614683

(a) Experiment B: ARN Results

F	Avg	Std	Best
F1	8.59125E-05	1.35525E-20	8.59125E-05
F2	14.62790667	1.77636E-15	14.62790667
F3	230561.0101	5.82077E-11	219818.4793
F4	7.406678261	4.44089E-15	7.406678261
F5	17.44935919	0.007563008	17.44080652
F6	1166.449969	0	1166.449969
F7	295.4342747	0.66144008	292.7307356
F8	-2043.53934	1.13687E-12	-2043.53934
F9	29.97729816	3.55271E-15	29.97729816
F10	6.5319389	8.88178E-16	6.5319389
F11	12.25398247	5.32907E-15	12.25398247
F12	0.700034548	5.55112E-16	0.700034548
F13	4.32815148	2.66454E-15	4.32815148

(b) Experiment B: BUR Results

F	Avg	Std	Best
F1	0.071465612	0.057843158	0.00033918
F2	9.35796E-05	1.35525E-20	9.35796E-05
F3	0.000360451	0	0.000360451
F4	0.000514712	0	0.000514712
F5	1264.636792	1105.377562	156.6247558
F6	1.97921E-07	5.29396E-23	1.97921E-07
F7	8.07844198	0.520894088	6.601984015
F8	-2447.50761	1.36424E-12	-3010.64868
F9	6.63492E-08	0	6.63492E-08
F10	5.98282E-05	0	5.98282E-05
F11	0.009857767	0	0.009857767
F12	0.049765827	3.46945E-17	0.049765827
F13	0.002614711	1.30104E-18	0.002614711

(c) Experiment B: DEL Results

F	Avg	Std	Best
F1	0	0	0
F2	0	0	0
F3	0	0	0
F4	0	0	0
F5	7.385283085	0.001176125	6.853206371
F6	2.15704E-32	0	2.15704E-32
F7	11.74431875	0.726820682	7.672571816
F8	-5706.53701	1.81899E-12	-5706.53701
F9	0	0	0
F10	7.10543E-15	0	7.10543E-15
F11	0.009857285	0	0.009857285
F12	0.049765827	2.77556E-17	0.049765827
F13	0.002614683	4.33681E-19	0.002614683

(d) Experiment B: DIS Results

F	Avg	Std	Best
F1	0	0	0
F2	0	0	0
F3	0	0	0
F4	0	0	0
F5	6.478994561	4.44089E-15	6.478994561
F6	4.31408E-32	0	4.31408E-32
F7	23.79414554	0.318756631	11.97234063
F8	-5706.53701	1.81899E-12	-5706.53701
F9	9.949590571	5.32907E-15	9.949590571
F10	7.10543E-15	0	7.10543E-15
F11	0	0	0
F12	0.049765827	6.93889E-18	0.049765827
F13	0.002614683	4.33681E-19	0.002614683

(e) Experiment B: HEN Results

F	Avg	Std	Best
F1	0	0	0
F2	0	0	0
F3	1.82749E-09	8.27181E-25	1.82749E-09
F4	0.002552123	0	0.002552123
F5	23.63073341	1.06581E-14	23.63073341
F6	8.62817E-32	0	8.62817E-32
F7	9.003789636	0.462546708	6.963820757
F8	-5706.53701	2.72848E-12	-5706.53701
F9	7.73603E-12	0	6.03961E-14
F10	7.81597E-13	0	3.90799E-14
F11	0.00739604	0	0.00739604
F12	0.049765827	2.08167E-17	0.049765827
F13	0.002614683	8.67362E-19	0.002614683

(f) Experiment B: IKE Results

TABLE 12: Experiment B Results By Randomizer (cont.)

F	Avg	Std	Best
F1	0	0	0
F2	0	0	0
F3	1.82749E-09	8.27181E-25	1.82749E-09
F4	0.002552123	0	0.002552123
F5	23.63073341	1.06581E-14	23.63073341
F6	8.62817E-32	0	8.62817E-32
F7	9.003789636	0.462546708	6.963820757
F8	-5706.53701	2.72848E-12	-5706.53701
F9	7.73603E-12	0	6.03961E-14
F10	7.81597E-13	0	3.90799E-14
F11	0.00739604	0	0.00739604
F12	0.049765827	2.08167E-17	0.049765827
F13	0.002614683	8.67362E-19	0.002614683

(a) Experiment B: LOZ Results

F	Avg	Std	Best
F1	0	0	0
F2	0	0	0
F3	0	0	0
F4	0	0	0
F5	3.499472469	1.801783042	1.700578903
F6	2.77334E-32	0	2.77334E-32
F7	11.62003778	0.618996041	7.665922397
F8	-5706.53701	1.81899E-12	-5706.53701
F9	0	0	0
F10	7.10543E-15	0	7.10543E-15
F11	0	0	0
F12	0.049765827	0	0.049765827
F13	0.002614683	2.1684E-18	0.002614683

(b) Experiment B: MER Results

F	Avg	Std	Best
F1	0	0	0
F2	0	0	0
F3	0	0	0
F4	0	0	0
F5	8.996824819	0.544740063	0.28386915
F6	4.00593E-32	0	4.00593E-32
F7	11.40174776	0.699454391	8.316231718
F8	-5706.53701	1.81899E-12	-5706.53701
F9	0	0	0
F10	7.10543E-15	0	7.10543E-15
F11	0.00739604	0	0.00739604
F12	0.049765827	2.08167E-17	0.049765827
F13	0.002614683	8.67362E-19	0.002614683

(c) Experiment B: SIN Results

TABLE 13: Experiment A: Average Result by Function

F	ARN	BUR	DEL	DIS	HEN	IKE	LOZ	MER	SIN
F1	**0**	0.043661512	0.007548975	**0**	**0**	**0**	**0**	**0**	**0**
F2	3.5333E-29	13.90106659	0.001602293	3.75627E-29	**3.26377E-29**	1.87269E-28	1.72416E-28	3.60209E-29	3.73065E-29
F3	**0**	255488.5050	0.013275417	7.41071E-10	5.56406E-27	1.50968E-06	2.992123707	9.76369E-35	5.68042E-25
F4	3.72163E-29	7.504859433	0.009622062	4.24864E-29	**3.10762E-29**	0.040345991	15.24070809	3.72061E-29	3.54613E-29
F5	16.28589017	26.46321272	1199.901017	20.33336368	**10.61670083**	27.25384659	26.46207232	18.14559995	82.98181026
F6	2.77334E-32	1213.198361	6.79345E-06	3.08149E-32	**2.15704E-32**	6.47112E-32	9.86076E-32	5.23853E-32	4.31408E-32
F7	11.62489593	369.9892887	**8.289090212**	12.66311041	16.86029191	9.727677739	1164.186661	12.45609533	13.41413424
F8	**-5706.53701**	-2043.53336	-2676.05903	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**
F9	**0**	30.80030413	1.57469E-05	**0**	13.92942683	2.4567E-12	29.84877171	**0**	**0**
F10	**7.10543E-15**	6.79569402	0.001195601	**7.10543E-15**	**7.10543E-15**	3.2685E-13	6.903162044	**7.10543E-15**	**7.10543E-15**
F11	**0**	12.47480977	1.32215E-05	0.00739604	0.029552184	0.012316073	1.47993E-13	**0**	**0**
F12	**0.049765827**	0.930433512	**0.049765864**	**0.049765827**	**0.049765827**	**0.049765827**	0.049765827	**0.049765827**	**0.049765827**
F13	**0.002614683**	6.424783374	**0.002616446**	**0.002614683**	**0.002614683**	**0.002614683**	0.002614683	**0.002614683**	**0.002614683**
Total	8	0	3	5	**9**	4	4	7	7
Unique	0	0	1	0	**4**	0	0	0	0

TABLE 14: Experiment A: Best Result by Function

F	ARN	BUR	DEL	DIS	HEN	IKE	LOZ	MER	SIN
F1	**0**	0.043296347	0.004085801	**0**	**0**	**0**	**0**	**0**	**0**
F2	3.00E-29	13.88247318	0.00160229	3.16248E-29	2.87307E-29	1.42147E-28	1.24773E-28	**2.80967E-29**	3.00926E-29
F3	**0**	255488.505	0.013275417	7.41071E-10	5.56406E-27	1.35691E-06	2.992123707	9.76369E-35	5.68042E-25
F4	2.8321E-29	6.58653868	0.00962206	3.09068E-29	**2.27044E-29**	0.040216101	12.4239081	2.95805E-29	2.59017E-29
F5	15.82159547	26.46321272	226.8987997	20.33336368	**10.61593587**	27.25384659	26.45954918	15.94578266	77.67640634
F6	2.77334E-32	1208.008648	6.79345E-06	3.08149E-32	**2.15704E-32**	6.47112E-32	9.86076E-32	5.23853E-32	4.31408E-32
F7	7.013377106	366.8989633	**6.953521788**	8.998286264	12.38971283	7.538286859	29.14785382	8.984489629	9.121128541
F8	**-5706.53701**	-2043.53336	-2771.17783	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**
F9	**0**	30.78861776	1.39053E-05	**0**	13.9294268	6.75016E-14	29.84877171	**0**	**0**
F10	**7.10543E-15**	6.79569402	0.001178262	**7.10543E-15**	**7.10543E-15**	5.32907E-14	6.903162044	**7.10543E-15**	**7.10543E-15**
F11	**0**	12.47480977	1.32215E-05	0.00739604	0.029552184	0.012316073	6.32827E-15	**0**	**0**
F12	**0.049765827**	0.928776944	0.049765864	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**
F13	**0.002614683**	6.424783374	0.002615889	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**
Total	**8**	0	1	6	**8**	4	4	**8**	7
Unique	1	0	1	0	**3**	0	0	1	0

TABLE 15: Experiment B: Average Result by Function

F	ARN	BUR	DEL	DIS	HEN	IKE	LOZ	MER	SIN
F1	**0**	8.59125E-05	0.071465612	**0**	**0**	**0**	**0**	**0**	**0**
F2	**0**	14.62790667	9.35796E-05	**0**	**0**	**0**	**0**	**0**	**0**
F3	**0**	230561.0101	0.000360451	**0**	**0**	1.82749E-09	0.069003187	**0**	**0**
F4	**0**	7.406678261	0.000514712	**0**	**0**	0.002552123	10.33869006	**0**	**0**
F5	16.31245035	17.44935919	1264.636792	7.385283085	6.478994561	23.63073341	17.5514933	**3.499472469**	8.996824819
F6	3.69779E-32	1166.449969	1.97921E-07	**2.15704E-32**	4.31408E-32	8.62817E-32	6.16298E-32	2.77334E-32	4.00593E-32
F7	11.51422017	295.4342747	**8.07844198**	11.74431875	23.79414554	9.003789636	3063.698934	11.62003778	11.40174776
F8	**-5706.53701**	-2043.53934	-2447.50761	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**
F9	**0**	29.97729816	6.63492E-08	**0**	9.949590571	7.73603E-12	29.84877171	**0**	**0**
F10	**7.10543E-15**	6.5319389	5.98282E-05	**7.10543E-15**	**7.10543E-15**	7.81597E-13	9.790103839	**7.10543E-15**	**7.10543E-15**
F11	0.014779777	12.25398247	0.009857767	0.009857285	**0**	0.00739604	0.00739604	**0**	0.00739604
F12	**0.049765827**	0.700034548	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**
F13	**0.002614683**	4.32815148	**0.002614711**	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**
Total	9	0	3	10	9	5	5	**11**	9
Unique	0	0	**1**	**1**	0	0	0	**1**	0

TABLE 16: Experiment B: Best Result by Function

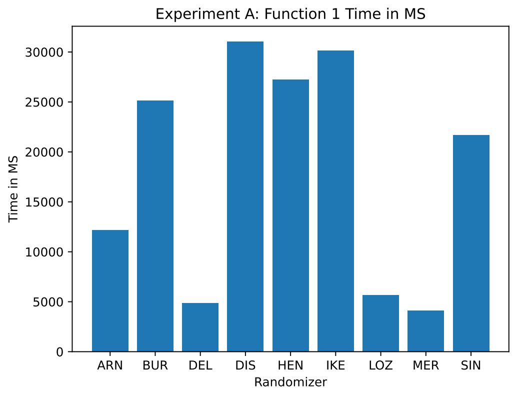
F	ARN	BUR	DEL	DIS	HEN	IKE	LOZ	MER	SIN
F1	**0**	8.59125E-05	0.00033918	**0**	**0**	**0**	**0**	**0**	**0**
F2	**0**	14.62790667	9.35796E-05	**0**	**0**	**0**	**0**	**0**	**0**
F3	**0**	219818.4793	0.000360451	**0**	**0**	1.82749E-09	0.069003187	**0**	**0**
F4	**0**	7.406678261	0.000514712	**0**	**0**	0.002552123	0.953430675	**0**	**0**
F5	0.418957607	17.44080652	156.6247558	6.853206371	6.478994561	23.63073341	17.54900026	1.700578903	**0.28386915**
F6	3.69779E-32	1166.449969	1.97921E-07	**2.15704E-32**	4.31408E-32	8.62817E-32	6.16298E-32	2.77334E-32	4.00593E-32
F7	**5.931027461**	292.7307356	6.601984015	7.672571816	11.97234063	6.963820757	24.0715619	7.665922397	8.316231718
F8	**-5706.537013**	-2043.53934	-3010.64868	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**
F9	**0**	29.97729816	6.63492E-08	**0**	9.949590571	6.03961E-14	29.84877171	**0**	**0**
F10	**3.55271E-15**	6.531938986	5.98282E-05	7.10543E-15	7.10543E-15	3.90799E-14	7.269229834	7.10543E-15	7.10543E-15
F11	0.014779777	12.25398247	0.009857767	0.009857285	**0**	0.00739604	0.00739604	**0**	0.00739604
F12	**0.049765827**	0.700034548	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**
F13	**0.002614683**	4.328151485	**0.002614711**	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**
Total	**10**	0	2	9	8	5	5	9	9
Unique	**2**	0	0	1	0	0	0	0	1

Comparison of Time

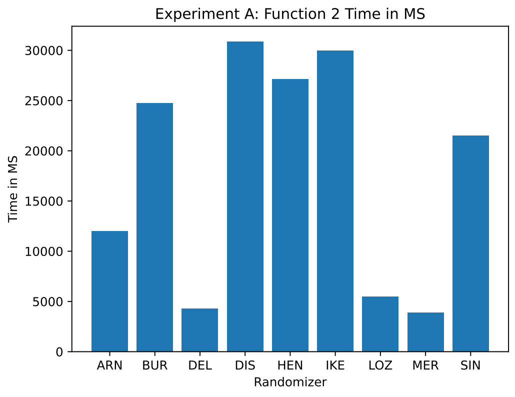
Figures 15 and 16 graphically show the time difference between the randomizers by function for Experiment A, while Figures 17 and 18 show the same for Experiment B. Additionally, Table 17 shows a time comparison of Experiment A numerically and Table 18 shows a time comparison of Experiment B numerically.

As can be seen in these figures and tables, the difference in fitness function accounts for very little difference in the placing of which randomizer completes these tests the fastest. In fact, at first glance, the tests appear so similar to one another that it's as if each randomizer has a flat time cost. However, when comparing Experiment A's results to one another, or experiment B's results to one another, they're extremely similar in shape, and thus time. But, when comparing the shape of the Experiment A's results to Experiment B's results, we see similar, yet slightly different shapes.

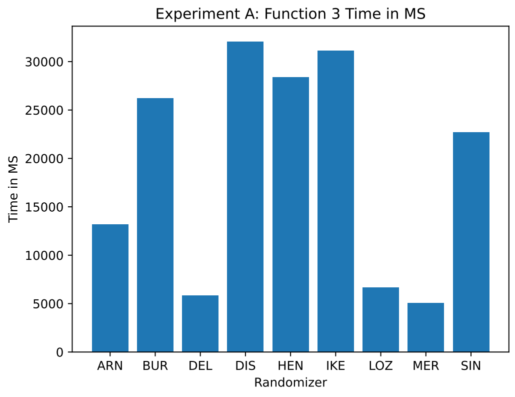
However, there's only two actions these randomizers have that could be associated with a time cost: initialization and generation of values. Simply put, the difference in time between Experiment A and Experiment B means that the randomizer's cost to generate values is different than its initialization cost, and thus the time growth is different between the randomizers. This may elevate other randomizers over the Henon Map, as Henon map's use cost may outweigh its potential to have optimum output.



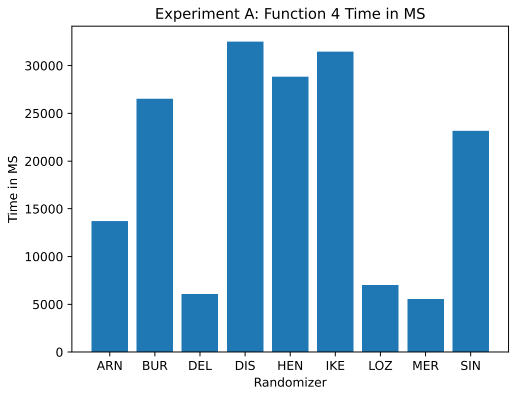
(a) Exp A: Function 1



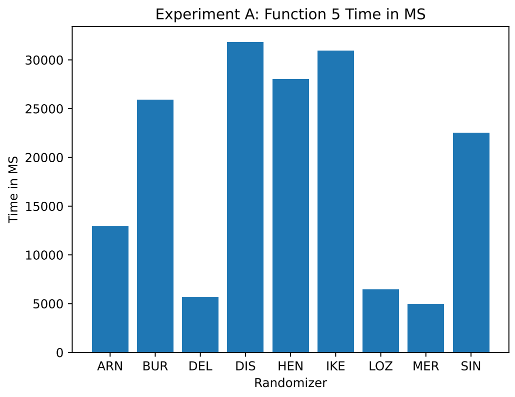
(b) Exp A: Function 2



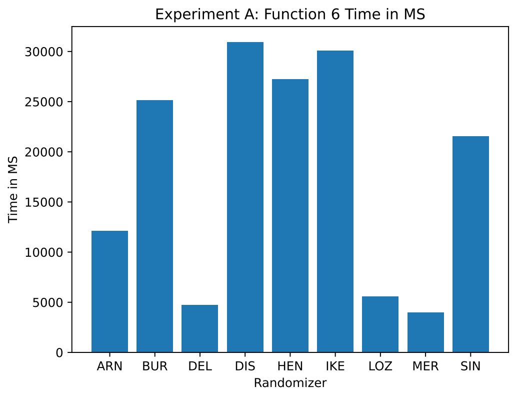
(c) Exp A: Function 3



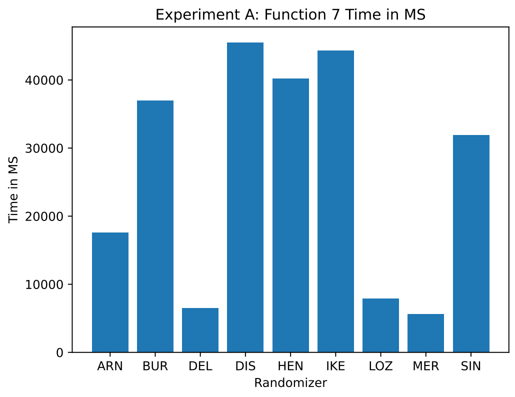
(d) Exp A: Function 4



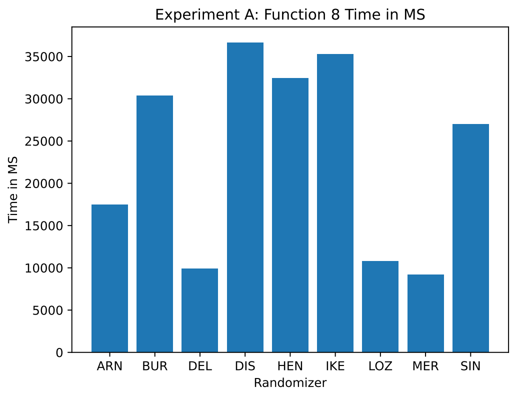
(e) Exp A: Function 5



(f) Exp A: Function 6

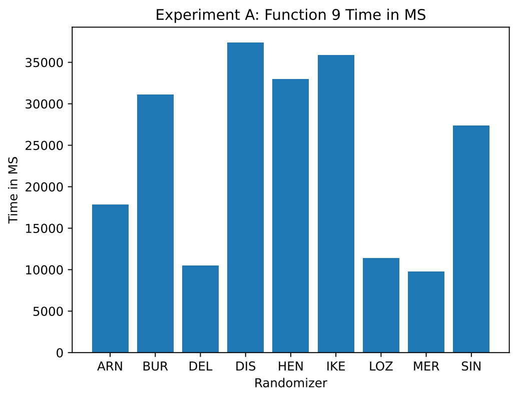


(g) Exp A: Function 7

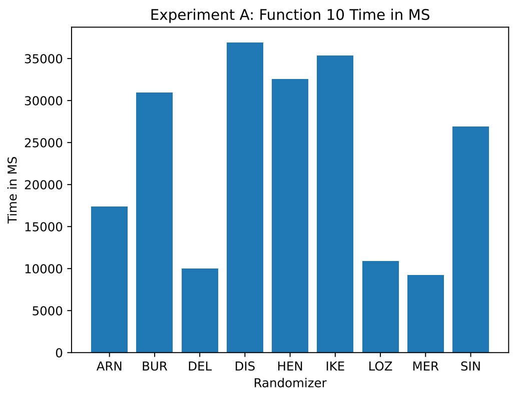


(h) Exp A: Function 8

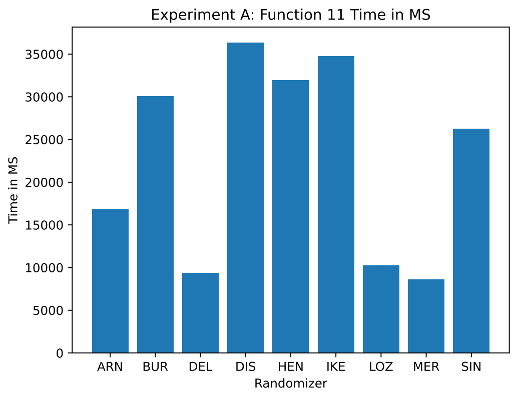
FIGURE 15: Experiment A Time Comparisons for Function 1 - 8



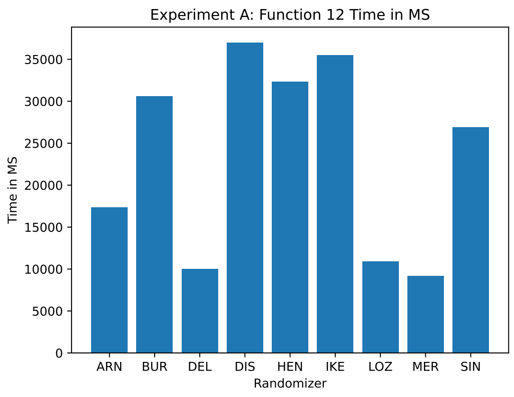
(a) Exp A: Function 9 Time



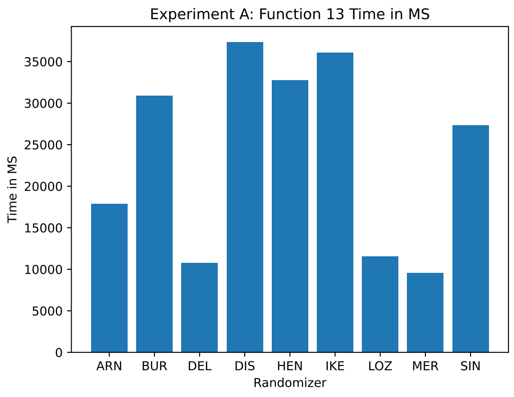
(b) Exp A: Function 10 Time



(c) Exp A: Function 11 Time

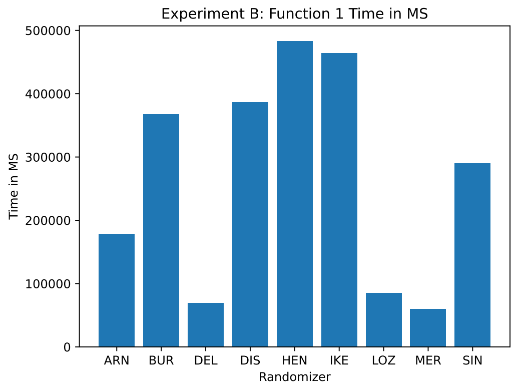


(d) Exp A: Function 12 Time

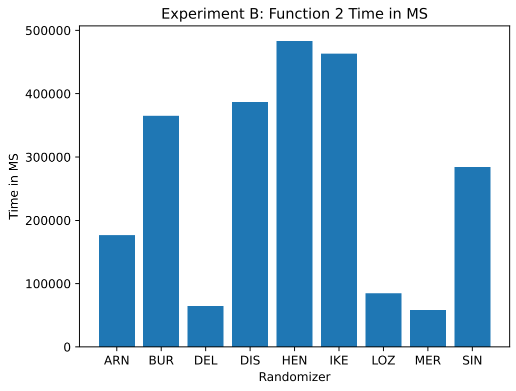


(e) Exp A: Function 13 Time

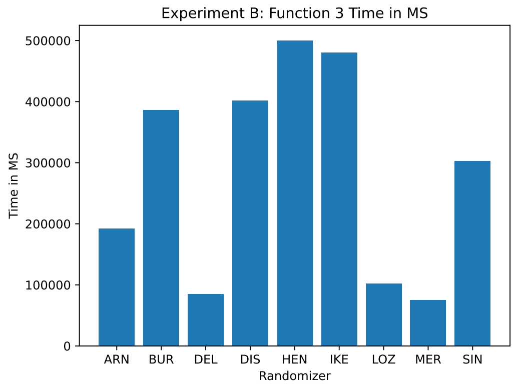
FIGURE 16: Experiment A Comparisons for Function 9 - 13 (cont.)



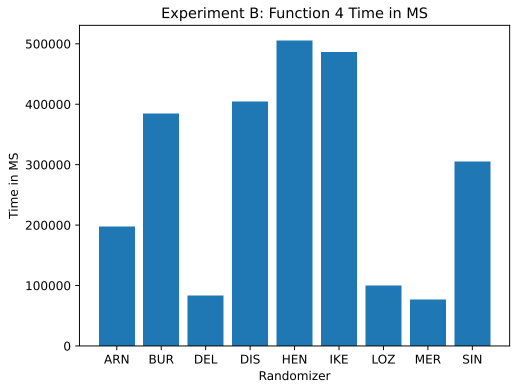
(a) Exp B: Function 1



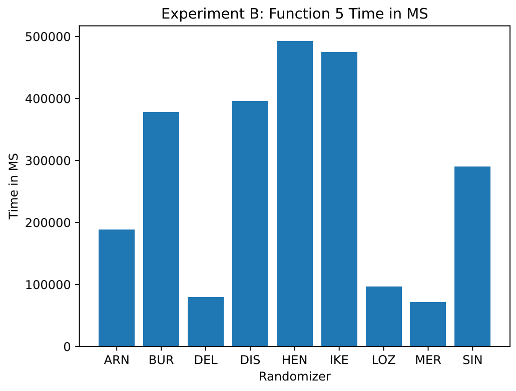
(b) Exp B: Function 2



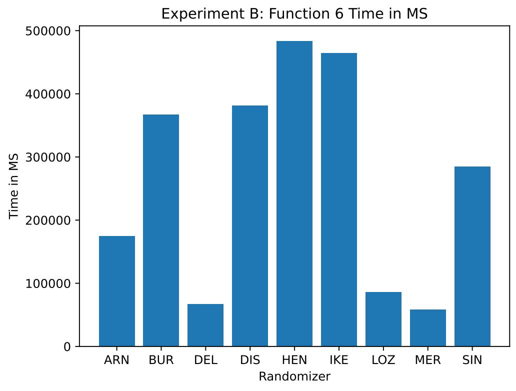
(c) Exp B: Function 3



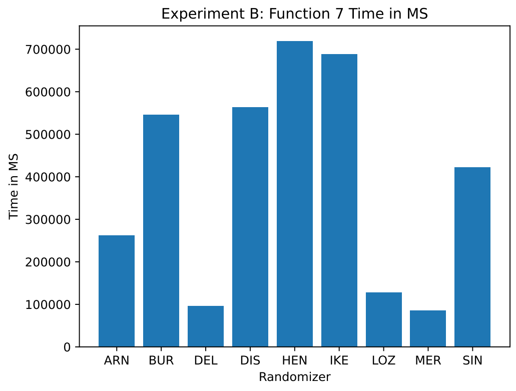
(d) Exp B: Function 4



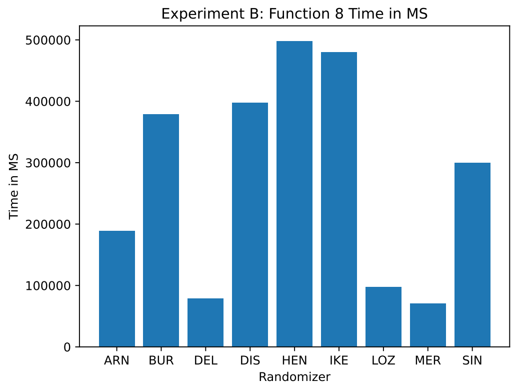
(e) Exp B: Function 5



(f) Exp B: Function 6

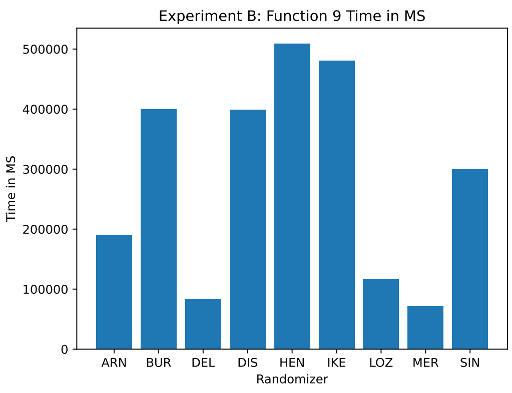


(g) Exp B: Function 7

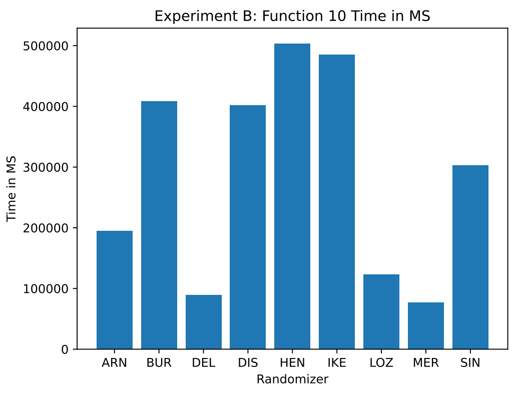


(h) Exp B: Function 8

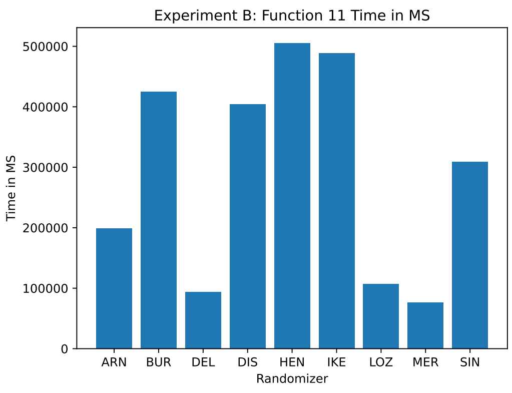
FIGURE 17: Experiment B Time Comparisons for Function 1 - 8



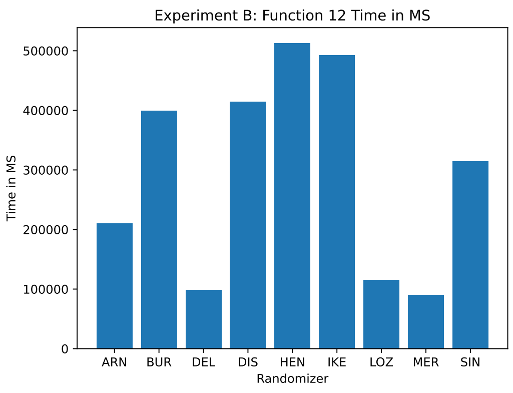
(a) Exp B: Function 9 Time



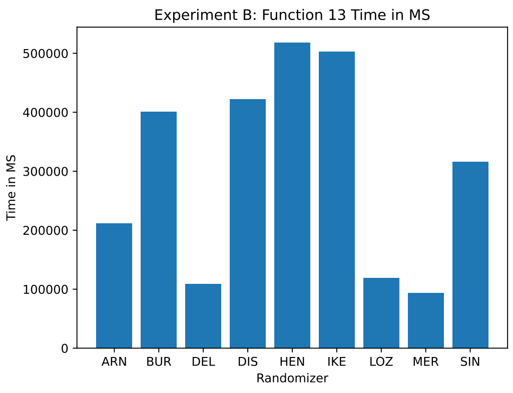
(b) Exp B: Function 10 Time



(c) Exp B: Function 11 Time



(d) Exp B: Function 12 Time



(e) Exp B: Function 13 Time

FIGURE 18: Experiment B Time Comparisons for Function 9 - 13 (cont.)

TABLE 17: Experiment A: Time in MSec by Function

F	ARN	BUR	DEL	DIS	HEN	IKE	LOZ	MER	SIN
F1	12157	25163	4872	31027	27261	30132	5685	**4097**	21683
F2	11998	24729	4304	30855	27133	29986	5503	**3891**	21508
F3	13190	26230	5834	32059	28359	31146	6674	**5076**	22682
F4	13688	26548	6092	32506	28842	31466	7003	**5537**	23178
F5	12967	25923	5683	31824	27985	30933	6456	**4940**	22527
F6	12085	25126	4692	30938	27218	30055	5582	**3979**	21543
F7	17563	36987	6512	45514	40223	44311	7900	**5590**	31878
F8	17458	30352	9890	36664	32431	35277	10794	**9180**	27019
F9	17855	31132	10473	37369	33003	35841	11374	**9741**	27355
F10	17384	30966	9982	36889	32522	35357	10876	**9232**	26921
F11	16772	30076	9335	36345	31900	34748	10236	**8597**	26264
F12	17309	30561	10026	36988	32304	35488	10903	**9138**	26895
F13	17878	30910	10779	37358	32758	36042	11534	**9576**	27359

TABLE 18: Experiment B: Time in MSec by Function

F	ARN	BUR	DEL	DIS	HEN	IKE	LOZ	MER	SIN
F1	177988	367357	69093	386403	483007	464149	85351	**60006**	289937
F2	175858	365218	64414	385974	482957	463132	84565	**58263**	283278
F3	192211	385592	84558	401688	499928	480096	101654	**75074**	302097
F4	197419	384572	83406	403790	505294	485946	99744	**76440**	305190
F5	187993	377932	79473	395203	492478	474933	96028	**71430**	290086
F6	174942	366622	66728	381533	483452	464378	85724	**58491**	284569
F7	261932	546150	96462	563816	718977	688162	127174	**85228**	421477
F8	188708	378590	79085	397310	497953	479643	97632	**70523**	299291
F9	190338	399670	83890	398661	509343	480446	117443	**71998**	299956
F10	195284	408764	89390	401723	503626	485440	123442	**77035**	302844
F11	198923	424712	93520	404305	505591	488558	106924	**76458**	309174
F12	210476	399227	98362	414979	513118	493066	115520	**90104**	315042
F13	211915	400860	108843	422427	518417	502551	119014	**93563**	316116

CHAPTER V

CHO ADJUSTMENT PARAMETER TUNING

In experimenting with different versions of pseudo-random number generation, what those numbers are being used for and what variables impact their usage also need to be adjusted in order to optimize the algorithm. The variable that has the largest impact on how random numbers interact with the results would be the adj value. This value determines whether or not a value is within the bounds set by f_{adj}, c_{adj}, and n_{adj}, and is what gives CHO its name. Therefore, this value needs to be evaluated in order to find what adj value best fits for each pseudo-random number generator for each fitness function.

Experimentation

In order to find an optimal value using Mersenne Twister, a number of experiments were run using different adj values, and almost all fitness functions require a value greater than 0.9. This is due to how this value interacts with the algorithm, the closer the value gets to 1.0 the more exploitation is happening between iterations, however, other randomizers may not need as much focus on exploitation to generate optimal results. Thus, every randomizer was experimented against an adj value in the range (0.81-0.99) using the following parameters as given in Table 19.

Results and Analysis

The condensed results are given in Table 20, which shows each randomizer's best average out of every experiment, and Table 21 which shows each randomizer's best found value out of all experiments. Additionally, Figures 19 and 20 show how every algorithm's

TABLE 19: Tuning Parameters

Parameter	Value
D	100
$iter$	100
tr	30
N	30
mer	500

average value responded to different adj values, and Figures 21 and 22 show how every algorithm's best value responded to different adj values.

As anticipated, each randomizer required a different adj value in order to obtain its most optimal result. An interesting thing to note, as well, is that not only did each randomizer have different points where the average fitness became increasingly more optimal, but each randomizer also had a different point where the adj value became too high, such as during the experimentation on Function 4. Each randomizer also converged on a adj value that was similar to others, meaning that the fitness function itself has the strongest effect on what that adj value should be, but the order in which each randomizer converged was also almost always the same, where Burger's Map tended to find it's most optimal results at lower adj values, while the Delayed Logisitic Map tended to require the highest values.

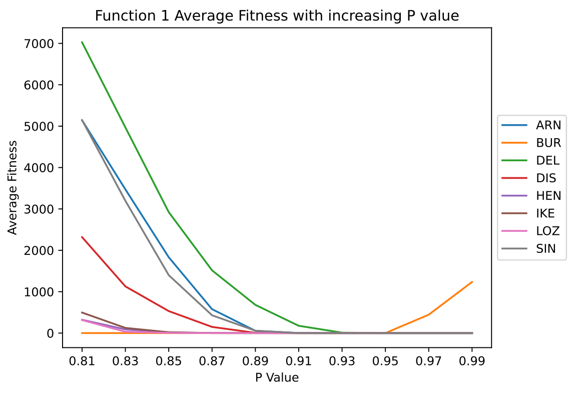
Interestingly, Mersenne Twister still manages to obtain the most optimal average result of all other randomizers, but other randomizer's are not far behind. Dissapative, Henon, and Sinai Maps are also very capable of finding optimal results, and Burger's Map was even capable of finding a best value better than what was found by Mersenne Twister on Function 7.

TABLE 20: Average with Optimal Parameters by Function

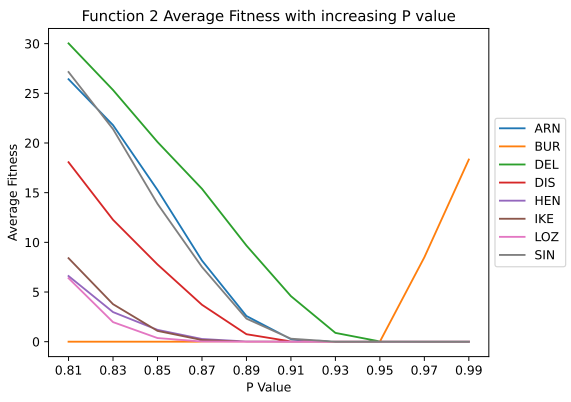
F	ARN	BUR	DEL	DIS	HEN	IKE	LOZ	MER	SIN
F1	**0**	3.84806E-08	4.30856E-08	**0**	**0**	**0**	**0**	**0**	**0**
F2	**0**	8.69481E-05	7.26257E-05	**0**	**0**	**0**	**0**	**0**	**0**
F3	**0**	2.05337E-05	2.4816E-05	**0**	**0**	1.21496E-10	1.56565E-07	**0**	**0**
F4	**0**	0.000815111	0.000600738	**0**	**0**	0.001426417	0.001022348	**0**	**0**
F5	1.5995E-08	18.59933908	12.02149283	0.000327398	0.000487669	14.4470888	17.97270476	**9.01585E-10**	**9.01585E-10**
F6	2.46519E-32	6.1238E-08	3.2904E-08	1.54074E-32	3.38964E-32	3.08149E-33	4.00593E-32	**0**	1.84889E-32
F7	10.96035422	**5.325513151**	8.139588979	10.37451319	13.05853361	9.016382655	9.662135583	**5.325513151**	11.7254749
F8	**-5706.53701**	-5706.53286	-5109.04922	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**
F9	**0**	2.88073E-08	2.44344E-08	**0**	**0**	7.89768E-12	1.46176E-11	**0**	**0**
F10	**7.10543E-15**	4.91076E-05	4.08633E-05	**7.10543E-15**	**7.10543E-15**	9.27258E-13	6.10001E-12	**7.10543E-15**	**7.10543E-15**
F11	0.012316073	1.96423E-07	5.27804E-08	**0**	**0**	6.40821E-13	5.29354E-13	**0**	**0**
F12	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**
F13	**0.002614683**	**0.002614685**	**0.002614684**	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**
Total	9	3	2	10	10	5	5	**13**	11
Unique	0	0	0	0	0	0	0	**1**	0

TABLE 21: Best Found with Optimal Parameters by Function

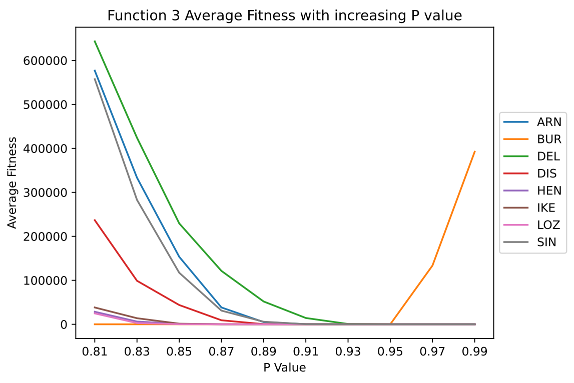
F	ARN	BUR	DEL	DIS	HEN	IKE	LOZ	MER	SIN
F1	**0**	3.84806E-08	4.30856E-08	**0**	**0**	**0**	**0**	**0**	**0**
F2	**0**	8.69481E-05	7.26257E-05	**0**	**0**	**0**	**0**	**0**	**0**
F3	**0**	2.05337E-05	2.4816E-05	**0**	**0**	1.21496E-10	1.56565E-07	**0**	**0**
F4	**0**	0.000815111	0.000600738	**0**	**0**	0.001426417	0.000978551	**0**	**0**
F5	1.5995E-08	18.59208552	9.068203711	0.000327398	0.000487669	14.44701179	17.97270476	**9.01585E-10**	**9.01585E-10**
F6	2.46519E-32	6.1238E-08	3.2904E-08	1.54074E-32	3.38964E-32	**0**	4.00593E-32	**0**	1.84889E-32
F7	6.098760117	**0.732129944**	6.540945463	8.096372464	10.80091633	6.825751383	7.001641856	5.325513151	8.080487363
F8	-5706.53701	**-5706.53286**	-5363.41386	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**	**-5706.53701**
F9	**0**	2.88073E-08	2.44344E-08	**0**	**0**	4.44089E-14	3.19744E-13	**0**	**0**
F10	**7.10543E-15**	4.91076E-05	4.08633E-05	**7.10543E-15**	**7.10543E-15**	3.55271E-14	2.45137E-13	**7.10543E-15**	**7.10543E-15**
F11	0.00739604	1.96423E-07	5.27804E-08	**0**	**0**	1.27676E-14	6.88338E-15	**0**	**0**
F12	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**	**0.049765827**
F13	**0.002614683**	**0.002614685**	**0.002614684**	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**	**0.002614683**
Total	8	4	2	10	10	6	5	**12**	11
Unique	0	**1**	0	0	0	0	0	0	0



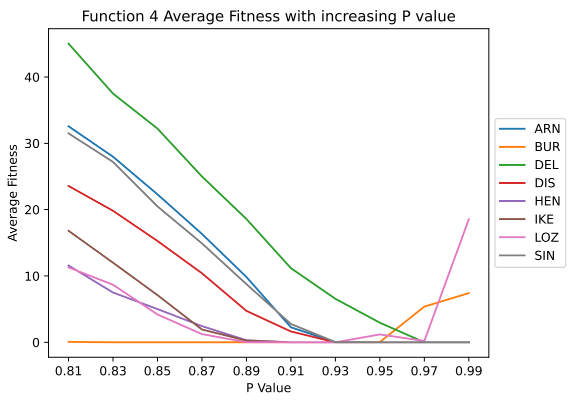
(a) Function 1



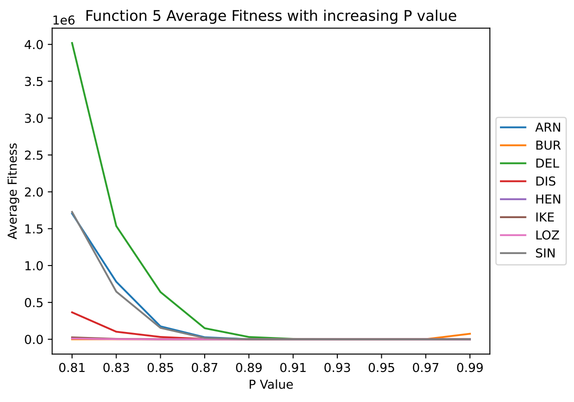
(b) Function 2



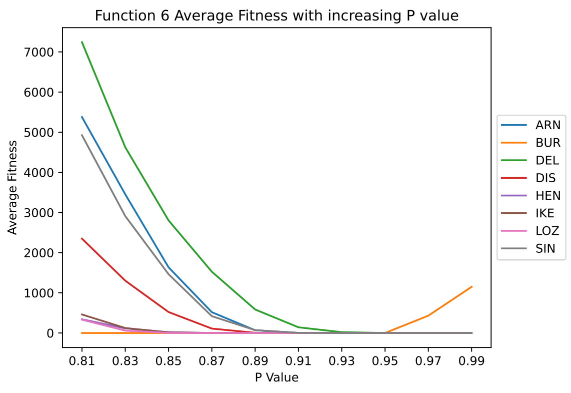
(c) Function 3



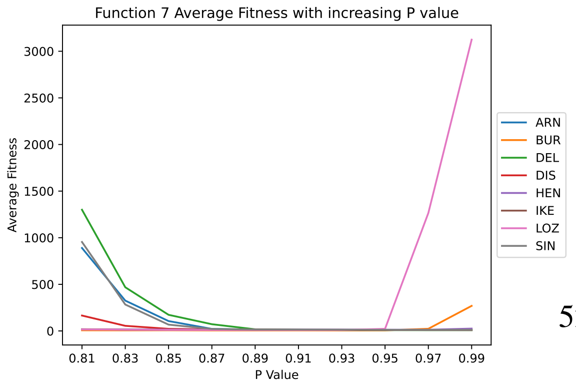
(d) Function 4



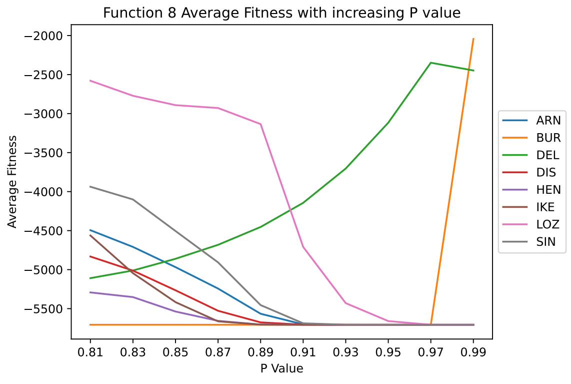
(e) Function 5



(f) Function 6

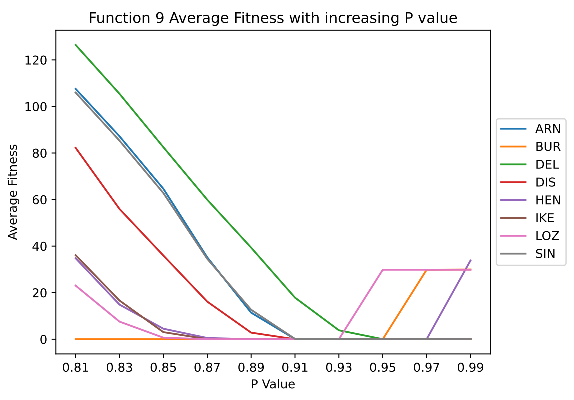


(g) Function 7

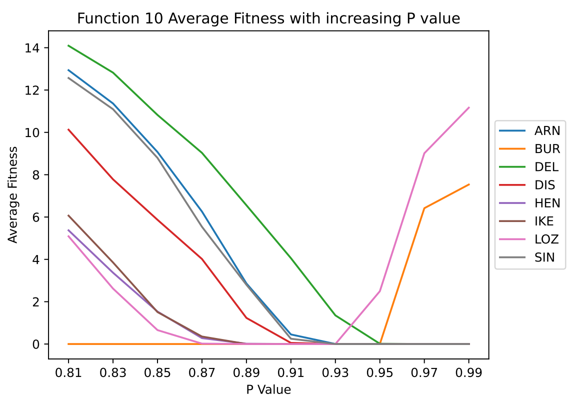


(h) Function 8

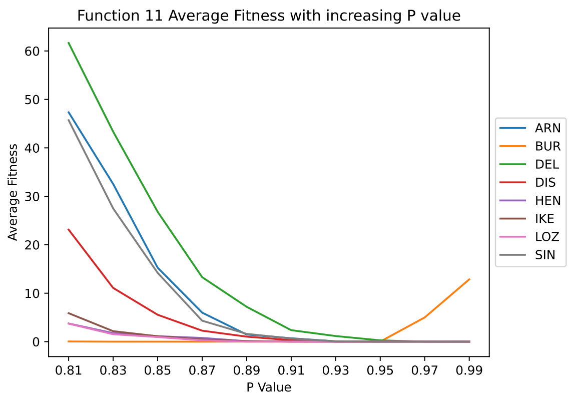
FIGURE 19: Randomizer Average Comparisons for Function 1 - 8



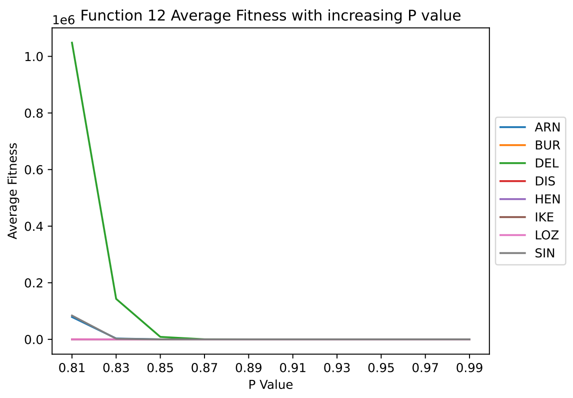
(a) Function 9



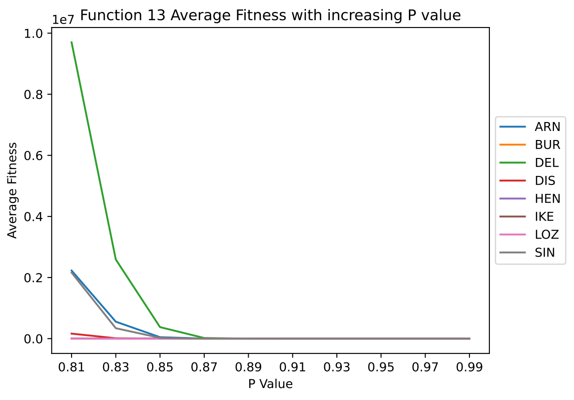
(b) Function 10



(c) Function 11

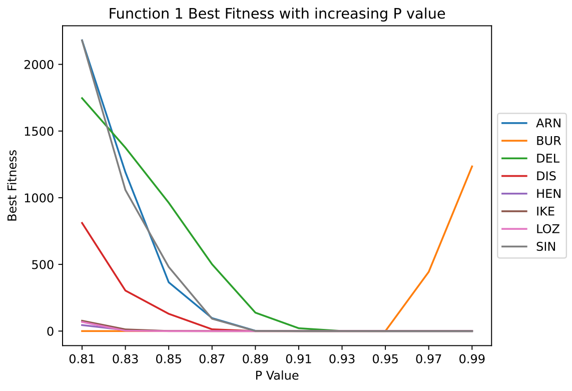


(d) Function 12

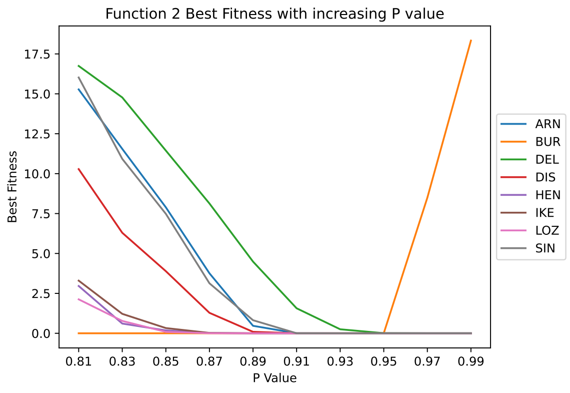


(e) Function 13

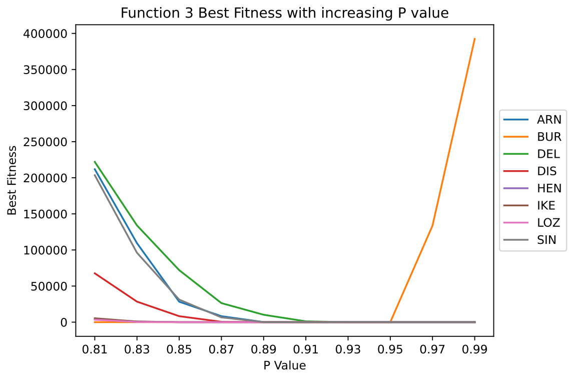
FIGURE 20: Randomizer Average Comparisons for Function 9 - 13 (cont.)



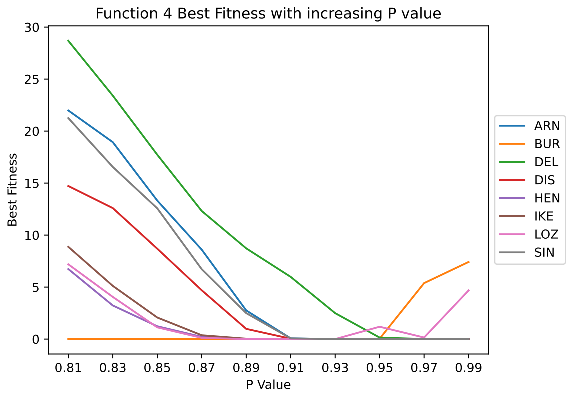
(a) Function 1



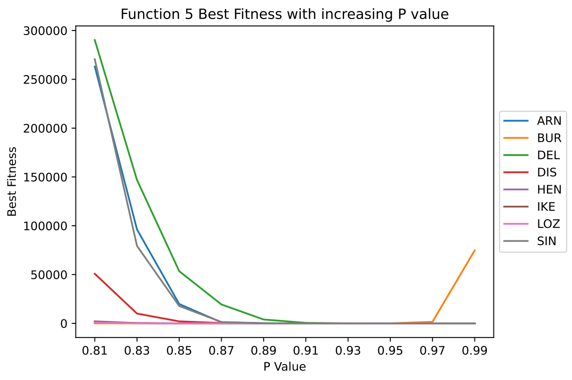
(b) Function 2



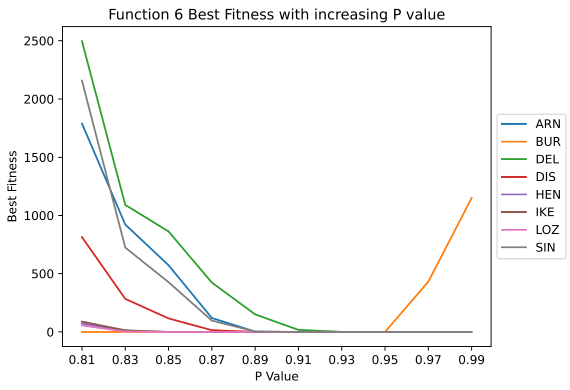
(c) Function 3



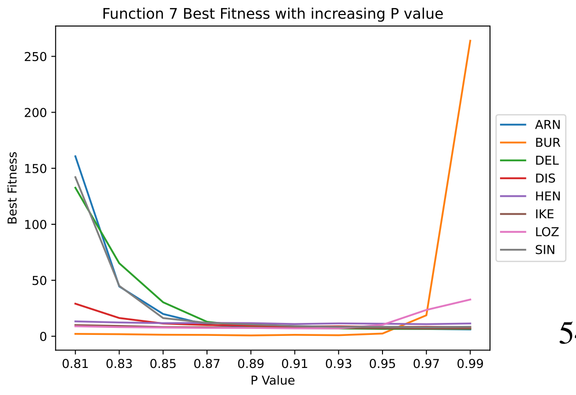
(d) Function 4



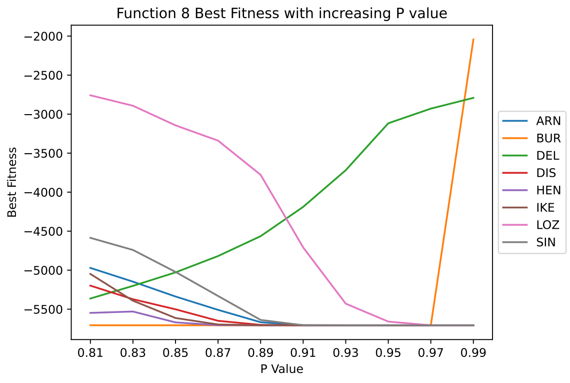
(e) Function 5



(f) Function 6

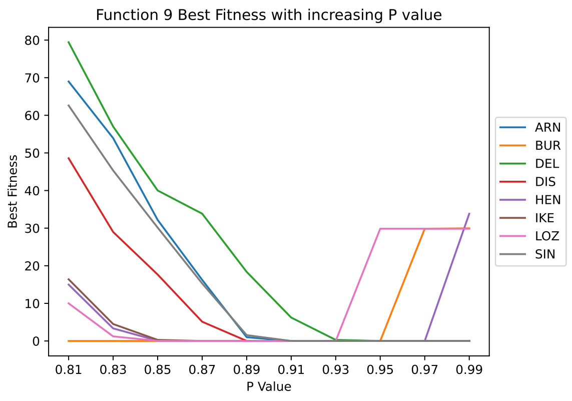


(g) Function 7

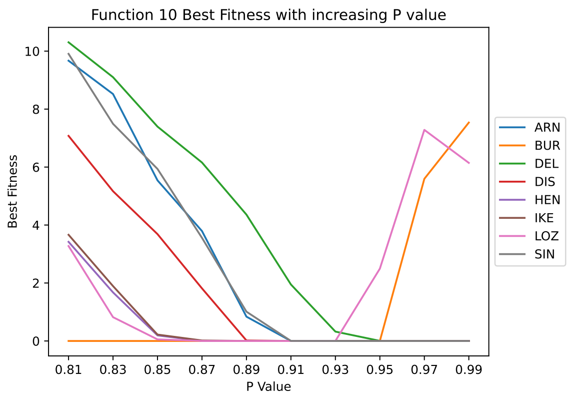


(h) Function 8

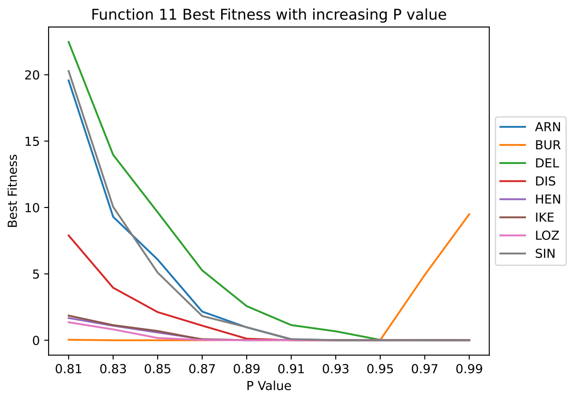
FIGURE 21: Randomizer Best Comparisons for Function 1 - 8



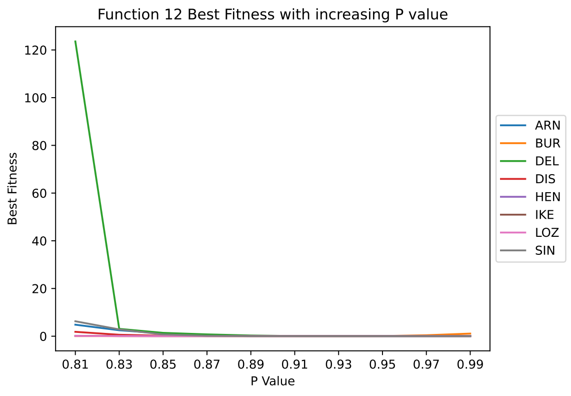
(a) Function 9



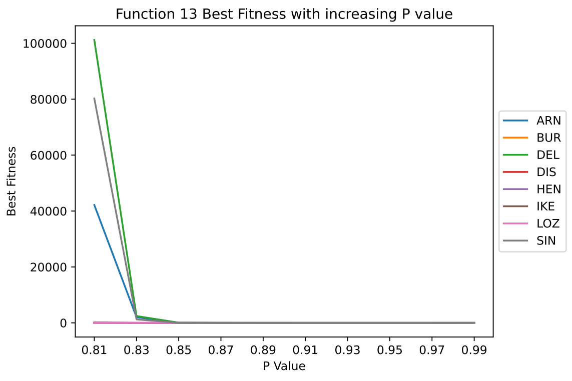
(b) Function 10



(c) Function 11



(d) Function 12



(e) Function 13

FIGURE 22: Randomizer Best Comparisons for Function 9 - 13 (cont.)

These experiments also lead to Mersenne Twister having more optimal parameters than during initial testing, leading it to finding the most optimal result on Function 6, which it had not done before. A table showing the new best values CHO has found compared to other Algorithms is shown on Table 22. CHO is now capable of finding the optimal result in eight of the thirteen functions, where early testing (Table 4) showed optimal results being found in only seven.

TABLE 22: Comparison of Algorithms with Optimized Parameters

F	CHO		GWO		PSO		GSA		DE		FEP	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std	Avg	Std	Avg	Std
F1	**0**	0	6.59E-28	6.34E-05	0.00013	0.0002	2.53E-16	9.67E-17	8.20E-14	5.90E-14	0.00057	0.00013
F2	**0**	0	7.18E-17	0.029	0.04214	0.04542	0.05565	0.19407	1.50E-09	9.90E-10	0.0081	0.00077
F3	**0**	0	3.29E-06	79.1495	70.1256	22.1192	896.534	318.955	6.80E-11	7.4E-11	0.016	0.014
F4	**0**	0	5.61E-07	1.315	1.08648	0.31703	7.35487	1.174145	**0**	0	0.3	0.5
F5	9.015-10	11.219	26.8125	69.9049	96.7183	60.1155	67.543	62.2253	**0**	0	5.06	5.87
F6	**0**	0	0.8165	0.0001	0.0001	8.28E-05	2.50E-16	1.74E-16	**0**	0	**0**	0
F7	0.7321	0.7612	**0.0022**	0.1002	0.12285	0.04495	0.08944	0.04339	0.00463	0.0012	0.1415	0.3522
F8	-5706.537	2.227E-12	-6123.1	-4087.44	-4841.29	1152.81	-2821.07	493.0375	-11080.1	574.7	**-12554.5**	52.6
F9	**0**	0	0.3105	47.356	46.7042	11.62938	25.9684	7.47006	69.2	383.8	0.046	0.012
F10	**7.11E-15**	0	1.06E-13	0.0778	0.27601	0.50901	0.06208	0.23628	9.70E-08	9.70E-08	0.018	0.0021
F11	**0**	0	0.0044	0.0066	0.00921	0.00772	27.7015	5.04034	**0**	0	0.016	0.022
F12	0.04976	3.095E-17	0.0534	0.0207	0.00691	0.026301	1.79961	0.95114	**7.90E-15**	8.00E-15	9.20E-06	3.60E-06
F13	0.00261	1.032E-18	0.6544	0.0044	0.00667	0.008907	8.89908	7.12624	**5.10E-14**	4.80E-14	0.00016	0.000073

CHAPTER VI

CONCLUSION

Optimization Algorithms have many facets, one of which is metaheuristic optimization algorithms. These algorithms focus largely on exploration and exploitation of a solution space, with most of the computational resources being devoted to the exploration of that space. Crosshair Optimizer takes a stochastic approach that primarily focuses on exploitation of a solution space, having particles randomly place each of their dimensions into bounds that are changed after every iteration. Values are occasionally also allowed, at random for each of their dimensions individually, to be placed outside of these bounds. These bounds are centered on the best member of the population.

Viewing the same two dimensions of the entire population in a scatterplot shows something similar to a crosshair, where the middle of the crosshair is the best value found thus far, which is where the algorithm gets its name. The area that the crosshair covers in the solution space will be explored the most. This algorithm continues to run until either the average fitness of the population hasn't exceed a user defined minimum bound, or a maximum number of experimentations have been completed. Initial testing of this algorithm showed promising results, finding the global best value of seven of the thirteen fitness functions.

When the idea of using High-Performance Computing came up to see if the algorithm could take full advantage of a machine with a large amount of computational resources, initial analysis of the algorithm gave evidence that the algorithm is very capable of effectively using those resources. This is due to CHO being able to calculate what each dimension of each particle separately without any reliance on other parts of the particle, or the rest of the population. The bounds for each dimension can be calculated once, and then each dimension needs to be randomly placed within those bounds. Thus this algorithm is easily split into many tasks without any need to communicate between those tasks, and no required memory structure to put those tasks into.

When experimenting using POSIX Threads (PThreads), this proved to be true, though the speed up of the algorithm was largely based on the overall complexity of the fitness function it was being tested on, the more complex, the more speedup was achieved.

During initial testing, Mersenne Twister was used to generate random values. Since this algorithm relies heavily on randomness, experiments were run using different Chaos maps to test their effect on the results produced by the canonical algorithm. Two experiments were run, one with a smaller population and less iterations, and another which had a larger population and more iterations. The reason behind these two experiments was to test how effective a randomizer is initially, versus how effective a randomizer is if given more time.

In results, the same randomizers tended to stand out as the most effective, those being Arnold's Cat map, Henon Map, Mersenne Twister, Sinai map, and Dissapative map. However, other randomizers were capable of finding optimal results with unique fitness functions. Thus, if using CHO for a different fitness function, multiple randomizers should be considered as an ensemble system.

When using other randomizers, the other user defined variable associated with that randomizers use also needs to be tuned. Thus, an experiment that ran every randomizer with a different adj value was run as well, the adj value being what determines if dimensions may land outside of its given bounds or not. Results showed that every value had a different adj that was needed for it to produce it's most optimal result, but evidence showed that what each randomizer needed was largely based on the fitness function, rather than the randomizer itself.

After all these experiments, CHO was capable of finding one additional global best value compared to its initial testing, now finding the global best in eight of the thirteen benchmark functions.

In conclusion, evidence shows that CHO is a very capable and versatile algorithm that not only generates optimal results, but also capable of use in a high-performance computing environment, and though it requires tuning to use other randomizer, using those other randomizers allows for CHO to be capable of finding optimal values with other fitness functions, making it more versatile.

REFERENCES CITED

[1] S. Mirjalili, S. M. Mirjalili, and A. Lewis, "Grey Wolf Optimizer," *Advances in Engineering Software*, vol. 69, pp. 46–61, 2014.

[2] D. Davendra and J. Torrence, "Crosshair Optimizer," in *2022 IEEE Workshop on Complexity in Engineering (COMPENG)*, pp. 1–5, 2022.

[3] T. Odagaki, "Variable range Random Walk," *Physica A: Statistical Mechanics and its Applications*, vol. 603, p. 127781, 2022.

[4] W. L. Dunn and J. K. Shultis, "Markov Chain Monte Carlo," in *Exploring Monte Carlo Methods (Second Edition)* (W. L. Dunn and J. K. Shultis, eds.), pp. 189–254, Elsevier, second edition ed., 2023.

[5] F. Busetto, G. Codognato, and S. Tonin, "Simple majority rule and integer programming," *Mathematical Social Sciences*, vol. 113, pp. 160–163, 2021.

[6] M. Metlicka, D. Davendra, F. Hermann, M. Meier, and M. Amann, "GPU accelerated NEH algorithm," in *2014 IEEE Symposium on Computational Intelligence in Production and Logistics Systems (CIPLS)*, pp. 114–119, 2014.

[7] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1989.

[8] J. Kennedy and R. Eberhart, "Particle Swarm Optimization," in *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, pp. 1942–1948 vol.4, 1995.

[9] G. Lancia, F. Rinaldi, and P. Serafini, "Local Search inequalities," *Discrete Optimization*, vol. 16, pp. 76–89, 2015.

[10] A. Banerjee, D. Singh, S. Sahana, and I. Nath, "Impacts of Metaheuristic and Swarm Intelligence approach in Optimization," in *Cognitive Big Data Intelligence with a Metaheuristic Approach* (S. Mishra, H. K. Tripathy, P. K. Mallick, A. K. Sangaiah, and G.-S. Chae, eds.), Cognitive Data Science in Sustainable Computing, pp. 71–99, Academic Press, 2022.

[11] T. Stojanovski and L. Kocarev, "Chaos-Based random number generators — Part I: Analysis," *IEEE Transactions on Circuits and Systems - I: Fundamental Theory and Applications*, vol. 48, no. 3, pp. 281–288, 2001.

[12] R. Storn and K. Price, "Differential Evolution –A simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[13] X. Yao, Y. Liu, and G. Lin, “Evolutionary Programming made faster,” *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 2, pp. 82–102, 1999.

[14] E. Rashedi, H. Nezamabadi-pour, and S. Saryazdi, “GSA: A Gravitational Search Algorithm,” *Information Sciences*, vol. 179, no. 13, pp. 2232–2248, 2009. Special Section on High Order Fuzzy Sets.

[15] M. Jafari Barani, P. Ayubi, M. Yousefi Valandar, and B. Y. Irani, “A new pseudo random number generator based on generalized Newton complex map with dynamic key,” *Journal of Information Security and Applications*, vol. 53, p. 102509, 2020.

[16] E. N. Lorenz, “Deterministic nonperiodic flow,” *Journal of atmospheric sciences*, vol. 20, no. 2, pp. 130–141, 1963.

[17] M. J. Barani, M. Y. Valandar, and P. Ayubi, “A new digital image tamper detection algorithm based on integer wavelet transform and secured by encrypted authentication sequence with 3D Quantum map,” *Optik*, vol. 187, pp. 205–222, 2019.

[18] C. Herring and P. Julian, “Random number generators are Chaotic,” *ACM Sigplan*, vol. 11, pp. 1–4, 1989.

[19] D. Lehmer, “Mathematical methods in large-scale computing units,” *Ann. Computing Lab, Harvard University*, vol. 26, pp. 141–146, 1951.

[20] J. Palmore and J. McCauley, “Shadowing by computable Chaotic Orbits,” *Physics Letters A*, vol. 121, p. 399, 1987.

[21] I. Zelinka, M. Chadli, D. Davendra, R. Senkerik, M. Pluhacek, and J. Lampinen, “Hidden Periodicity - Chaos dependance on numerical precision,” *Advances in Intelligent Systems and Computing*, vol. 210, pp. 47–59, 2013.

[22] R. Lozi, “New enhanced Chaotic number generators,” *Indian Journal of Industrial and Applied Mathematics*, vol. 1, no. 1, pp. 1–23, 2008.

[23] R. Lozi, “Chaotic pseudo random number generators via ultra weak coupling of Chaotic Maps and double threshold sampling sequences,” in *ICCSA 2009 The 3rd International Conference on Complex Systems and Applications*, (University of Le Havre, France), pp. 1–5, June 2009.

[24] I. Zelinka, M. Chadli, D. Davendra, R. Senkerik, M. Pluhacek, and J. Lampinen, “Do Evolutionary Algorithms indeed require random numbers? Extended study,” *Advances in Intelligent Systems and Computing*, vol. 210, pp. 61–75, 2013.

[25] X.-Y. Wang and X. Qin, “A new pseudo-random number generator based on CML and Chaotic iteration,” *Nonlinear Dynamics*, vol. 70, pp. 1589–1592, Oct 2012.

[26] D. Lambić and M. Nikolić, “Pseudo-random number generator based on discrete-space Chaotic map,” *Nonlinear Dynamics*, vol. 90, pp. 223–232, Oct 2017.

[27] D. Lambić, “Security analysis and improvement of the pseudo-random number generator based on Quantum Chaotic map,” *Nonlinear Dynamics*, vol. 94, pp. 1117–1126, Oct 2018.

[28] A. Akhshani, A. Akhavan, A. Mobaraki, S.-C. Lim, and Z. Hassan, “Pseudo random number generator based on Quantum Chaotic map,” *Communications in Nonlinear Science and Numerical Simulation*, vol. 19, no. 1, pp. 101–111, 2014.

[29] K. Wang, W. Pei, H. Xia, and Y. ming Cheung, “Pseudo random number generator based on asymptotic deterministic randomness,” *Physics Letters A*, vol. 372, no. 24, pp. 4388–4394, 2008.

[30] H. Zhu, C. Zhao, X. Zhang, and L. Yang, “A novel Iris and Chaos-based random number generator,” *Computers & Security*, vol. 36, pp. 40–48, 2013.

[31] M. François, T. Grosges, D. Barchiesi, and R. Erra, “Pseudo-random number generator based on mixing of three Chaotic maps,” *Communications in Nonlinear Science and Numerical Simulation*, vol. 19, no. 4, pp. 887–895, 2014.

[32] M. A. Dastgheib and M. Farhang, “A digital pseudo-random number generator based on Sawtooth Chaotic map with a guaranteed enhanced period,” *Nonlinear Dynamics*, vol. 89, pp. 2957–2966, Sep 2017.

[33] J. Sprott, *Chaos and Time-Series Analysis*. UK: Oxford University Press, 2003.

[34] J. Burgers, “Mathematical examples illustrating relations occurring in the theory of turbulent fluid motion,” in *Selected Papers of J. M. Burgers* (F. Nieuwstadt and J. Steketee, eds.), pp. 281–334, Springer Netherlands, 1995.

[35] D. Aronson, M. Chory, G. Hall, and R. McGehee, “A discrete dynamical system with subtly wild behavior,” in *New Approaches to Nonlinear Problems in Dynamics*, pp. 339–359, Philadelphia, Pennsylvania: SIAM Publications, 1980.

[36] R. Senkerik, I. Zelinka, M. Pluhacek, D. Davendra, and Z. O. Kominkova, “Chaos Enhanced Differential Evolution in the task of Evolutionary Control of selected set of discrete Chaotic systems,” *The Scientific World Journal*, vol. 2014, 2014.