CUED - Engineering Tripos Part IIB 2023-2024 **Module Coursework**

Module	**4M17**	Title of report	**Assignment 2: Optimisation Algorithm Performance Comparison Study**

UNDERGRADUATE STUDENTS ONLY		Assessment for this module is ☑ 100% / ☐ 25% coursework of which this assignment forms **50%**	
Candidate number:	5553C		
MPHIL STUDENTS ONLY		Date submitted:	19/01/2024
Candidate number:			
OTHER POSTGRADUATE STUDENTS			
Name:			
CRSid:			

Feedback to the student ☐ **See also comments in the text**		Very good	**Good**	Needs imprvmt
CONTENT	**Completeness, quantity of content:** Has the report covered all aspects of the assignment? Has the analysis been carried out thoroughly?			
	Correctness, quality of content Is the data correct? Is the analysis of the data correct? Are the conclusions correct?			
	Depth of understanding, quality of discussion Does the report show a good technical understanding? Have all the relevant conclusions been drawn?			
	Comments:			
PR	**Attention to detail, typesetting and typographical errors** Is the report free of typographical errors? Are the figures/tables/references presented professionally?			

Comparison of Genetic Algorithms and Particle Swarm Optimisation

5535C

January 19, 2024

Abstract

Real-world applications requiring optimisation often have multi-modal objective functions with solutions in difficult places to reach. In this report we explore the application of Genetic algorithms, Particle Swarm Optimisation and a hybrid of the two, on the 8-D Keane's Bump Function. We found that PSO is consistently able to find the best solution of all the methods, with GA performing poorly in comparison. This was expected as the GA is suited for optimising within discrete solution spaces. The poor performance of the GA ultimately led to the hybrid yielding insignificant results compared to the PSO. Furthermore, due to the vectorisation of PSO, it was able to process the same number of objective function evaluations approximately 6 times quicker than the GA and the hybrid method.

1 Introduction

Whilst optimisation problems tend to be complex human constructs, often effective and simple approaches to finding the best solution can be found by observing nature; despite the lack of fluency in mathematics, nature has had the advantage of developing optimal approaches to solving problems for thousands of years.

Here we will use two bio-inspired algorithms, which don't evaluate any derivatives, that replicate processes found in the wild, to maximise the 2-dimensional and 8-dimensional Keane's Bump Function. The equation and constraints used are given in Equation 1, where n is the number of dimensions.

$$\begin{aligned} \textbf{Maximise } f(\mathbf{x}) &= \left| \frac{\sum_{i=1}^{n}(cos(x_i)^4 - 2\prod_{i=1}^{n}(cos(x_i)^2}{\sqrt{\sum_{i=1}^{n} i(x_i)^2}} \right| \\ \textbf{Subject to } & 0 \leq x_i \leq 10 \\ & \prod_{i=1}^{n} x_i > 0.75 \\ & \sum_{i=1}^{n} < \frac{15n}{2} \end{aligned} \tag{1}$$

As we can see in Figure 1, this is a highly multi-modal function, with the maximum located at a non-linear boundary, hence this will be a hard problem to solve, with many algorithms likely getting stuck in sub-optimal maxima.

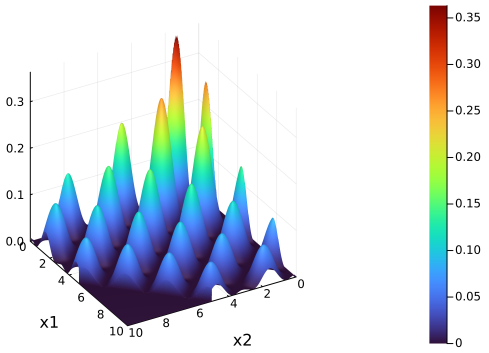


Figure 1: Surface plot of the 2-D KBF given by Equation 1.

2 Genetic Algorithm (GA)

The Genetic Algorithm focuses on utilising Darwinism on the genes of the population of possible solutions; it uses three mechanisms to replicate natural evolution: parent selection, crossover/breeding and mutation. Firstly, the fitness of a particular solution increases its chance of being selected as a parent. Secondly, the selected parents are the 'crossed over' producing two offspring that inherit parts of their genotype from the parents. Lastly, a small chance of mutation of the genotype is present to diversify the population.

Algorithm 1 Genetic Algorithm

```
Generate Initial Population
Evaluate Initial Populations Fitness
while Objective Function Evaluations < 10,000 do
    Select Parents
    Breed New Population
    Mutate New Population
    Evaluate New Populations Fitness
    Update Dissimilar Archive
Stop
```

2.1 Parent Selection

After the initial population is randomly generated, Parent selection is used to replicate 'survival of the fittest' in nature, whereby the most capable and genetically suitable species survive the tests of time and can pass on their genetic material. Here we will propose several different ways of selecting the parent population. Note that elitism, the practice of ensuring the best solution is always a parent, is not used, as this is found to perform poorly in multi-modal problems (De Jong[1975]). This is because it increases the chance of the solution remaining within a local optimum.

2.1.1 Roulette Selection

Roulette Selection simply draws N parents, with the probability of a solution selected determined by Equation 2. Where f_i is the fitness of a solution.

$$P_{si} = \frac{f_i}{f_{\sum}} \tag{2}$$

When implementing this technique, two samples are drawn at a time to create a parent pair that would breed together, Resulting in N/2 parent pairs. Note a solution can be selected as a parent more than once.

2.1.2 Tournament Selection

Tournament Selection works by Uniformly selecting a subset of the population and selecting the best solution(s) to be part of the parent population. This is repeated until N parents have been selected. Like Roulette Selection, this method is easy to implement. Furthermore, in implementation, the best two solutions of each subset are taken as a parent pair immediately.

2.1.3 Stochastic Remainder Selection without Replacement

The most complicated of selection methods seeks to introduce a greater sense of determinism in the parent population. We start by calculating the expected number of times a parent will be selected using Equation 3.

$$E_i = N \times P_{si} \tag{3}$$

The number of times a parent is selected is determined by the integer part of Equation 3, then the rest of the parents are selected using the remainder (decimal part) as the probability of selection until a total of N parents have been selected. The parents are then randomly paired up.

2.2 Crossover

With the parent population selected, a process by which their genes are passed on to their offspring must be created. The aim is that an offspring will be made which inherits the high-performing genes from both parents and therefore evolves to become a stronger solution.

2.2.1 Binary Crossover

In Binary Crossover, we represent each solution (more, specifically the coordinates of the solution) as a vector of bitstrings; each bitstring is determined by the binary representation of the Float64 number representing the solutions state in one of the dimensions. When breeding two parents together, the bitstrings of each dimension are compared to each other.

In the One Point Crossover, a single number is uniformly sampled from [1:64] (Note it is 64 due to Float64 using 64 bits). This is the crossover point; Child 1 will inherit all bits to the left of this point (inclusive) from Parent 2 and the rest of the bits from Parent 1. The remaining bits from both parents are then simply put together to create Child 2.

In the Two Point Crossover, two points are selected and the substring in between is exchanged between parents, creating two new offspring.

Lastly, in Random Crossover if the parents have identical bits in certain positions, both offspring will inherit these bits; if the parents have different bits in corresponding positions, it is randomly chosen which child will inherit from which parent.

2.2.2 Simulated Binary Crossover (SBX)

Simulated Binary Crossover [1] attempts to replicate the way new offspring are produced using Binary One Point Crossover, but without the overhead of converting the coordinates into a vector of bitstrings, then crossing over bits, and then converting back. Instead, the coordinates are kept in their original numeric state. The coordinate of an offspring is determined using Equation 4. This is repeated for each dimension (Child1, Child2, Parent1 and Parent2 represent the value in a given dimension).

$$\begin{aligned} u &= Uniform(0:1) \\ \beta &= (2u)^{\frac{1}{3}} \text{ if } u \leq 0.5 \\ &= \left(\frac{0.5}{1-u}\right)^{\frac{1}{3}} \text{ otherwise} \\ Child1 &= 0.5[(1+\beta)Parent1 + (1-\beta)Parent2] \\ Child2 &= 0.5[(1-\beta)Parent1 + (1+\beta)Parent2] \end{aligned} \tag{4}$$

2.3 Mutation

Lastly, in nature often dominant species arise via the minuscule chance of a gene mutating and leading to beneficial phenotype. This mechanism is implemented in GA as similarly, it allows the population to diversify and explore other possible solutions. This is key to ensuring GA doesn't get stuck in a local optimum.

As mentioned above, editing the bits of numbers introduces considerable overhead, therefore when mutating, we will be simply introducing noise with a normal distribution to the coordinates of a solution.

2.4 Implementation Details

Due to the simplicity of the inequality constraints, the initial population is generated randomly within an area that doesn't violate the constraints of the problem. Furthermore, offspring that, either through crossover or mutation, violate the constraints are rejected; If N offspring have not been accepted after the parent pairs have each produced two offspring, then the parent pairs will be looped through again until N offspring have been accepted.

Furthermore, it is important to note that in each iteration, a completely new generation is computed. Alternate schemes may only breed and mutate a subset of the population, which has its advantages, for example when optimising temporal fitness functions.

Lastly, for each iteration, N objective function calls are used. With our limit of 10,000 calls that would mean, for example, a population of 100 would give 100 possible iterations.

2.5 Hyper-parameters

Population_size (N) is the number of solutions in a given generation.

Breed_Probability determines the chance that a parent pair will breed and produce offspring. If they do not breed, the parents are simply passed into the next generation as they are.

Mutation_Probability controls how often a solution is mutated before it is passed into the next generation.

2.6 Technique Selection

In this section, we will assess the performance of varying techniques for the GA. We will use the 2D-KBF to determine the best Parent Selection and Crossover techniques.

When determining the best Parent Selection and Crossover techniques, we will use the following commonly used hyper-parameter settings:

1. Population_size (N) = 100
2. Breed_Probability = 0.65
3. Mutation_Probability = 0.1

2.6.1 Parent Selection

As we can see from Table 1, Tournament Selection is the superior selection process when testing on 2D-KBF. Both Roulette Selection and SRSWR obtain similar results; from Figure 1 we observe that both selection techniques regularly find their best-archived solution near the global maximum, yet are not able to locate its peak. Furthermore, the final fitness of the final top 10 solutions is located on a different maxima, indicating that the algorithm is stuck in another local maxima.

This is in contrast to Tournament Selection, which can not only archive a better maximum solution but also consistently converge on this solution as shown by the similarly valued top 10 final solution average.

Testing various tournament subsizes yields 25% as the optimal value (25% of the population is randomly sampled, with the bets two solutions within becoming a parent pair).

	Archive Min. Fitness		Top10 Avg. Fitness	
	Avg.	Std.	Avg.	Std.
Roulette	-0.3196	0.03369	-0.2670	0.03185
Tournament (subsize=30%)	-0.3359	0.03667	-0.3352	0.03692
Tournament (subsize=25%)	-0.3394	0.03479	-0.3392	0.03488
Tournament (subsize=15%)	-0.3360	0.04082	-0.3351	0.04117
SRSWR	-0.3114	0.03485	-0.2638	0.01143

Table 1: Table comparing the results (4 s.f.) of different Parent Selection techniques on 2D-KBF with Simulated Binary Crossover, Population_size (N) = 100, Breed_Probability = 0.65 and Mutation_Probability = 0.1. Each selection process was tested 100 times. The average fitness and standard deviation of both the archive maximum and the average of the final top 10 solutions were recorded.

Tournament selection was likely the dominant process due to its ability to consistently select a variety of high-performing solutions through random subsamples. Hence when breeding these solutions, offspring could draw from a larger pool of optimal genes, leading to a greater level of exploration in areas of interest. Whereas the other two selection methods are more susceptible to being dominated by a few super-individuals as a result of using P_{si}. Individuals initialised in local maxima, would dominate this distribution and hence be selected many times. This would lead to a low variety in the parent population and hence premature convergence. The larger archive maximum indicates that a solution on the global maxima was consistently found, yet due to the vast number of solutions on a local maxima, this was unable to migrate the population to the better solution.

2.6.2 Crossover

From Table 2 we observe SBX is the best-performing crossover technique, achieving the highest average fitness in both categories as well as the lowest standard deviation, indicating that it is the best and also the most consistent.

The dominant performance of SBX highlights the drawback of using GAs to solve problems in a continuous domain. Initially, GAs were predominantly used in discrete domains, such as integer-only, which could easily be represented in binary, whereas continuous domains were traditionally operated on by the closely related Evolution Strategies (Rechenberg[1973]). The first three crossover methods rely on altering the bits of a Float64 number, which due to the design of this, means that altering the vast majority of the 52 fraction-bits has little impact, whilst altering the 11 exponent-bits will have a huge impact, likely producing an offspring that violates the constraints (this explains in part the considerable amount of time take). Due to this, the GA struggles to perform well altering the 'genes' as it usually would from traditional crossover on binary representations of integers.

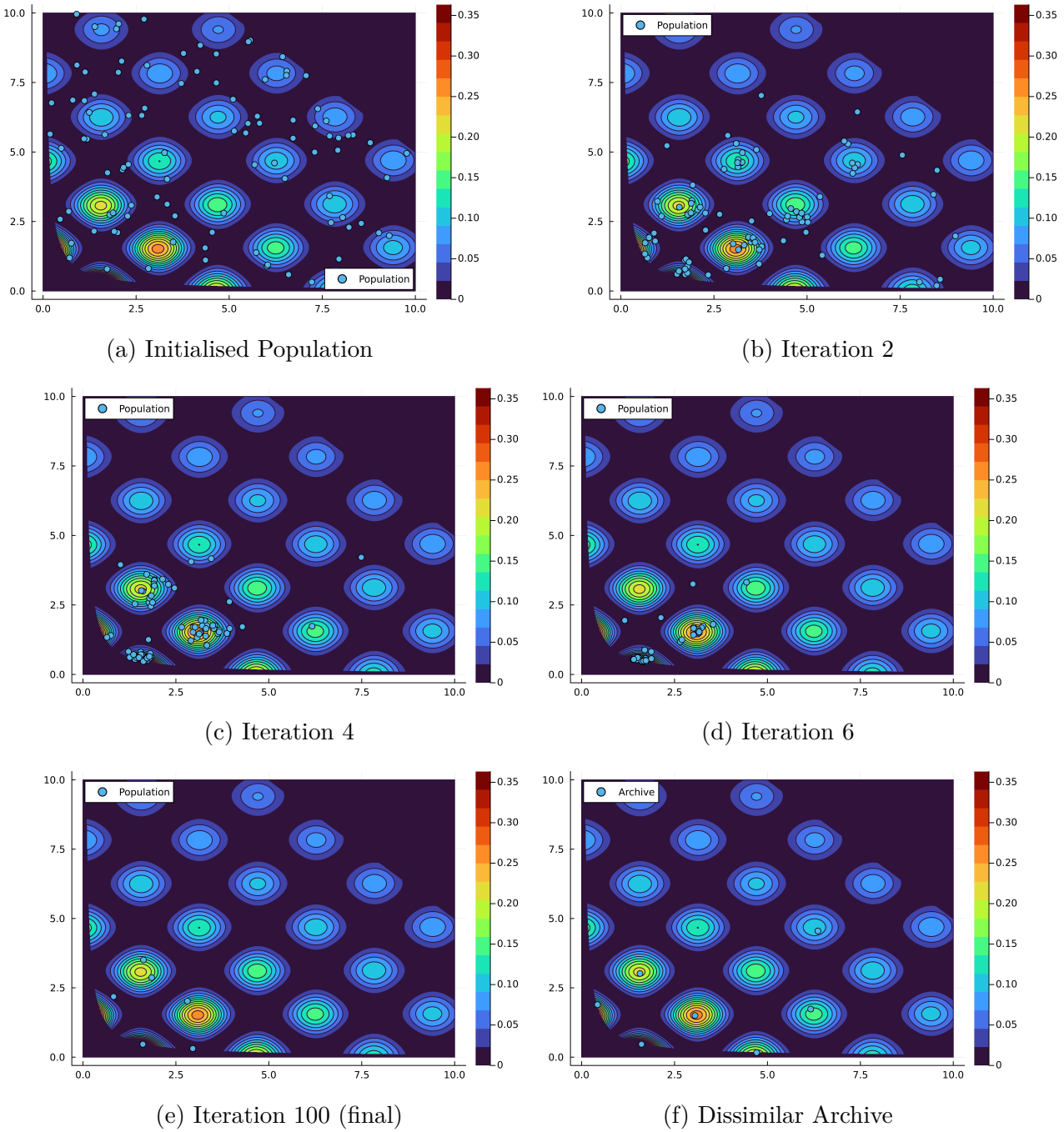
SBX on the other hand simulates the one-point crossover of integers and can capture the greater optimality of the performance a GA usually experiences in solving these problems. Furthermore, it will scale better in higher dimensions, as it is fully vectorised, whereas for binary representation each new dimension introduces a new bitstring that has to be decoded, looped through and then encoded again.

	Archive Min. Fitness		Top10 Avg. Fitness		Time Taken (ms)	
	Avg.	Std.	Avg.	Std.	Avg.	Std.
One Point Crossover	-0.3142	0.03496	-0.3135	0.03526	144.9	6.952
Two Point Crossover	-0.3171	0.03541	-0.3170	0.03550	148.5	9.613
Random Crossover	-0.3097	0.03735	-0.3086	0.03745	155.6	7.491
SBX	-0.3429	0.03374	0.3424	0.03439	86.98	4.763

Table 2: Table comparing the results (4 s.f.) of different Crossover techniques on 2D-KBF with Tournament Selection (subsize=25%), Population_size (N) = 100, Breed_Probability = 0.65 and Mutation_Probability = 0.1. Each crossover technique was tested 100 times. The average fitness and standard deviation of both the archive maximum and the average of the final top 10 solutions were recorded.

2.6.3 2-D KBF Results

With the Hyper-parameters defined above and the chosen techniques, we can now visualise the performance of the GA, as shown in Figure 2. We observe that we can quickly converge to the area containing the maxima. In Figure 2f we note that through adequate exploration the GA can locate other largest local maximums too. The Figure below also shows us how the GA can rapidly converge in 2D.



(a) Initialised Population

(b) Iteration 2

(c) Iteration 4

(d) Iteration 6

(e) Iteration 100 (final)

(f) Dissimilar Archive

Figure 2: Contour plot of the constrained 2-D KBF with a scatter plot of the GA population at different iterations.

2.7 8-D KBF Hyper-parameter Tuning and Results

Now using the 8-D KBF we measure the average archive max fitness and standard deviation for 100 runs on each possible combination of hyper-parameter values from Population_size (N) = [100, 150, 200, 250, 400], Breed_Probability = [0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9] and Mutation_Probability = [0.4, 0.35, 0.3, 0.25, 0.2, 0.1, 0.05].

We found the best combination of hyper-parameters was given by:

1. Population_size (N) = 400
2. Breed_Probability = 0.85

3. Mutation_Probability = 0.35

For the 8-D KBF, this yielded an average archive maximum fitness of $\mathbf{-0.5614}$ and a standard deviation of **0.08188**. This takes **84.49ms** on average (CPU time) with a standard deviation of **6.608ms**.

3 Particle Swarm Optimisation (PSO)

Particle Swarm Optimisation was inspired by the smooth, coordinated movement of flocks of birds; despite the limited space and visibility of each bird, as well as high speeds, the flock can maneuver effectively due to social interaction between the birds. PSO seeks to use social interactions between solutions to explore the solution space more efficiently and converge upon the global maxima.

Algorithm 2 Genetic Algorithm

```
Generate Initial Population
Generate Random Velocities
Evaluate Initial Population
while Objective Function Evaluations < 10,000 do
    Update Population Velocities
    Update Population Positions
    Evaluate New Populations Fitness
    Update Dissimilar Archive
Stop
```

Unlike the GA, PSO considers each solution within the population to consistently be present with each iteration. In each iteration, the position of all the solutions are updated depending on their velocities. At the beginning of each iteration, the velocity of the particles are calculated using Equation 5. $\mathbf{x_{best}}$ is the best solution an individual within the population has found and $\mathbf{x_{populationbest}}$ is the best solution found by the entire population. Note '*' denotes element-wise multiplication. The use of $\mathbf{U_p}$ and $\mathbf{U_g}$ introduces a degree of randomness and ensures the solution space is explored.

$$\begin{aligned} \mathbf{U_p} &= Uniform(0,1) \\ \mathbf{U_g} &= Uniform(0,1) \\ \mathbf{v_{i+1}} &= \omega * \mathbf{v_i} + \phi_p * \mathbf{U_p} * (\mathbf{x_{particlebest}} - \mathbf{x_i}) + \phi_g * \mathbf{U_g} * (\mathbf{x_{populationbest}} - \mathbf{x_i}) \end{aligned} \tag{5}$$

3.1 Implementation Details

Like GA the initial population is randomly generated within such that the constraints are not violated. Also, the velocities are initialised uniformly between -3 and 3.

Secondly, if a particle were to violate $0 \leq x_i \leq 10$ constraint for any given dimension after the position update step, that dimension will be fixed to either 0 or 10, depending on whichever is closer.

Lastly, the velocity is truncated at each step such that the magnitude in any given dimension is ≤ 3.

3.2 Hyper-parameters

Equation 5 introduces three of the hyper-parameters present in PSO: ω dictates the contribution from the velocity in the previous iteration and ϕ_g and ϕ_g control the contribution of the distance of the particle from its own best solution and the populations best solution, respectively, to the velocity. The remaining hyper-parameter is population size (N).

ω is often called the inertia weight, as it carries on the 'momentum' of the particle, leading to a smoother trajectory.

ϕ_g is called the cognitive coefficient, as it determines how much the velocity of a particle is determined by its desire to move towards its own best solution.

ϕ_g is called the social coefficient, as it is responsible for emulating the social interactions seen in swarms. This weight encourages the particle to move towards the best solution found within the history of the whole population.

3.2.1 Temporal vs Atemporal Hyper-parameters

The hyper-parameters ω, ϕ_g and ϕ_g do not have to remain constant throughout all the iterations; At first, it may be beneficial to have higher ω and ϕ_p, as these will encourage the particles to explore a greater amount of the solution space. Then as the space is explored, ϕ_g could increase to encourage the particles to gravitate towards the best solution.

The proposed temporal hyper-parameter scheme described above was implemented following Equation 6. Both ω and ϕ_p will halve in value from start to finish, whereas ϕ_g will double in value.

$$\begin{aligned} \omega &= \omega - \frac{0.5\omega_{initial}}{iteration} \\ \phi_p &= \phi_p - \frac{0.5\phi_{p\_initial}}{iteration} \\ \phi_g &= \phi_g + \frac{\phi_{g\_initial}}{iteration} \end{aligned} \tag{6}$$

To fairly compare both of these schemes, we will tune both versions of the PSO for the 2-D KBF problem and compare the results, shown in Table 3.

Hyper-parameters	Archive Max.		Top10 Avg.	
	Avg.	Std.	Avg.	Std.
Temporal $\omega = 0.9$, $\phi_p = 3$, $\phi_g = 1.5$	-0.3649	0.0001113	-0.3636	0.0008425
Atemporal $\omega = 0.9$, $\phi_p = 1$, $\phi_g = 1.5$	-0.3649	0.0001184	-0.3607	0.002683

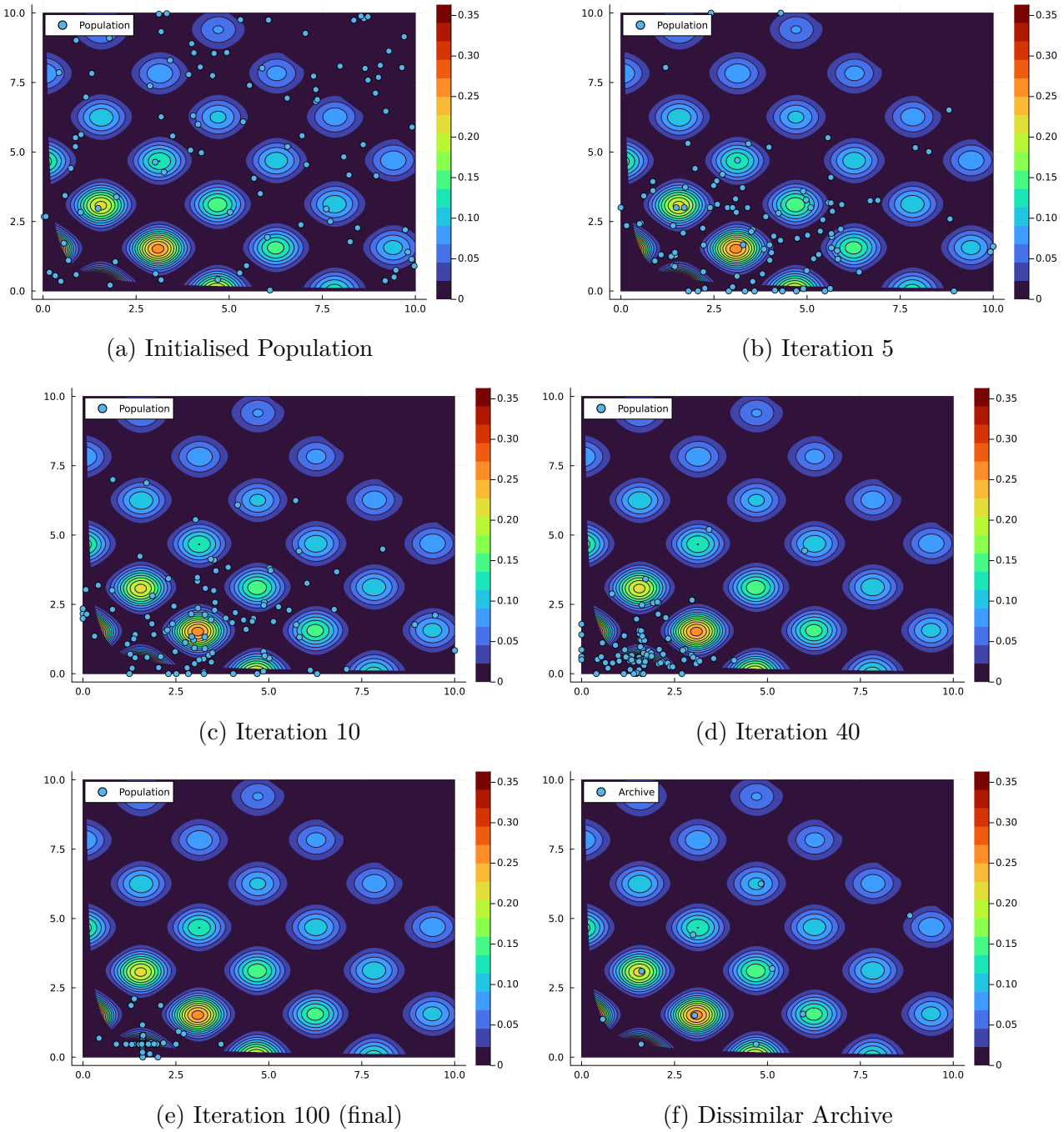
Table 3: Table of PSO results (4 s.f.) for tuned Temporal and Atemporal hyper-parameters on the 2-D KBF. N = 100. All combinations of ω = [0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9], ϕ_p = [0.5, 1, 1.5, 2, 2.5, 3] and ϕ_g = [0.5, 1, 1.5, 2, 2.5, 3] were tested. The average fitness and standard deviation of both the archive maximum and the average of the final top 10 solutions were recorded.

3.2.2 2-D KBF Results

We observe that performance is exceptional for both variations, although the top 10 final solutions do indicate that the temporal hyper-parameters do offer an advantage, achieving a

more optimal solution with a lower standard deviation. This was expected due to the decrease of ω and ϕ_p, which encourage explorations, and the increase of ϕ_g, which encourages convergence on the best solution, as more iterations are completed.

In Figure 3 we see this in action, as at first the space is explored, then towards the end the 'swarm' converges on the maximum. Figure 3f shows us that the global maximum was found as well as several other local maxima.



(a) Initialised Population

(b) Iteration 5

(c) Iteration 10

(d) Iteration 40

(e) Iteration 100 (final)

(f) Dissimilar Archive

Figure 3: Contour plot of the constrained 2-D KBF with a scatter plot of the PSO (with temporal hyper-parameters) population at a given iteration.

3.3 8-D KBF Hyper-parameter Tuning and Results

Now testing the temporal hyper-parameters for the 8-D KBF, with combinations of Population size (N) = [100, 150, 200, 250, 400], ω = [0.65, 0.7, 0.75, 0.8, 0.85, 0.9], ϕ_p = [2, 2.25, 2.5, 2.75,

3] and $\phi_g = [0.75, 1, 1.25, 1.5, 1.75, 2]$.

We found the best combination of hyper-parameters was given by:

1. Population_size (N) = 200
2. $\omega = 0.85$
3. $\phi_p = 2.5$
4. $\phi_g = 1.25$

This yielded an average archive maximum fitness of $\mathbf{-0.6839}$ and a standard deviation of $\mathbf{0.05253}$. This takes **14.66ms** on average with a standard deviation of **1.719ms**.

4 Hybrid GAPSO

In nature, swarm behaviour as well as natural evolution is displayed within thriving populations. Hence the combination of the two in optimisation seems fitting, as one seeks to take advantage of their respective strengths.

Algorithm 3 Hybrid GAPSO

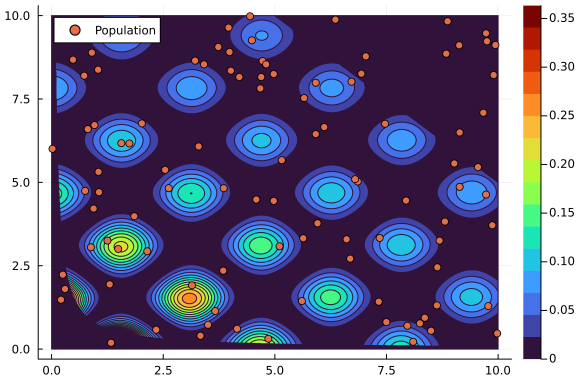
```
Generate Initial Population
Generate Random Velocities
Evaluate Initial Population
while Objective Function Evaluations < 10,000 do
    Update Population Velocities
    Divide Population Based on Fitness into LowPop and UpperPop
    if LowPop then
        Update LowPop Positions
    else
        Select Parents From UpperPop
        Breed New Population
        Mutate New Population
        Evaluate New Populations Fitness
    Evaluate New Populations Fitness
    Update Dissimilar Archive
Stop
```

When implementing this hybrid, the worst-performing 25% of the population has their positions updated via PSO, whilst the upper 75% is updated via the GA. The best solutions with dominant traits are therefore bred together to find a better solution, whilst the rest are encouraged to move towards the global best after exploring the solution space to a greater degree. Offspring that are bad solutions are automatically taken out of the breeding pool and begin moving back to better solutions.

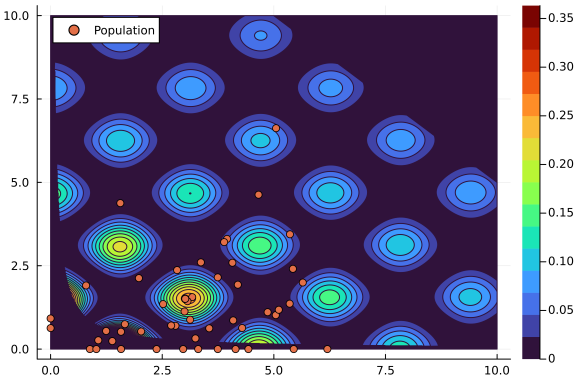
The hyper-parameters are identical to the independent tuned versions of GA and PSO, with only N tested for the GAPSO.

4.1 2-D KBF Results

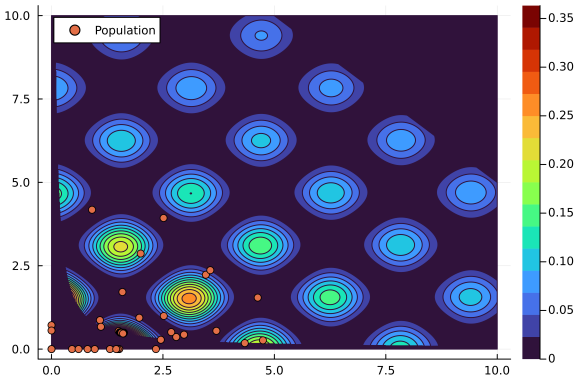
Using N = 100, from Figure 4 we observe that GAPSO can quickly move to the area containing the global maximum, and then explore much of the area surrounding this consistently as seen in Figure 4f.



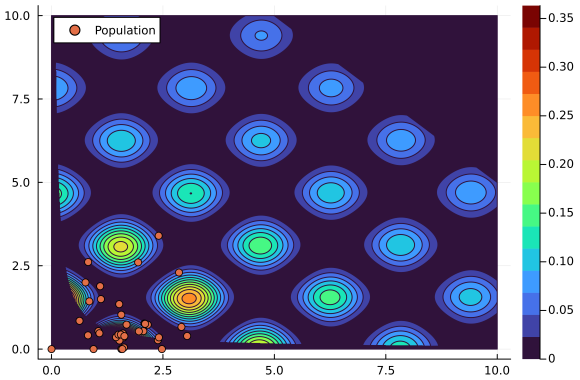
(a) Initialised Population



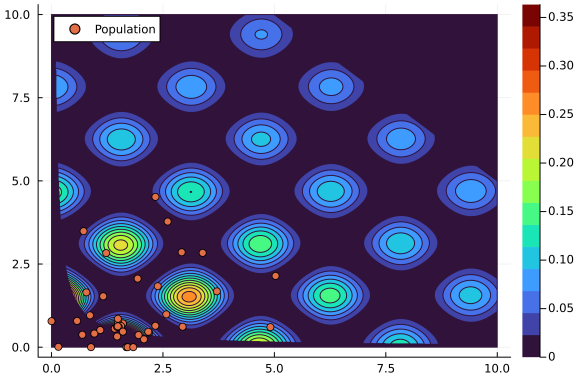
(b) Iteration 5



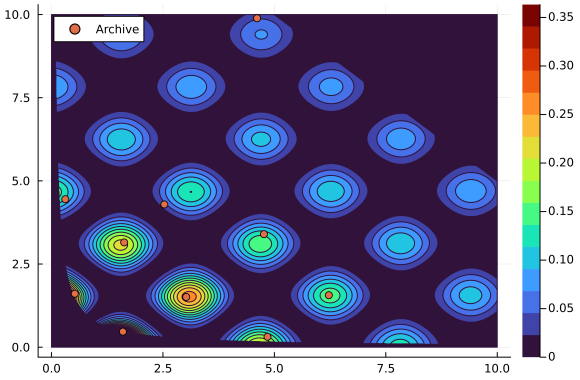
(c) Iteration 10



(d) Iteration 40



(e) Iteration 100 (final)



(f) Dissimilar Archive

Figure 4: Contour plot of the constrained 2-D KBF with a scatter plot of the GAPSO population at a given iteration.

4.2 8-D KBF Hyper-parameter Tuning and Results

The optimal population size was N = 400, which resulted in an average archive maximum fitness of $-\mathbf{0.6168}$ and a standard deviation of **0.07602**. This takes **88.64ms** on average with a standard deviation of **5.074ms**.

5 Discussion

Whilst the Genetic Algorithm is impressive due to its modularity, allowing one to customise crossover and parent selection in such a manner that suits the given problem, it is clear that in

a continuous setting, PSO dominates in every single manner. PSO is designed to operate in a continuous environment, which along with its ability to be highly vectorised, leads to greater consistency in finding a more optimal solution, as well as being able to do it approximately 6 times faster than the GA. This performance difference is displayed in Figure 5.

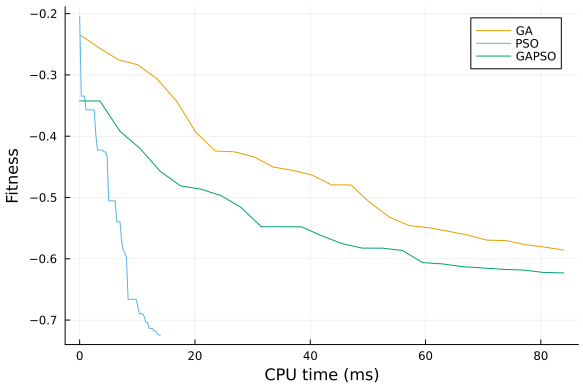


Figure 5: Fitness of the archive maximum vs CPU time for GA, PSO and GAPSO with 10,000 objective function calls.

	Archive Max.		Time Taken (ms)	
	Avg.	Std.	Avg.	Std.
Genetic Algorithm	-0.5614	0.08188	84.49	6.608
Particle Swarm Optimisation	-0.6839	0.05253	14.66	1.719
GAPSO	0.6168	0.07602	88.64	5.074

Table 4: Comparision of the performance of GA, PSO and GAPSO.

From Table 4, we can see that just updating the bottom 25% of the solutions in GAPSO with PSO, considerably increases the performance compared to GA alone. Unfortunately due to the vastly superior performance of PSO compared to GA, we are unable to observe whether GAPSO can surpass both by using the strengths of both algorithms. Further research on a more optimal GA implementation may allow this; for example, the use of temporal hyper-parameters, as seen in PSO, could be applied to GA; the mutation noise could be increased at the beginning and then drop as iteration count increases, for example.

6 References

[1] K. Deb and R. B. Agrawal, “Simulated binary crossover for continuous search space,” Complex systems, vol. 9, no. 2, pp. 115–148, 1995.

7 Code

7.1 Miscellaneous Functions (Misc.jl)

```
1 using Printf
2 using Statistics, StatsBase, Distributions
3 using LinearAlgebra
```

```
4
5  include("KBFunc.jl")
6
7
8
9  function score_top10(f::Vector{Float64})::Float64
10     """
11     Takes in the fitness of all the solutions
12     and returns the top 10 soliutions' average
13     fitness.
14     """
15     return -sum(partialsort(f , 1:10, rev=true)) / 10
16 end
17
18
19 function score_top1(f::Vector{Float64})::Float64
20     """
21     Takes in the fitness of the all solutions
22     and returns the fitness of the best
23     soltuino.
24     """
25     return -sum(partialsort(f , 1:1, rev=true))
26 end
27
28
29
30 function ranking(f::Vector{Float64})::Array{Int}
31     """
32     Returns the rankings of the solutions based on their fitness.
33     """
34     ordered_f = sort(f , rev=true)
35     f_ranks = Int[]
36     for i in f
37         rank = findnext(x -> x==i, ordered_f, 1)
38         while rank in f_ranks
39             rank = findnext(x -> x==i, ordered_f, rank + 1)
40         end
41         push!(f_ranks, rank)
42     end
43     return f_ranks
44 end
45
46
47 function update_archives(pos_archive, val_archive, val, pos)
48     """
49     Updates the dissimilar archive based on the incoming solutions
50     based on their postions and scores.
51     """
52     norms = sum(norm.(pos_archive .- pos'), dims=2)
53     D_min = 2
54     D_sim = 0.2
55
56     if any(x-> x<val, val_archive) && all(x-> x>D_min ,norms)
57         val_archive[argmin(val_archive)] = val
58         pos_archive[argmin(val_archive)[1], :] = pos
59     elseif all(x-> x<val, val_archive)
60         val_archive[argmin(norms)] = val
61         pos_archive[argmin(norms)[1], :] = pos
62     else
63         loc = findfirst(x->x==1, (norms .< D_sim) .& (val_archive .< val))
```

```
64          if loc !== nothing
65              val_archive[loc[1]] = val
66              pos_archive[loc[1], :] = pos
67          end
68      end
69      return pos_archive, val_archive
70  end
71
72  function contscatplot(pos, range, objfunc, label::String, plots::Bool)
73      """
74      Used to plot the contours with the current iterations populations
75      on top.
76      """
77      if plots
78          contourf(range, range, objfunc, camera=(180, 30), color=:turbo, dpi=800)
79          if label == "archive"
80              scatter!(Tuple.(pos), label="Archive")
81          else
82              scatter!(Tuple.(pos), label="Population")
83          end
84          savefig("Figures\\iter_" * label * ".png")
85      end
86  end
```

7.2 Problem Definition (KBFunc.jl)

```
1   using Printf
2   using Statistics
3
4
5   function constraint(x::Vector{Float64})::Bool
6       """
7       Used to confirm if a solution violates the problem constraints.
8       """
9       dim = length(x)
10      return all(x-> 0<=x<=10, x) && (sum(x) < 7.5*dim) && (prod(x) > 0.75)
11  end
12
13
14  function KBF(points::Union{Matrix{Float64}, Vector{Vector{Float64}}})::Vector{
        Float64}
15      """
16      Inputs a the solutions (in either a vector or matrix) and evaluates
17      the Keane's Bump Function at the given locations.
18      """
19      cost = Float64[]
20      if typeof(points) == Vector{Vector{Float64}}
21          points = mapreduce(permutedims, vcat, points)
22      end
23      for x in eachrow(points)
24          x = vec(copy(x))
25          if constraint(x)
26              cosargs =  cos.(x)
27              num =  sum(cosargs .^4) - 2prod(cosargs .^2)
28              denom = sqrt(sum([index*num^2 for (index, num) in enumerate(x)]))
29              push!(cost, num/denom)
30          else
31              push!(cost, 0)
32          end
33      end
```

```
    return cost
end
```

7.3 Genetic Algorithm (GeneticAlgo.jl)

```
using Statistics, StatsBase, Distributions, LinearAlgebra

include("KBFunc.jl")
include("Misc.jl")


mutable struct GA_Popul
    pop_size::Int
    pop_dim::Int
    positions::Vector{Vector{Float64}}
    scores::Vector{Float64}

    """
    Struct for containing information regarding the population
    from iteration to iteratino.
    the function within are for initilising the struct under
    different scenerios
    """

    function GA_Popul(pop_size, pop_dim)
        positions = pop_initial(LinRange(0, 10, 1000), pop_size, pop_dim)
        scores = KBF(positions)
        return new(pop_size, pop_dim, positions, scores)
    end

    function GA_Popul(pop_size, pop_dim, positions)
        scores = KBF(positions)
        return new(pop_size, pop_dim, positions, scores)
    end

    function GA_Popul(pop_size, pop_dim, positions, scores)
        return new(pop_size, pop_dim, positions, scores)
    end
end


function pop_initial(range, pop_size::Int, dim::Int)::Vector{Vector{Float64}}
    """
    Intialises the population within the constrained area.
    """
    pops = Vector{Vector{Float64}}()
    while length(pops) < pop_size
        sample = vec(rand(range, (dim,)))
        if constraint(sample)
            push!(pops, sample)
        end
    end
    return pops
end


function prop_Psi(f::Vector{Float64})::Vector{Float64}
    """
    Using the fitness of the solutions, calculates
    P_si for each solution.
```

```
56      """
57      f_   = sum(f)
58      prob = f ./ f_
59      return prob
60  end
61
62
63  function rank_Psi(f::Vector{Float64})::Vector{Float64}
64      """
65      Creates a distribution for drawing each solution
66      based on rank, as opposed to fitness.
67      """
68      s = 2
69      n = length(f)
70      r_i = ranking(f)
71      num = @. s * (n + 1. - 2. * r_i) + 2. * (r_i - 1.)
72      denom = n * (n - 1.)
73      prob = num ./ denom
74      return prob
75  end
76
77
78  function srswr(e_i, pop_size)
79      """
80      Returns a parent based on the expected number (e_i).
81      e_i is also returned with the index of the selected
82      parent reduced by one.
83      """
84      if all(x-> x<1, e_i)
85          parent_f = sample(1:pop_size, Weights(e_i), 1)[1]
86      else
87          parent_f = sample(1:pop_size, 1)[1]
88          while e_i[parent_f] < 1
89              parent_f = sample(1:pop_size, 1)[1]
90          end
91          e_i[parent_f] -= 1
92      end
93      return parent_f
94  end
95
96
97  function roulette_parents(f::Vector{Float64}, prob_method)::Vector{Vector{Int64
        }}
98      """
99      Selects N parents from the solutions f, based on the supplied probability
100     distribution (rank_psi or prop_psi).
101     """
102     p_si = prob_method(f)
103     parents = sample(1:length(f), Weights(p_si), (2,50))
104     parents = [[a,b] for (a,b) in eachcol(parents)]
105     return parents
106 end
107
108
109 function tournament_parents(f::Vector{Float64}, sub_size)::Vector{Vector{Int64}}
110     """
111     Samples a subset of the solutions randomly, and returns the best two
112     solutions as a parent pair.
113     """
114     pop_size = length(f)
```

```
115     parents = Vector{Vector{Int64}}()
116     while (length(parents) < floor(pop_size/2))
117         sub_pop_f = sample(f, sub_size, replace=false)
118         parent1_f, parent2_f = partialsort(sub_pop_f , 1:2, rev=true)
119         parent1 = findfirst(parent -> parent == parent1_f, f)
120         parent2 = findfirst(parent -> parent == parent2_f, f)
121         push!(parents, [parent1, parent2])
122     end
123     return parents
124 end
125
126
127 function srsw_parents(f::Vector{Float64})::Vector{Vector{Int64}}
128     """
129     Uses srswr() to select N parents.
130     """
131     pop_size = length(f)
132     parents = Vector{Vector{Int}}()
133     e_i = rank_Psi(f) .* pop_size
134     while (length(parents) < floor(pop_size/2))
135         parent11 = srswr(e_i, pop_size)
136         parent22 = srswr(e_i, pop_size)
137         pair = [parent11, parent22]
138         push!(parents, pair)
139     end
140     return parents
141 end
142
143
144 function locus_crossover(bit1::Char, bit2::Char, pos::Int, locus::Int)::Tuple{
    Char, Char}
145     """
146     Swaps the bits for one-point crossover.
147     """
148     if pos <= locus
149         bit1, bit2 = bit2, bit1
150     end
151     return bit1, bit2
152 end
153
154 function loci_crossover(bit1::Char, bit2::Char, pos::Int, loci1::Int, loci2::Int
    )::Tuple{Char, Char}
155     """
156     Swaps the bits for two-point crossover.
157     """
158     if (loci1 < pos <= loci2)
159         bit1, bit2 = bit2, bit1
160     end
161     return bit1, bit2
162 end
163
164 function rand_crossover(bit1::Char, bit2::Char)::Tuple{Char, Char}
165     """
166     Randomly the bits for random crossover if the bits aren't identical
167     """
168     if bit1 != bit2
169         temp = rand(0:1)
170         bit1 = only(string(temp))
171         bit2 = only(string(1 - temp))
172     end
```

```
173     return bit1, bit2
174 end
175
176
177 function bin_breed(parent1::Vector{Float64}, parent2::Vector{Float64}, p_c::
        Float64
178                     )::Tuple{Vector{Float64}, Vector{Float64}}
179     """
180     Loops through the bits in each parent pair and performs binary crossover.
181     returns the offspring.
182     """
183     if sample([0, 1], Weights([1-p_c, p_c])) == 1
184         dim = length(parent1)
185         child1, child2 = String["" for i in 1:dim], String["" for j in 1:dim]
186         child1_final, child2_final = Vector{Float64}(undef, dim), Vector{Float64
                }(undef, dim)
187         for (index, (parent1_x, parent2_x)) in enumerate(zip(parent1, parent2))
188             locus1 = rand(2:50)
189             locus2 = rand((locus1+1):64)
190             for (pos, (bit1, bit2)) in enumerate(zip(bitstring(parent1_x)[2:end
                    ], bitstring(parent2_x)[2:end]))
191                 bit1, bit2 = rand_crossover(bit1, bit2)
192                 #bit1, bit2 = locus_crossover(bit1, bit2, pos, locus1)
193                 #bit1, bit2 = loci_crossover(bit1, bit2, pos, locus1, locus2)
194                 child1[index] = child1[index] * bit1
195                 child2[index] = child2[index] * bit2
196             end
197             child1_final[index] = reinterpret(Float64, parse(Int, child1[index],
                     base=2))
198             child2_final[index] = reinterpret(Float64, parse(Int, child2[index],
                     base=2))
199         end
200         return child1_final, child2_final
201     else
202         return parent1, parent2
203     end
204 end
205
206
207 function cont_breed(parent1::Vector{Float64}, parent2::Vector{Float64}, p_c::
        Float64
208     )::Tuple{Vector{Float64}, Vector{Float64}}
209     """
210     Breeds parent pairs together using Simulated Binary Crossover.
211     """
212     if sample([0, 1], Weights([1-p_c, p_c])) == 1
213         dim = length(parent1)
214         u = rand(0:0.001:1, dim)
215         beta = Vector(undef, dim)
216         beta[u .<= 0.5] = (2 .* u[u .<= 0.5]) .^ (1/3)
217         beta[u .> 0.5] = (0.5 ./ (1 .- u[u .> 0.5])) .^(1/3)
218         child1_final = 0.5 .* ((1 .+ beta) .* parent1 + (1 .- beta) .* parent2)
219         child2_final = 0.5 .* ((1 .- beta) .* parent1 + (1 .+ beta) .* parent2)
220         return child1_final, child2_final
221     else
222         return parent1, parent2
223     end
224 end
225
226
```

```
227 function cont_mutate(offspring::Vector{Float64}, p_m::Float64)::Vector{Float64}
228     """
229     Adds gaussian to a chosen solution (Mutation).
230     """
231     if sample(1:2, Weights([1.0-p_m, p_m])) == 2
232         offspring += 2 .* randn(length(offspring)) #rand(0.:0.0001:10., length(
                offspring))
233     end
234     return offspring
235 end
236
237
238 function single_iteration(popu::GA_Popul, p_c::Float64, p_m::Float64)::GA_Popul
239     """
240     Helper functino that performs a single iteration.
241     Returns the new generation.
242     Parent selection -> Crossover -> Mutation
243     """
244     selected_parents_indices = tournament_parents(popu.scores, 25)
245     #selected_parents_indices = srsw_parents(popu.scores)
246     #selected_parents_indices = roulette_parents(popu.scores, prop_Psi)
247     new_pop = Vector{Vector{Float64}}()
248     pair = 1
249     while length(new_pop) < popu.pop_size
250         if pair > length(selected_parents_indices)
251             pair = 1
252         end
253         parent1, parent2 = popu.positions[selected_parents_indices[pair]]
254         ofs1, ofs2 = cont_breed(parent1, parent2, p_c)
255         offspring1, offspring2 = cont_mutate(ofs1, p_m), cont_mutate(ofs2, p_m)
256         if constraint(offspring1) && (length(new_pop) < popu.pop_size)
257             push!(new_pop, offspring1)
258         end
259         if constraint(offspring2) && (length(new_pop) < popu.pop_size)
260             push!(new_pop, offspring2)
261         end
262         pair += 1
263     end
264     popu.positions = new_pop
265     popu.scores = KBF(new_pop)
266     return popu
267 end
268
269
270 function GA(dim::Int, pop_size::Int, p_c::Float64, p_m::Float64, plots::Bool)
271     """
272     Determines the number of iterations, performs the optimisation, and
273     creates the needed plots. Returns the archive best sol, top1 and top10
274     scores for each iteration
275     """
276     iterations = floor(10000/pop_size)
277     range = LinRange(0, 10, 1000)
278     if plots
279         objfunc = KBF(vec([[i,j] for i in range, j in range]))
280     else
281         objfunc = Vector()
282     end
283     popu = GA_Popul(pop_size, dim)
284     pos_archive, val_archive = rand(15.:15., 7, dim), rand(0.:0., 7,1)
285     archive_scorings = Float64[score_top1(popu.scores)]
```

```
286     ga1_scorings = Float64[score_top1(popu.scores)]
287     ga10_scorings = Float64[score_top10(popu.scores)]
288     contscatplot(popu.positions, range, objfunc, string(0), plots)
289     for iter in 1:iterations
290         popu = single_iteration(popu, p_c, p_m)
291         for (val, pos) in zip(popu.scores, popu.positions)
292             pos_archive, val_archive = update_archives(pos_archive, val_archive,
                    val, pos)
293         end
294         if (iter % 10 == 0) | (iter in 1:10)
295             contscatplot(popu.positions, range, objfunc, string(iter), plots)
296         end
297         push!(archive_scorings, score_top1(vec(val_archive)))
298         push!(ga1_scorings, score_top1(popu.scores))
299         push!(ga10_scorings, score_top10(popu.scores))
300     end
301     contscatplot(eachrow(pos_archive), range, objfunc, "archive", plots)
302     plot(0:iterations, [archive_scorings, ga1_scorings, ga10_scorings])
303     savefig("Figures\\f_sum.png")
304     return [archive_scorings, ga1_scorings[end], ga10_scorings[end]]
305 end
```

7.4 Particle Swarm Optimisation (ParticleSwarm.jl)

```
1  using Statistics, StatsBase
2  using LinearAlgebra
3
4  include("KBFunc.jl")
5  include("Misc.jl")
6
7
8  mutable struct Swarm_Popul
9      pop_size::Int
10     pop_dim::Int
11     innertia::Float64
12     phi_p::Float64
13     phi_g::Float64
14     innertia_ini::Float64
15     phi_p_ini::Float64
16     phi_g_ini::Float64
17     innertia_temp::Matrix{Float64}
18     cog_temp::Matrix{Float64}
19     social_temp::Matrix{Float64}
20     pos::Matrix{Float64}
21     vels::Matrix{Float64}
22     part_best_pos::Vector{Union{Matrix{Float64}, Vector{Float64}}}
23     sw_best_pos::Tuple{Vector{Float64}, Float64}
24     val::Vector{Float64}
25     pos_archive::Matrix{Float64}
26     val_archive::Matrix{Float64}
27     """
28     Struct for storing anything relevant for the swarm.
29     includes temporary variables in order to avoid increasing new allocations
30     with each iteration.
31     The function below constructs the struct using the required inputs.
32     """
33
34     function Swarm_Popul(pop_size, pop_dim, innertia, phi_p, phi_g)
35         innertia_ini, phi_p_ini, phi_g_ini = innertia, phi_p, phi_g
36         pos = rand(LinRange(0, 10, 1000), pop_size, pop_dim)
```

```
37          val = KBF(pos)
38          part_best_pos = [pos, val]
39          sw_best_pos = (pos[argmax(val), :], maximum(val))
40          vels = rand(LinRange(-3, 3, 1000), pop_size, pop_dim)
41          innertia_temp, cog_temp = Matrix{Float64}(undef, 1,1), Matrix{Float64}(
                undef, 1,1)
42          social_temp = Matrix{Float64}(undef, 1,1)
43          pos_archive = rand(15.:15., 10, pop_dim)
44          val_archive = rand(0.:0., 10,1)
45          return new(
46                      pop_size, pop_dim, innertia, phi_p, phi_g, innertia_ini,
                            phi_p_ini,
47                      phi_g_ini, innertia_temp, cog_temp, social_temp, pos, vels,
48                      part_best_pos, sw_best_pos, val, pos_archive, val_archive)
49      end
50  end
51
52
53  function update_velocity(swarm::Swarm_Popul)::Swarm_Popul
54      """
55      Swarm_popul method which updates the veolcity of each
56      partcile based on its previous velocity, current position,
57      best positions recorded and the swarms best recorded position.
58      Limited to [-3:3].
59      """
60      swarm.innertia_temp = swarm.innertia .* swarm.vels
61      swarm.cog_temp = swarm.phi_p .* rand(0:0.00001:1, swarm.pop_size, swarm.
            pop_dim) .*
62                          (swarm.part_best_pos[1] - swarm.pos)
63      swarm.social_temp = swarm.phi_g .* rand(0:0.00001:1, swarm.pop_size, swarm.
            pop_dim) .*
64                              (reshape(swarm.sw_best_pos[1], (1, swarm.pop_dim)) .-
                                    swarm.pos)
65      swarm.vels = max.(min.(swarm.innertia_temp  + swarm.cog_temp + swarm.
            social_temp, 3), -3)
66      return swarm
67  end
68
69
70  function update_positions(swarm::Swarm_Popul)::Swarm_Popul
71      """
72      Swarm_popul method which updates the position of the
73      solutions using their respective
74      velocities. Ensures this stays in [0:10].
75      """
76      swarm.pos = max.(min.(swarm.pos + swarm.vels, 10), 0)
77      swarm.val = KBF(swarm.pos)
78      return swarm
79  end
80
81  function update_pos_archives(swarm::Swarm_Popul)::Swarm_Popul
82      """
83      Swarm_popul method which updates the archives.
84      """
85      for (index, (val1, val2)) in enumerate(zip(swarm.val, swarm.part_best_pos
            [2]))
86          if val1 > val2
87              swarm.part_best_pos[1][index, :] = swarm.pos[index, :]
88          end
89          if val1 > swarm.sw_best_pos[2]
```

```
90              swarm.sw_best_pos = (swarm.pos[index, :], val1)
91          end
92          swarm.pos_archive, swarm.val_archive =
93           update_archives(swarm.pos_archive, swarm.val_archive, val1, swarm.pos[
                index, :])
94      end
95      return swarm
96  end
97
98
99  function update_hparams(swarm::Swarm_Popul, iters::Int)::Swarm_Popul
100     """
101     Swarm_popul method which updates the hyper-parameters each iteeration.
102     """
103     swarm.innertia = swarm.innertia - (0.5 * swarm.innertia_ini / iters)
104     swarm.phi_p = swarm.phi_p - (0.5 * swarm.phi_p_ini / iters)
105     swarm.phi_g = swarm.phi_g + (swarm.phi_g_ini / iters)
106     return swarm
107 end
108
109
110 function PSO(dim::Int, pop_size::Int, innertia::Float64, phi_p::Float64,
111             phi_g::Float64, plots::Bool)
112     """
113     Function which generates the struct, and performs each iteration
114     (the number is calculated from pop size), creates plots and
115     returns the archive max, top1 and top10 avgs for each iteration.
116     """
117     iterations = floor(Int, 10000/pop_size)
118     range = LinRange(0, 10, 1000)
119     if plots
120         objfunc = KBF(vec([[i,j] for i in range, j in range]))
121     else
122         objfunc = Vector()
123     end
124
125     swarm = Swarm_Popul(pop_size, dim, innertia, phi_p, phi_g)
126     archive_scorings = Float64[score_top1(swarm.val)]
127     swarm1_scorings = Float64[score_top1(swarm.val)]
128     swarm10_scorings = Float64[score_top10(swarm.val)]
129     contscatplot(eachrow(swarm.pos), range, objfunc, string(0), plots)
130     for iter in 1:iterations
131         swarm = update_velocity(swarm)
132         swarm = update_positions(swarm)
133         swarm = update_pos_archives(swarm)
134         swarm = update_hparams(swarm, iterations)
135
136         push!(archive_scorings, score_top1(vec(swarm.val_archive)))
137         push!(swarm1_scorings, score_top1(swarm.val))
138         push!(swarm10_scorings, score_top10(swarm.val))
139
140         if (iter % 10 == 0) | (iter in 1:10)
141             contscatplot(eachrow(swarm.pos), range, objfunc, string(iter), plots
                    )
142         end
143     end
144     contscatplot(eachrow(swarm.pos_archive), range, objfunc, "archive", plots)
145     plot(0:iterations, [archive_scorings, swarm1_scorings, swarm10_scorings])
146     savefig("Figures\\f_sum.png")
147
```

```
148     return [archive_scorings, swarm1_scorings[end], swarm10_scorings[end]]
149 end
```

7.5 GAPSO (HybridGAPSO.jl)

```
 1 using Printf
 2 using Statistics, StatsBase, Distributions
 3 using Plots
 4 using BenchmarkTools
 5 using LinearAlgebra
 6 using Random
 7
 8 include("KBFunc.jl")
 9 include("GeneticAlgo.jl")
10 include("Misc.jl")
11 include("ParticleSwarm.jl")
12
13
14 function SGAPSO(
15                 dim::Int, pop_size::Int, innertia::Float64, phi_p::Float64,
16                 phi_g::Float64, p_c::Float64, p_m::Float64, plots::Bool
17                 )
18     """
19     Performs hybrid GA and PSO. the population experiences each
20     process in series, alternating every 5 iterations.
21     (This was not used in the report).
22     """
23     iterations = floor(Int, 10000/pop_size)
24     range = LinRange(-2, 12, 1000)
25     if plots
26         objfunc = KBF(vec([[i,j] for i in range, j in range]))
27     else
28         objfunc = Vector()
29     end
30
31     swarm = Swarm_Popul(pop_size, dim, innertia, phi_p, phi_g)
32     gapopu = GA_Popul(pop_size, dim)
33     swarm.pos = mapreduce(permutedims, vcat, gapopu.positions)
34
35     archive_scorings = Float64[score_top1(vec(swarm.val_archive))]
36     swarm1_scorings = Float64[score_top1(swarm.val)]
37     swarm10_scorings = Float64[score_top10(swarm.val)]
38     contscatplot(eachrow(swarm.pos), range, objfunc, string(0), plots)
39
40     for iter in 1:iterations
41         if 0 < iter % 10 <= 5
42             swarm = update_velocity(swarm)
43             swarm = update_positions(swarm)
44             swarm = update_pos_archives(swarm)
45             swarm = update_hparams(swarm, iterations)
46             vec_pos = [row[:] for row in eachrow(swarm.pos)]
47             gapopu.positions = vec_pos
48             gapopu.scores = swarm.val
49         else
50             swarm = update_velocity(swarm)
51             gapopu = single_iteration(gapopu, p_c, p_m)
52             swarm.pos = mapreduce(permutedims, vcat, gapopu.positions)
53             swarm.val = gapopu.scores
54             swarm = update_pos_archives(swarm)
55             swarm = update_hparams(swarm, iterations)
```

```
56          end
57
58          push!(archive_scorings, score_top1(vec(swarm.val_archive)))
59          push!(swarm1_scorings, score_top1(swarm.val))
60          push!(swarm10_scorings, score_top10(swarm.val))
61
62          if (iter % 10 == 0) | (iter in 1:10)
63              contscatplot(eachrow(swarm.pos), range, objfunc, string(iter), plots
                    )
64          end
65      end
66
67      contscatplot(eachrow(swarm.pos_archive), range, objfunc, "archive", plots)
68      plot(0:iterations, [archive_scorings, swarm1_scorings, swarm10_scorings])
69      savefig("Figures\\f_sum.png")
70
71      return [archive_scorings[end], swarm1_scorings[end], swarm10_scorings[end]]
72  end
73
74
75  function PGAPSO(
76      dim::Int, pop_size::Int, innertia::Float64, phi_p::Float64,
77      phi_g::Float64, p_c::Float64, p_m::Float64, plots::Bool
78      )
79      """
80      Performs GAPSO in parallel by splitting up the population based
81      on their fitness. Creates the required plots for evaluation, as
82      well as returning the archive max, top1 and top10 avgs for each
83      iteration.
84      """
85      iterations = floor(Int, 10000/pop_size)
86      range = LinRange(0, 10, 1000)
87      if plots
88          objfunc = KBF(vec([[i,j] for i in range, j in range]))
89      else
90          objfunc = Vector()
91      end
92
93      swarm = Swarm_Popul(pop_size, dim, innertia, phi_p, phi_g)
94      archive_scorings = Float64[score_top1(vec(swarm.val_archive))]
95      swarm1_scorings = Float64[score_top1(swarm.val)]
96      swarm10_scorings = Float64[score_top10(swarm.val)]
97      contscatplot(eachrow(swarm.pos), range, objfunc, string(0), plots)
98
99      for iter in 1:iterations
100
101         sampleindex = sortperm(swarm.val, rev=false)
102         split = floor(Int, pop_size * 0.25)
103         swarm = update_velocity(swarm)
104
105         # Updates the top 75% of solutions using GA
106         vec_pos = [row[:] for row in eachrow(swarm.pos[sampleindex[(split+1):
                pop_size], :])]
107         gapopu = GA_Popul(floor(Int,pop_size* 0.75), dim, vec_pos, swarm.val[
                sampleindex[(split+1):pop_size]])
108         gapopu = single_iteration(gapopu, p_c, p_m)
109
110         # Updates the bottom 25% of solutions using PSO
111         swarm.pos[sampleindex[(split+1):pop_size], :] = mapreduce(permutedims,
                vcat, gapopu.positions)
```

```
112         swarm.pos[sampleindex[1:split], :] = max.(min.(swarm.pos[sampleindex[1:
                split], :] + swarm.vels[sampleindex[1:split], :], 10), 0)
113
114         # Updates solutions scores, archives and hyper-parameters
115         swarm.val = KBF(swarm.pos)
116         swarm = update_pos_archives(swarm)
117         swarm = update_hparams(swarm, iterations)
118
119         push!(archive_scorings, score_top1(vec(swarm.val_archive)))
120         push!(swarm1_scorings, score_top1(swarm.val))
121         push!(swarm10_scorings, score_top10(swarm.val))
122
123         if (iter % 10 == 0) | (iter in 1:10)
124             contscatplot(eachrow(swarm.pos), range, objfunc, string(iter), plots
                    )
125         end
126     end
127
128     contscatplot(eachrow(swarm.pos_archive), range, objfunc, "archive", plots)
129     plot(0:iterations, [archive_scorings, swarm1_scorings, swarm10_scorings])
130     savefig("Figures\\f_sum.png")
131     return [archive_scorings, swarm1_scorings[end], swarm10_scorings[end]]
132 end
```