

Module	4M17	Title of report	Assignment 2: Optimisation Algorithm Performance Comparison Study
UNDERGRADUATE STUDENTS ONLY		Assessment for this module is <input checked="" type="checkbox"/> 100% / <input type="checkbox"/> 25% coursework of which this assignment forms 50%	
Candidate number:			
MPHIL STUDENTS ONLY		Date submitted:	
Candidate number:			
OTHER POSTGRADUATE STUDENTS			
Name:			
CRSid:			

Feedback to the student

☐ See also comments in the text

		Very good	Good	Needs imprvmt
C O N T E N T	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:			
P R E S E N T A T I O N	Attention to detail, typesetting and typographical errors Is the report free of typographical errors? Are the figures/tables/references presented professionally?			
	Comments:			

Marker:

Date:

4M17 Coursework #2

Optimisation Algorithm Performance Comparison

Candidate No: 5730E

December 19, 2023

1 Introduction

This report conducts a comparative analysis of two optimisation algorithms applied to minimise Keane's Bump Function, (KBF). In particular, the study focuses on a Continuous Genetic Algorithm, (CGA), as well as an alternative algorithm not covered in the lectures: Parallel Tempering, (PT). The initial sections involve fine-tuning the algorithms on the two-dimensional KBF to enhance their performance and investigate their respective capabilities. Following this, the algorithms are deployed on the eight-dimensional KBF, and a comprehensive performance evaluation is conducted to discern differences and similarities between them.

2 Keane's Bump Function

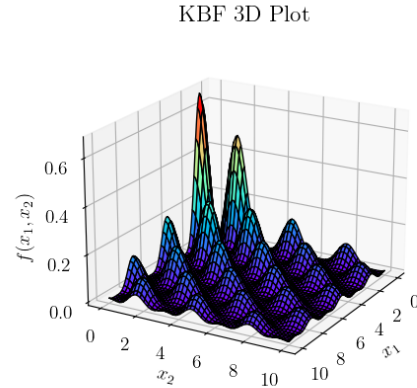
To compare the performances of the two algorithms, the Keane's Bump Function, (KBF), is used as the objective function. In particular, the n-dimensional constrained optimisation problem is defined as the maximisation of:

$$f(\mathbf{x}) = \left| \frac{\sum_{i=1}^n (\cos(x_i))^4 - 2 \prod_{i=1}^n (\cos(x_i))^2}{\sqrt{\sum_{i=1}^n i \cdot x_i^2}} \right| \quad (1)$$

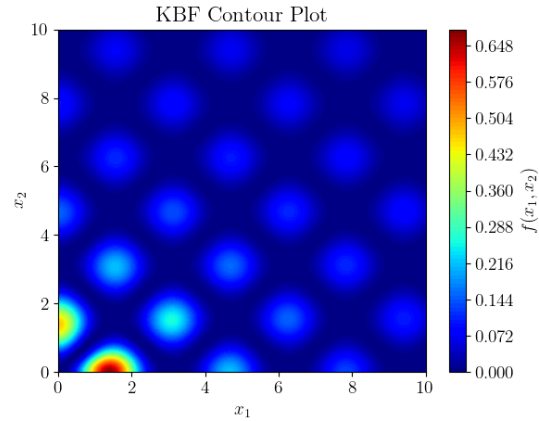
subject to $0 \leq x_i \leq 10 \quad \forall i \in \{1, \dots, n\}$

$$\begin{aligned} \prod_{i=1}^n x_i &> 0.75 \\ \sum_{i=1}^n x_i &< \frac{15n}{2} \end{aligned} \quad (2)$$

The two-dimensional form of the function has been plotted in Figure 1. Some notable properties are as follows:



(a) Surface plot.



(b) Contour plot.

Figure 1: Two-dimensional visualisation of the Keane's Bump Function, (KBF).

- The function is undefined at the origin, (0, 0). This is due to the division by zero in the denominator of Equation (1). Otherwise, the function is continuous and differentiable everywhere.
- The function is highly multi-modal. Its global maximum is located on the constraint boundary $x_n = 0$, where x_n denotes the final variable in the n-dimensional space. However, there are many

local maxima located inside the feasible region, all of which have quite similar amplitudes.

- The function is nearly symmetric about the line $x_1 = x_2$. This stems from its construction in (1), using the sums of squared, symmetric terms, x_i^2 , $(\cos(x_i))^2$, and $(\cos(x_i))^4$. This results in some invariance regarding the order of the input variables. Overall, the peaks consistently manifest in pairs, yet there is a notable pattern wherein one peak always surpasses its counterpart in magnitude.

Given the above properties, the KBF is a challenging function to optimise. The presence of multiple, similar-amplitude local maxima makes it difficult for an optimisation algorithm to converge to the global maximum. On the other hand, all control variables share the same nature, (continuous variables), and exhibit identical scales. Additionally, all constraints are of the inequality type, and the feasible space is non-disjoint.

The problem becomes more complicated with the inclusion of the constraints outlined in (2). Figure 2 illustrates the resulting feasible region carved out of the original function space. Notably, the constraint boundaries are non-linear. The problem complexity is additionally exacerbated by the presence of multiple optima along the constraint boundaries, including the global maxima that we seek to identify.

These properties make the KBF a suitable candidate for the comparative analysis of the two optimisation algorithms, as discussed in the previous work of [3].

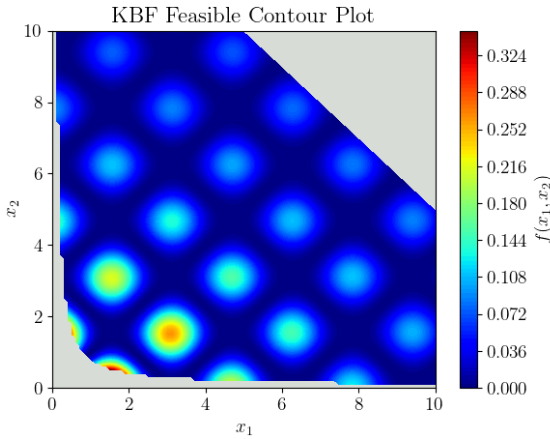


Figure 2: Feasible region carved out from the two-dimensional visualisation of the Keane's Bump Function, (KBF).

3 The Continuous Genetic Algorithm (CGA)

The discrete nature of the GA presented in [9] makes it unsuitable for the optimisation of the KBF. An implementation of a Continuous Genetic Algorithm, (CGA), is used instead, which lends itself better to the problems presented in (1)-(2).

The CGA, a technique inspired by natural selection and genetics, presents itself as particularly well-suited to tackling challenges associated with multiple local optima. Furthermore, the algorithm lends itself well to parallelisation with low implementation effort, and offers ample opportunities for modifications and adaptations, supported by a rich body of literature on the subject.

The primary difference between the CGA and the GA in [9] is the representation of individuals, (solutions of the state space), within the population. Rather than representing an individual as a vector of binary values or bits, (0s and 1s), the CGA uses a real-valued vector of floating-point numbers to represent each individual, as discussed in [4]. This allows for a direct representation of the problem, and eliminates the need for a decoding function, which reduces overhead in function evaluations.

This adjustment marks a significant departure from conventional GAs, aligning the algorithm more closely with Evolution Strategies (ES), another member of the evolutionary algorithms family presented in [12]. However, the algorithm presented in 3.1 is still classified as a GA in accordance with the differences presented in [5], given that mutation does not serve as the primary search mechanism for exploring the state space. Instead, it functions as a non-adaptive, background operator.

3.1 Implementation

In accordance with the terminology presented in [9], a vector solution of the state space will be referred to as an *individual* or *chromosome*. Correspondingly, a collection of such individuals arranged in a matrix format will be denoted as a *population*. Each individual is delineated as an $n \times 1$ vector of real-valued (floating-point) numbers, where n signifies the number of variables in the state space. The population itself is represented as a $m \times n$ matrix, where m designates the count of individuals in the population. This count is explicitly defined as a hyperparameter within the code, and is referred to as *POPULATION_SIZE*.

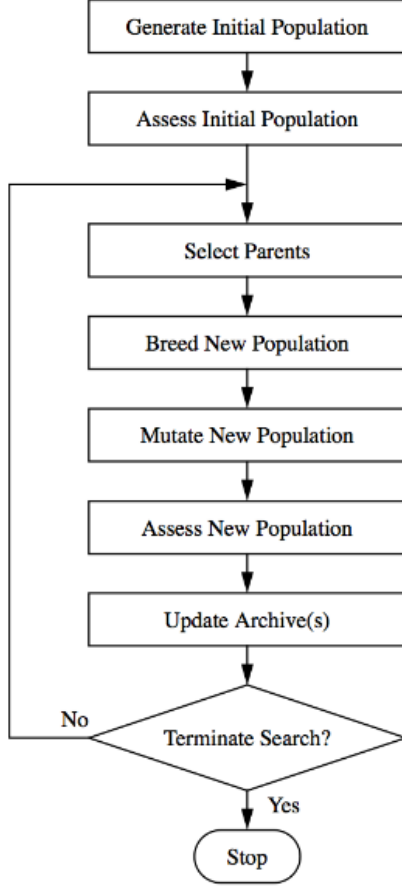


Figure 3: A flowchart depicting the CGA process, taken from [9].

The CGA process is outlined in Figure 3. Notably, three selection strategies and two mutation procedures were deliberately implemented to harness the flexibility inherent in the CGA, tailoring it to the optimisation challenges posed by the KBF. The subsequent section, 3.2, elucidates the selection method and mutation procedure choices that form the basis of the forthcoming comparison.

The CGA can be finely tuned for a specific objective function by modifying its fitness function, which assesses the quality of an individual. The management of constraints is woven into the selection process, as outlined later in Section 3.1.2.

3.1.1 Initialisation

The population is initialised with random values uniformly distributed within the range of 0 to 10, which represents the bounds of the state space, encompassing both feasible and infeasible values. This population comprises 250 individuals, as defined by the *POPULATION_SIZE* hyperparameter.

3.1.2 Parent Selection

Three selection methods were implemented: proportional selection, tournament selection, and stochastic remainder selection without replacement (SRS). The hyperparameter governing the quantity of parents chosen, denoted as *NUM_PARENTS*, is established at 25% of the population size, (rounded down to the nearest even number to facilitate the creation of parent pairs).

Another important aspect of the selection process is the handling of constraints. In particular, the selection process is repeated until only feasible individuals are selected. Infeasible individuals are simply not chosen as parents, a strategy advised by [9], which was deemed suitable considering that the feasible space is non-disjoint, and the constraints are of the inequality type.

Proportional Selection

The probability of an individual being selected for mating is proportional to its fitness:

$$P_i = \frac{f_i}{\sum_{i=1}^n f_i}$$

Here, f_i denotes the fitness of the i th individual. This is the simplest selection method, and is implemented in accordance with the theory presented in [9].

As noted in [9], this approach is susceptible to high variance in individual selection, primarily because there is no assurance of choosing the optimal individual. Alternatively, the following two procedures show greater potential, incorporating a degree of determinism into the selection process.

However, it would be premature to disregard the proportional selection method. There is a chance that the determinism inherent in the other two methods could negatively impact the KBF optimisation process. A notable degree of stochasticity may improve the algorithm’s effectiveness in exploring the search space, which may prove vital given the multi-modal nature of the KBF.

Tournament Selection

As outlined in [9], this strategy involves taking a small subset of the population, and selecting the top two individuals with the highest fitness. This is repeated until the required number of parents is achieved. Selection pressure can then be adjusted by varying the size of the subset, controlled by the *TOURNAMENT_SIZE* hyperparameter.

This is a popular selection method, as it is simple to implement, and is known to perform well in practice. It can be improved by including some of the concepts presented in [7], however, these have been avoided to

reduce the complexity of the algorithm's implementation.

Stochastic Remainder Selection without Replacement (SRS)

This strategy draws inspiration from [9]. Specifically, a group of chosen individuals is curated by generating an expected number of copies for each individual, denoted as:

$$E_i = N * P_i$$

Here, N represents the population size, and P_i is elucidated above in the context of proportional selection. The anticipated number of duplicates is subsequently divided into an integer part, $I_i = \lfloor E_i \rfloor$, and a remainder, $R_i = E_i - I_i$.

The integer part is used for deterministic selection of individuals. The i th individual is selected I_i times. Subsequently, the remained, R_i , is then used to stochastically augment the collection of individuals until the required number of parents is achieved. This is done by selecting the i th individual with a probability of R_i .

The discussion in [9] highlights that this approach appears to yield superior performance, ascribed to the inclusion of a degree of determinism in the selection criteria. Nevertheless, it remains worthwhile to evaluate the performance of the other two methods, as they might present distinct advantages within the framework of the KBF.

3.1.3 Mating Procedure

Similarly, two mating procedures have implemented: crossover and heuristic crossover. The entire population is replaced by offspring, bred from two randomly allocated parents from the pool of selected parents.

Crossover

The crossover procedure adheres to the principles detailed in [9]. Initially, a crossover point is randomly chosen. Genes from the first parent are incorporated into the offspring until this point, beyond which genes from the second parent take their place. Specifically, the sequencing of the parents is governed by the probability specified by the hyperparameter *CROSSOVER_PROB*.

Heuristic Crossover

Heuristic crossover is presented in [6]. By this variation, a random number β in the interval $[0, 1]$ is generated. The genes of the offspring are then determined as a blend of the original two parents, p_1 and p_2 , as follows:

$$o_i = \beta(p_{1i} - p_{2i}) + p_{2i}$$

With this inspiration in mind, heuristic crossover was implemented in the CGA. Specifically, the sequence of parents in the above formula is determined by the *CROSSOVER_PROB* hyperparameter, aligning with the approach used previously in the original crossover procedure.

A crucial factor to bear in mind is that certain offspring may be produced outside the feasible region. This implementation deviates from the recommendation in [6] by not outright rejecting these offspring during the mating procedure. Rather, they are simply excluded from consideration as parents in the subsequent selection process.

The decision to incorporate this mutation procedure stems from the continuous nature of the state space. While the conventional crossover methods may be well-suited to binary representations, a more intuitive approach emerges when grappling with real-valued variables - using a blend of characteristics from both parents.

Moreover, the use of the heuristic crossover method aims to address previous limitations associated with standard crossover. Unlike conventional crossover, which confines offspring values to those of the parents, blending permits the generation of offspring beyond the parent values, allowing for potentially new information. This feature may prove particularly advantageous in the context of the KBF, enhancing the algorithm's capability to navigate the search space adeptly and thoroughly explore local optima.

An additional noteworthy observation is that the introduction of β serves to bring the CGA into closer alignment with Evolutionary Strategies (ES). This resemblance becomes evident as the formulation above bears some similarity to the intermediate recombination strategy outlined in [12]. Nevertheless, it is essential to emphasise, as mentioned earlier, that mutation does not serve as the primary search mechanism in the CGA. Consequently, the CGA can still be appropriately categorised as a GA, as per the classification seen in [5].

3.1.4 Mutation

The mutation procedure proposed by [4] involves perturbing a gene within a chromosome by a normally distributed random number with a mean of zero and a standard deviation of σ . However, the effectiveness of this procedure is constrained by the selection of σ as an additional hyperparameter, necessitating careful tuning.

Instead, a simpler approach was adopted, where genes within the population have an probability, given by the

mutation rate, of being reset to a random value drawn from a uniform distribution within the confines of the state space, mirroring the initialisation procedure.

3.1.5 Evaluation

The population is then evaluated, and the individuals are ranked in order of fitness. Here, the fitness is defined as the negative of the KBF cost function, outlined in (1). This is done to align the algorithm with the maximisation of the KBF.

3.1.6 Termination

For this section of the report, the algorithm terminates after a maximum number of iterations, (defined as a hyperparameter). A more stringent convergence criterion is adopted for the comparison between the two algorithms in Section 5. The individual with the lowest fitness value is then returned as the optimal solution to the optimisation problem.

3.2 Tuning the CGA

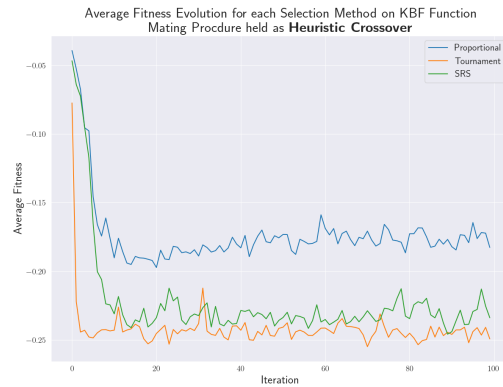
Given the flexibility of the CGA, multiple selection methods and mutation procedures were implemented within the codebase, as discussed previously. These are then selected as hyperparameters within the experiments presented in 10, to take advantage of the flexibility offered by the CGA.

To choose a selection method and mutation procedure going forwards into the main comparison of this report, code 10 was used as a platform for experimentation. Each selection method and mutation procedure was evaluated over 10 and 100 iterations, with a constant mutation rate of 0.05, and a crossover probability of 0.7. The number of parents was established at 25% of the population size, (rounded down to the nearest even number to facilitate the creation of parent pairs). The population size itself was set at 250, and the tournament size was similarly defined as 25% of this.

Supplementary results regarding this initial exploration of hyperparameter tuning are detailed in Section 6.1. Specifically, the final fitness values are outlined in Table 3. At a first glance, it would appear that all selection methods and mutation procedures perform similarly, offering convergences to similar fitness values. Minor variations could be attributed to the stochastic nature of the CGA, and the small number of iterations.



(a) Mutation Procedure: Crossover



(b) Mutation Procedure: Heuristic Crossover

Figure 4: Evolution of the average fitness values of the CGA population over 100 iterations for each of the selection methods. The results are presented for both mutation procedures: crossover and heuristic crossover. Here, the mutation rate was set to 0.05, and the crossover probability was set to 0.7.

However, more insightful conclusions can be drawn from the figures presented in 4, which illustrate the rapid evolution of the average fitness values of the CGA population over 100 iterations for each of the selection methods. They illustrate that tournament selection seemed to outperform the other two selection methods when maximising the KBF, which is surprising given the favourable description in [9] regarding SRS.

The experiment was repeated several times, yielding results that were admittedly variable, attributed to the inherent stochastic nature of the CGA. At times, the superiority of SRS over tournament selection was evident, yet proportional selection generally did not prove to perform better. Despite this variability, the outcomes exhibited sufficient consistency to justify the designation of tournament selection as the primary method for

the upcoming comparison.

The observed differences in performance may be attributed to the distinct characteristics of the KBF. Tournament selection appears to demonstrate greater efficacy in navigating the search space compared to SRS, perhaps due to its less deterministic nature. This becomes particularly significant in light of the multi-modal nature of the KBF, where a certain level of stochasticity may prove essential for thorough exploration of the search space.

The results additionally indicate that when combined with tournament selection, the heuristic crossover procedure generally outperformed the standard crossover method. This is evident in the typically steeper decline in average fitness values, as illustrated in Fig. 4b. Furthermore, the heuristic crossover exhibited greater result consistency across repeated experiments and generally converged to higher maximas compared to the standard crossover. Although the observed differences were not always significant, the overall trend supports the choice of heuristic crossover as the preferred mating procedure for the forthcoming comparison.

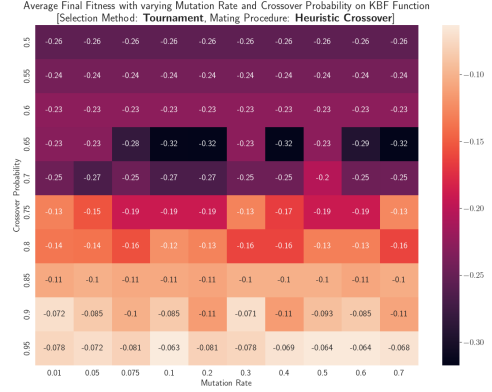
This may be attributed to the heuristic crossover’s capacity to generate offspring beyond the existing genotypes present within a generation. This attribute is especially beneficial within the framework of the KBF, enhancing the algorithm’s ability to navigate the search space adeptly and systematically explore the numerous local optima.

Having cemented the selection method and mutation procedure choices as tournament selection and heuristic crossover respectively, the CGA was further tuned to optimise its performance.

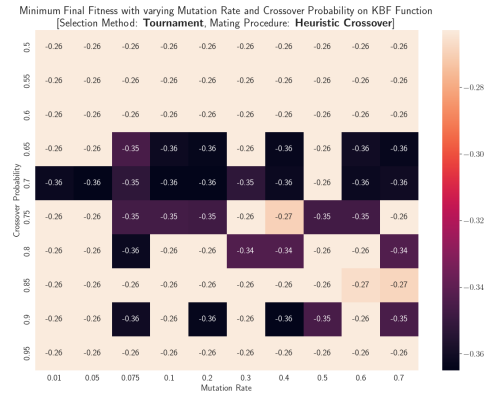
Specifically, choices for the mutation rate and crossover probability were explored. The results are presented in Figure 5, which showcases two heat maps of the final fitness values after 100 iterations for the CGA using the chosen procedures. The plots are presented for both the average final fitness, as well as the final fitness of the best individual.

Notably, as depicted in Fig. 5a, it is evident that an increased stochasticity of the CGA, indicated by higher crossover probabilities, leads to improved average final fitness. This aligns with expectations, as a broader and more randomly distributed population enhances the likelihood of individuals being situated in the proximity of local maxima. Consequently, a greater number of individuals within the population receive lower fitness values, a trend reflected in the observed average final fitness.

Fig. 5b offers deeper insights. It is evident that the crossover probability stands out as the most influential hyperparameter, while the fine-tuning of the mutation rate plays less of a role in shaping the performance of the CGA.



(a) Average final fitness across entire population.



(b) Final (minimum) fitness of best individual.

Figure 5: Heat maps of the final fitness values after 100 iterations for the CGA using tournament selection and heuristic crossover. Here, the mutation rates and crossover probabilities were varied to assess their impact on performance.

Overall, a clear optimal set of hyperparameters emerges. Moving forward, the mutation rate is fixed at 0.1, and the crossover probability is set to 0.65. These values have proven to yield a consistently optimal performance for the algorithm, comfortably lying within the darker regions of the heat maps.

To reiterate and summarise this section of the report, the CGA is now constructed as follows:

- **Selection Method:** Tournament Selection
- **Mutation Procedure:** Heuristic Crossover

- **Mutation Rate:** 0.1
- **Crossover Probability:** 0.65
- **Tournament Size:** 62
- **Number of Parents:** 62
- **Population Size:** 250

Fig. 6 depicts the evolution of the fitness values over 100 iterations using the optimal hyperparameters delineated above. Note that the (minimum) fitness attributed to the best individual remains constant, because an individual within the population is stochastically initialised near the local maxima that the algorithm converges to. This is less likely to occur in the 8-dimensional comparison, where the minimum fitness is expected to evolve more noticeably, given that the search space is significantly larger, and initialisation will be handled with more care.

Fig. 7 depicts the convergence of the population to the second largest maxima over 4 and 80 iterations using the optimally-tuned CGA. Note that the horizontally and vertically distributed outliers are attributed to crossover. Regrettably, global maximum convergence remained elusive even after 100 iterations. However,

there does seem to be a discernible trend of the algorithm converging towards the second-largest maxima in Fig. 7, offering a promising indication of progress.

The primary challenge lies in the fact that both the first and second largest maxima are situated along the non-linear constraint boundary of the optimisation problem, posing a difficult navigational challenge.

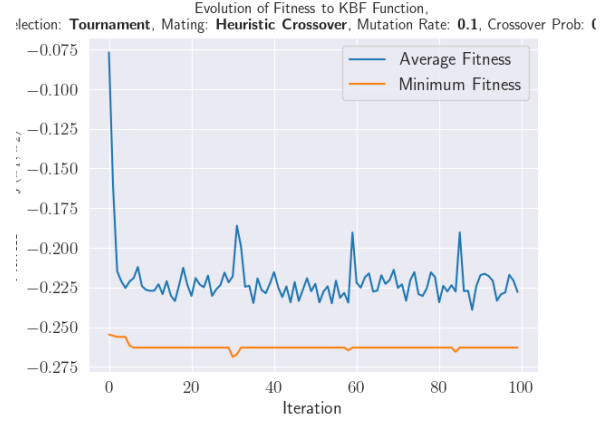


Figure 6: Evolution of the fitness values of the CGA population over 100 iterations for optimally-tuned CGA.

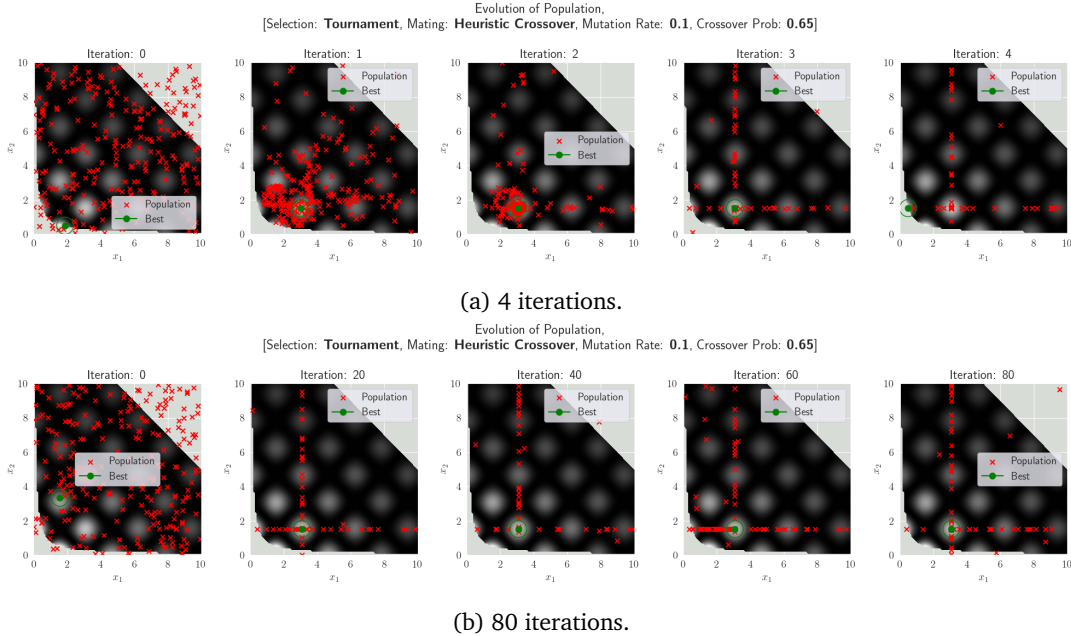


Figure 7: Evolution of the CGA population over 4 and 80 iterations using the optimal hyperparameters outlined at the end of Section 3.2. The second plot depicts the algorithm converging to a local maxima as a result of an individual being initialised near it.

4 Parallel Tempering (PT)

Having observed the CGA’s performance and its difficulty in identifying global maxima effectively, an informed decision was made to introduce Parallel Tempering (PT) as the second algorithm for comparison.

This is a Markov Chain Monte Carlo (MCMC) optimisation approach presented in [2]. Similar to simulated annealing, (SA), it revolves around the idea of smoothening the search space by introducing a temperature parameter, t .

By incorporating a Metropolis-Hasting acceptance probability that is proportional to $\exp(1/t)$, exploration of the solution space is encouraged at high temperatures, while exploitation of the local nature of the function is facilitated at low temperatures. At high temperatures, the acceptance distribution is smoothed out by this proportionality, and the algorithm is then able to explore the search space more thoroughly. As the temperature diminishes, the acceptance distribution becomes more dependent on the fitness of the proposed solution to the KBF, enabling the algorithm to explore the local nature of the function.

However, PT incorporates the novel concept of Replica Exchange Monte Carlo. Unlike SA, which operates along a single trajectory, starting with a high temperature that gradually decreases, PT maintains multiple replicas of the system at different temperatures simultaneously. Each replica explores its own solution space, and exchanges of the solutions can occur between replicas at adjacent temperature levels. This exchange mechanism enhances global exploration by facilitating the movement of solutions between replicas operating at high and low temperature.

This parallel exploration across temperatures improves the algorithm’s ability to escape local optima, providing a valuable alternative to traditional single-trajectory methods like SA. Consequently, PT could emerge as a well-suited approach for maximising the highly multi-modal KBF.

Unfortunately, this introduces an elevated computational cost, due to the management of multiple replicas and the periodic exchange of solutions. However, the potential advantages in terms of global exploration of the KBF outweigh the added computational expense. This is especially true, considering the highly parallelisable nature of PT, which may allow for even more efficient utilisation of parallel computing resources than the CGA.

4.1 Implementation

The discourse presented in [2] emphasises a delicate trade-off between optimising MCMC sampling outcomes and minimising computational efforts. Hence, the PT implementation presented here is deliberately crafted for flexibility, offering the ability to fine-tune the algorithm, as shown in Section 4.2.

As per the guidance in [10], constraints are handled by simply rejecting any proposed changes that violate the constraints. This is accomplished within the Metropolis-Hastings criterion regarding the acceptance or rejection of a proposed change. This was determined as a suitable approach, given that the feasible space is non-disjoint, and the constraints are of the inequality type.

The approach can additionally be characterised as population-based, incorporating 10 replicas in accordance with the *NUM_REPLICAS* hyperparameter. Within each replica, there are 25 chains of solutions determined by the *NUM_CHAINS* hyperparameter. Consequently, the total number of solutions being evaluated at any given time is 250, which is equivalent to the population size of the CGA.

4.1.1 Initialisation

The chains are initialised within the range of 0 to 1, as advised by the guidance provided in [11], specifically under the SA-derived solution generation framework presented in Section 4.1.4.

This range persists throughout the algorithm, until function evaluations are necessitated, at which point the *scale_up* lambda function is called to return the solution in the original state space.

Mirroring the CGA implementation, solutions were initialised without consideration for their feasibility. Whilst initialising within the feasible solution space was observed to enhance performance, this approach was deliberately omitted to ensure fair comparisons between the PT and CGA. However, the performance was notably compromised, given that solutions initialised deep within the infeasible zones of the search space faced challenges in escaping under the Metropolis-Hastings acceptance criterion.

4.1.2 Temperature Scheduling

A significant source of flexibility in PT lies in setting and adapting the temperatures of the replicas, as discussed in [2]. Drawing insights from the previous work of [1] on an unrelated, yet similar problem, this was taken advantage of by parameterising the temperature

schedule between 0 and 1, as follows:

$$T_i = \left(\frac{i}{N_{\text{replicas}}} \right)^p \quad \forall i \in \{1, \dots, N_{\text{replicas}}\} \quad (3)$$

Here, T_i denotes the temperature of the i th replica, and N_{replicas} is the total number of replicas. The exponent p is a hyperparameter referred to as *POWER_TERM* that can be tuned to optimise the performance of the algorithm.

Fig. 8 illustrates the influence of parameter p on the temperature schedule. It intuitively demonstrates how p affects the balance between exploration and exploitation. A higher value of p corresponds to a slower and more gradual increase in temperature, promoting a conducive environment for exploration. A smaller value of p results in a more rapid increase in temperature, encouraging exploitation of the local nature of the function.

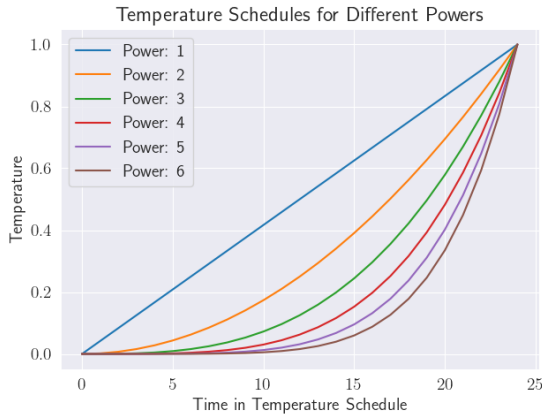


Figure 8: Temperature schedule (3) for the PT algorithm for different values of the *POWER_TERM* hyperparameter.

Tuning this balance is crucial for the algorithm’s performance against the KBF, and is conducted in Section 4.2. While alternative approaches, such as geometric progression [2] are recommended in most literature, the above formulation was made to enhance control over the replica temperatures.

4.1.3 Replica Exchange

The most popular approach to replica exchange, as outlined in [2], involves periodic execution. The interchange of solutions between replicas at adjacent temperature levels occurs at regular intervals. In this implementation, periodic exchange is guided by the hyperparameter *EXCHANGE_PARAM*. This hyperparameter assumes values in the range of 0 to 1, representing

the proportion of total iterations that must pass before another swap takes place. A value of 0.05 indicates that a swap occurs every 5 iterations when the maximum number of iterations is set to 100.

However, a second approach has also been implemented, which will be referred to as stochastic exchange. In this configuration, the possibility of replica exchange exists in every iteration, and the hyperparameter *EXCHANGE_PARAM* now functions as the probability governing the occurrence of this swap.

Both have been introduced to enhance the flexibility of the algorithm, allowing for either a more deterministic or stochastic approach to replica exchange. The optimal choice is determined in Section 4.2.

Upon meeting the conditions for a swap, the replicas are sequentially traversed in ascending order of temperature. Each replica undergoes an attempt to exchange all of its solutions with the subsequent replica in the sequence. The likelihood of accepting such a swap is dictated by the adapted Metropolis-Hastings acceptance criterion, as detailed in Section 4.1.5.

4.1.4 Metropolis Update

The inspiration for proposing a new solution stems from [11], which introduces a routine that leverages accumulated experience from prior iterations to generate SA update steps. Given the similarity between SA and PT, this approach was deemed suitable for the PT implementation.

In particular, a new solution is generated by:

$$\mathbf{x}_{\text{new}} = \mathbf{x}_{\text{old}} + \mathbf{D}\mathbf{u} \quad (4)$$

In this context, \mathbf{u} represents a vector of uniformly distributed random numbers within the range of $[-1, 1]$, while \mathbf{D} is a diagonal matrix specifying the maximum permissible step size in each dimension of the state space. In this PT implementation, \mathbf{D} is uniquely defined for each solution across all replicas. Following the acceptance of a new solution, the matrix \mathbf{D} is updated using the formula:

$$\mathbf{D}_{\text{new}} = (1 - \alpha)\mathbf{D}_{\text{old}} + \alpha\omega\mathbf{R}$$

Here, the constants α and ω are set at 0.1 and 2.1 respectively, in accordance with the guidelines presented in [11]. The matrix \mathbf{R} is a diagonal matrix whose elements represent the magnitudes of the successful changes made to each dimension within a specific solution.

4.1.5 Metropolis-Hastings Acceptance Criterion

The probability of accepting a proposed solution is determined by the Metropolis-Hastings acceptance criterion. This has been specifically formulated to work alongside the "adapting" update step, (4), with additional insights drawn from the works of [11] and [8]. The probability of accepting a new solution generated by Eq. (4) is given by:

$$P_{\text{accept}} = \min \left[1, \exp \left(\frac{f(\mathbf{x}_{\text{new}}) - f(\mathbf{x}_{\text{old}})}{k \cdot T_i \cdot \tilde{d}} \right) \right]$$

In the above formulation, the negation of the objective function: $-f(\mathbf{x})$, represents the fitness of a solution \mathbf{x} , k is the Boltzmann constant, T_i is the temperature of the i th replica, and \tilde{d} is euclidean normed distance between the old and new solutions.

However, when judging the swap of two solutions through a replica exchange, the acceptance probability is instead adapted to:

$$P_{\text{accept}} = \min \left[1, \exp \left(\frac{f(\mathbf{x}_{\text{new}}) - f(\mathbf{x}_{\text{old}})}{k \tilde{d}} \left(\frac{1}{T_{\text{old}}} - \frac{1}{T_{\text{new}}} \right) \right) \right]$$

in accordance with [8]. This imposes a harsher standard for the acceptance of a swap between replicas operating at vastly different temperatures, which is a desirable feature, especially when the power term, p , is set to a high value.

4.1.6 Termination

Again, the algorithm terminates after a maximum number of iterations, (defined as a hyperparameter), in this section of the study. A more stringent convergence criterion is adopted for the comparison between the two algorithms in Section 5. The solution with the lowest fitness value across all replicas is then returned as the optimal solution to the optimisation problem.

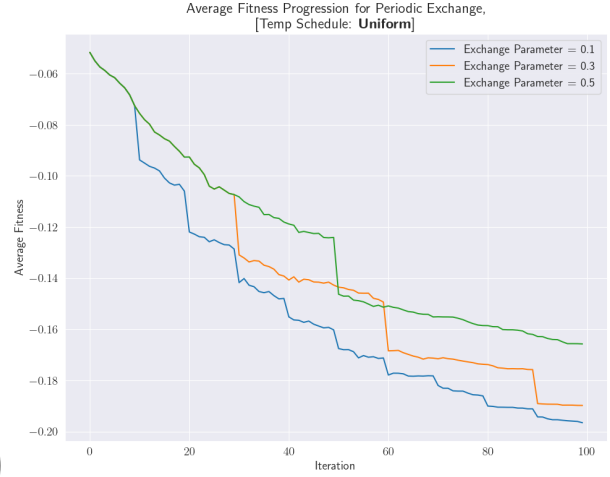
The total number of iterations is fixed at 100, which generally proved insufficient for the algorithm to achieve complete convergence. Rather than emphasising complete convergence, the assessment centers around tracking the evolution of average fitness values. This approach provides insights into the algorithm's effectiveness and speed, gauging its progress within the limited span of 100 iterations.

4.2 Tuning the PT

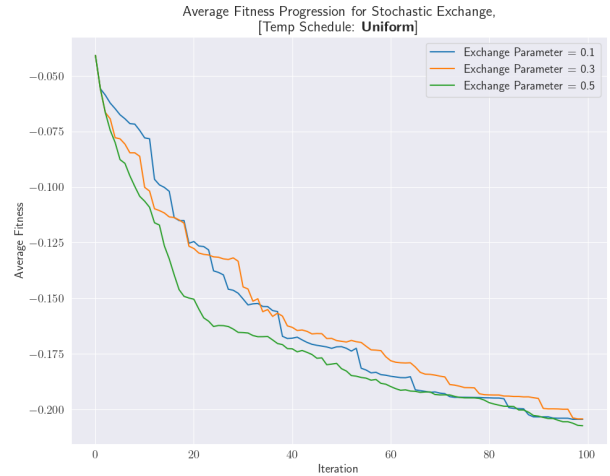
As with Section 3.2, the flexibility offered by PT was first explored using the parallelised experiments presented in code 11.

Unlike the CGA table (Table 3), the preliminary exploratory findings presented for PT (Table 4) imply a

subtle correlation between the power term of the temperature schedule and the algorithm's performance. Final fitness values exhibit relative constancy, hinting at no potential connection. However, the variability in algorithm performance appears more pronounced when considering the approach to replica exchange.



(a) Periodic Exchange



(b) Stochastic Exchange

Figure 9: Evolution of average fitness values of the PT solutions over 100 iterations for the two approaches to replica exchange. Temperature scheduling was kept uniform.

Fig. 9 instead illustrates the evolution of the average fitness values across all 100 iterations for both exchange procedures. The result present a clear distinction between the two approaches that remains consistent with repetition.

The periodic exchange approach seems to exhibit a higher dependency on the value assigned to the ex-

change parameter. In contrast, the stochastic exchange approach appears to follow a consistent descent, irrespective of the exchange parameter value. This observation is unexpected, considering that the exchange parameter is anticipated to exert a similarly significant influence on both approaches. In both cases, the exchange parameter affects the frequency of swaps, which is expected to impact the algorithm’s ability to escape local optima.

Additionally, the descent depicted in Fig. 9a seems steeper when the exchange parameter assumes smaller values. This aligns with expectations, as a smaller exchange parameter corresponds to a higher frequency of swaps in the periodic case. Here, the exchange parameter is interpreted as the proportion of total iterations that transpire before another swap takes place. When applied to the KBF, PT demonstrates greater efficacy when the algorithm is permitted to exchange solutions more frequently. This implies that increased exploration of the search space is well-suited to optimising the KBF.

Figures 11 and 12 present a final analysis of these hyperparameters. However, the stochastic heatmap, 12b, did not exhibit any consistency or superiority over the periodic heatmap, 11b, leading to no further consideration of stochastic exchange, given its suboptimal and unpredictable behavior.

In contrast, a consistent pattern unveiled itself in Fig. 11b, which persisted across repetitions. The PT algorithm, when applied to the KBF, consistently demonstrated a inclination towards a lower exchange parameter paired with a low power term. This may be rationalised by the fact that a lower exchange parameter corresponds to a heightened frequency of swaps, facilitating dialogue between the replicas and encouraging more exploration. Conversely, a low power term fosters a delicate equilibrium between exploration and exploitation, as evident in Fig. 8. The preference for uniform temperature scheduling likely stems from its enhanced capability to focus on local changes, after arriving at a particular optimum. Any exploratory capabilities stem from replica exchange, after which the algorithm can then focus on exploiting the local nature of the function.

Overall, these results imply that optimal settings for the PT algorithm on the KBF involve a power term of 1 and an exchange parameter of 0.01. These values are associated with a uniform temperature schedule and replica exchange taking place at every iteration. They position the PT algorithm favorably within the lighter regions of Fig. 11b.

In summary and to reiterate, the PT algorithm is now constructed as follows:

- **Number of Replicas:** 10
- **Number of Chains:** 25
- **Power Term:** 1
- **Exchange Procedure:** Always
- **Exchange Parameter:** N/A

Fig. 10 depicts the evolution of the fitness values over 100 iterations using the optimal hyperparameters delineated above. The fitness of the best solution drops noticeably once better solutions are identified and arrived at through the MCMC updating process, however the progress is much slower than that of the CGA.

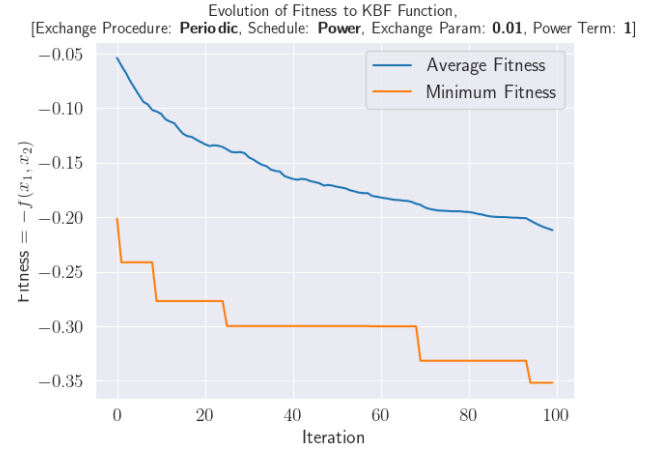


Figure 10: Evolution of the fitness values of the optimally-tuned PT solutions over 100 iterations.

The top row in Fig. 13 depicts the iterative evolution of the solutions. Notably, all solutions consistently converge towards the prominent maxima, and the global maximum is successfully identified in every repetition. The PT algorithm has demonstrated superior capabilities in exploration compared to the CGA.

The optimal solution is denoted by a green circle. For a more insightful comprehension of the Markov Chain Monte Carlo (MCMC) updating process, a particular solution within the final replica is emphasised in yellow. This solution is monitored throughout all iterations and it seems to become ensnared in a local optimum during the earlier iterations. However, it then successfully breaks free through the exchange mechanism and begins to converge towards a larger maximum in subsequent iterations.

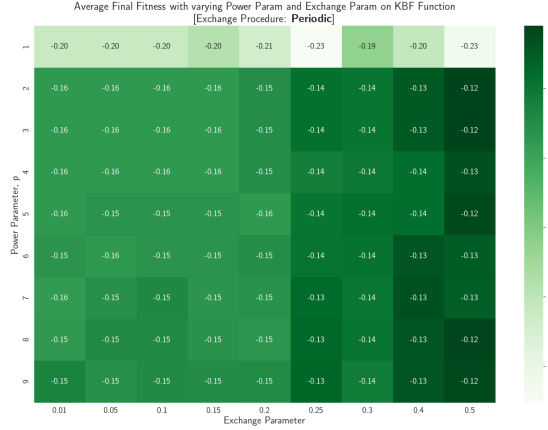
The bottom row in Fig. 13 showcases the final solutions of each replica, organised by temperature and su-

perimposed over the temperature-smoothed KBF contours. The arrangement follows an ascending temperature order, with the replica at the lowest temperature positioned on the left. These results shed light on the fundamental dynamics of the algorithm.

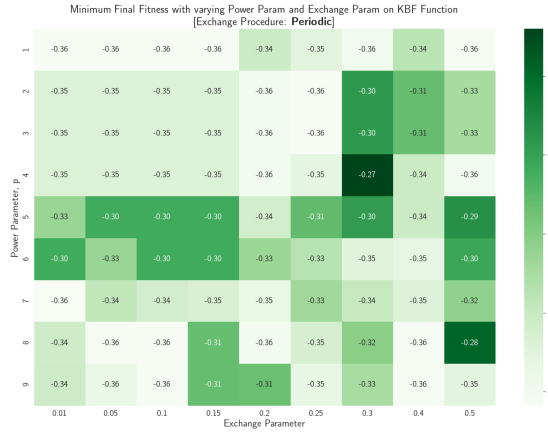
At lower temperatures, all solutions exhibit a clear inclination toward the higher, global maxima of the KBF. This preference arises from ample exploration of the search space. In contrast, solutions at higher tempera-

tures display a more dispersed distribution, converging around local optima.

Additionally, although these higher temperatures encourage localised exploration, the replica exchange mechanism facilitates the movement of solutions within the final temperature towards the larger maximas. Subsequent iterations are anticipated to drive convergence of solutions at higher temperatures towards the global maxima as well.

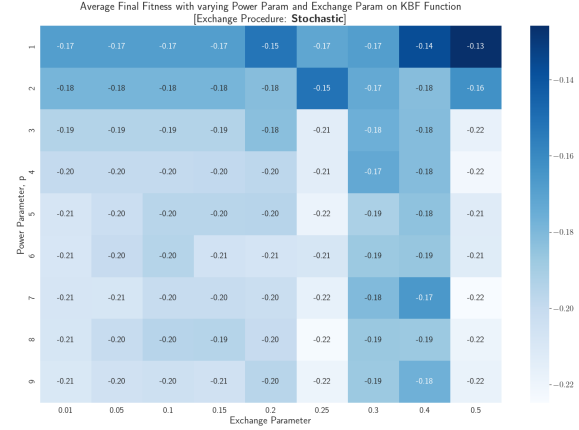


(a) Average Fitness Values

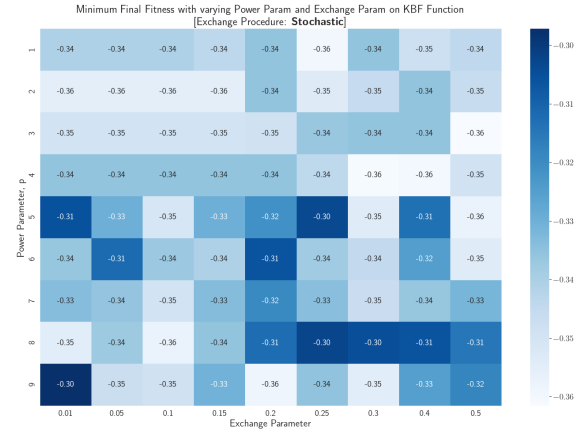


(b) Minimum Fitness Values

Figure 11: Heatmaps of the final fitness values after 100 iterations for the PT algorithm using periodic exchange. Here, the power term and exchange parameter were varied to assess their impact on performance. Despite the somewhat varied outcomes between repetition, a consistent trend emerges - the PT algorithm demonstrates a preference for a lower exchange parameter coupled with a moderate power term.



(a) Average Fitness Values



(b) Minimum Fitness Values

Figure 12: Heatmaps of the final fitness values after 100 iterations for the PT algorithm using stochastic exchange. Here, the power term and exchange parameter were varied to assess their impact on performance. The values for the best (minimum) final fitness varied greatly upon repetition. Generally, performance was outmatched by periodic exchange, which was also more consistent and controllable.

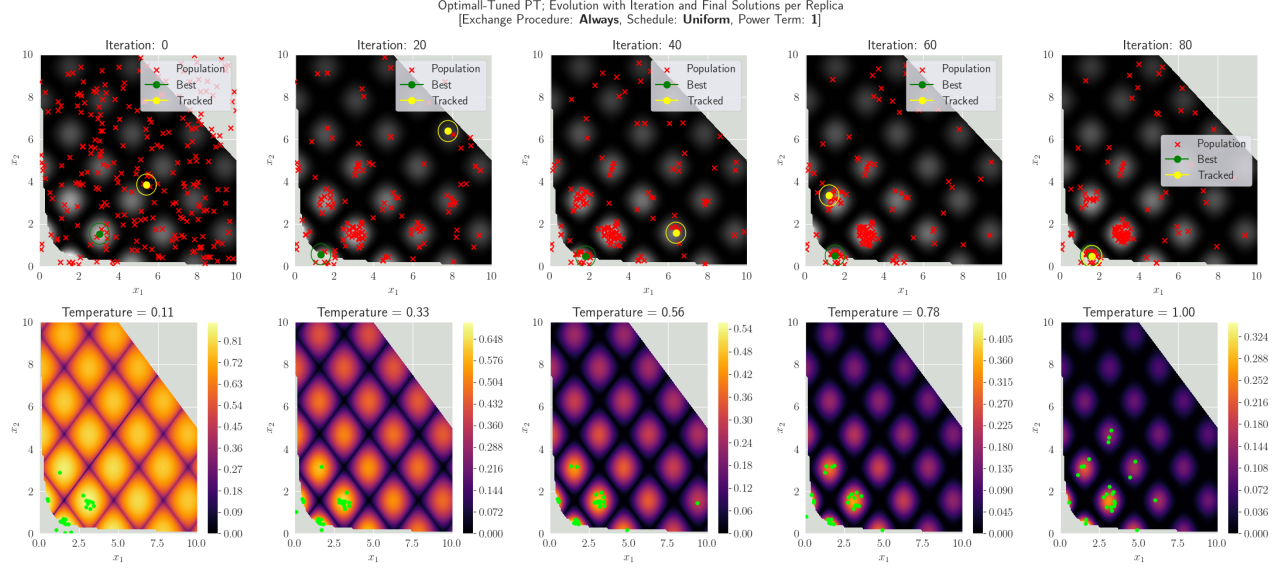


Figure 13: Visualisation of the optimally-tuned PT algorithm. The first row illustrates its evolution with each iteration. The optimal solution is marked with a green circle. To enhance the understanding of the MCMC updating process, a specific solution within the final replica is highlighted in yellow. The second row displays the final solutions arranged for each replica, overlaid across the smoothened KBF, $f(\mathbf{x})^T$, to help visualise the theory of PT. Lower temperatures facilitate exploration, while higher temperatures encourage exploitation. However, exchange ensures that solutions at higher temperatures are also driven towards the global maxima.

5 Comparison

After fine-tuning both algorithms for the 2-dimensional KBF, a comparative analysis was conducted to assess their performances in optimising the 8-dimensional KBF using the code presented in 12. 50 optimisation loops were conducted for each algorithm, each with a different initialisation. Collected data included expected values and standard deviations of fitness values, as well as the mean CPU time taken to complete all 10,000 iterations and the average iteration count at which convergence occurred.

To uphold fairness and reproducibility, these initialisations were shared between both algorithms. They were generated using random number seeds within the *generate_initial* function, which is part of the *helper_functions.py* file 7. The implementation of this function was crucial for ensuring reproducible outcomes and guaranteeing that solutions were initialised away from the global maxima. This was achieved by imposing a constraint that the maximum function value of these initial collections never exceeded 0.3.

Each run was limited to a maximum of 10,000 function evaluations, and convergence was identified as the point where the l_2 -normed difference between the minimum fitness values of consecutive iterations stayed within a tolerance of 0.00025 for 1300 consecutive iterations. This was deemed appropriate after the

trial run outlined in Section 5.1.

5.1 Trial Run

Prior to the experiment, a trial run of 15,000 function evaluations was conducted using one initialisation to assess the suitability of the convergence criterion, and to set expectations.

The outcomes are detailed in Table 1 and illustrated in Fig. 14. These findings underscore a particularly noteworthy result.

The two algorithms yielded fairly divergent solutions, as described in the last row of Table 1. The CGA underwent significant change within the initial 1% of iterations, after which it stabilised near its ultimate solution. In contrast, the PT algorithm consistently evolved over the entire span of 15,000 iterations. Consequently, the CGA demonstrated superior performance in the early iterations, but the PT algorithm ultimately surpassed it by converging to a lower minimum fitness value. This observation is a promising indication of the PT's capacity to escape local optima and explore the search space more efficiently.

However, this advantage comes with a trade-off in terms of increased computational demands. Specifically, the PT algorithm required approximately double the time to finish the 15,000 iterations compared to the

CGA. Additionally, the PT algorithm required an extra 2551 iterations to converge to its optimal solution, resulting in a greater time investment of approximately 335.9 seconds, with respect to the CGA.

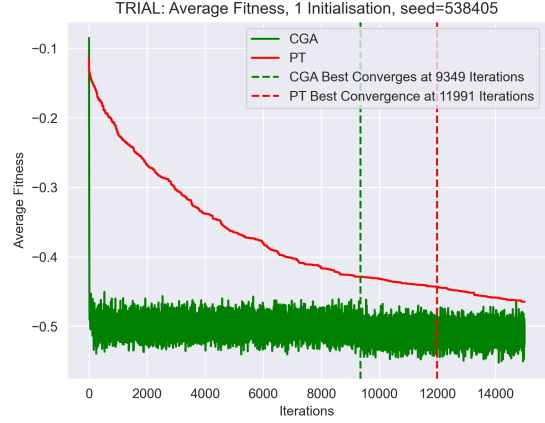
While the CGA required less implementation effort, it consistently exhibited swift convergence, quickly reaching solutions in close proximity to its optimal outcome. In contrast, the PT algorithm displayed a gradual descent, highlighting its greater exploratory nature.

Upon repeated trials, the PT algorithm was unable to consistently overtake the CGA within 10,000 iterations. As a result, it is unlikely that this dynamic will be evident in the final comparison. Nevertheless, the trial run yields valuable insights into the relative performance of the two algorithms and sheds light on potential trade-offs. Additionally, it reinforces the appropriateness of the convergence criterion, which was subsequently employed in the final comparative analysis.

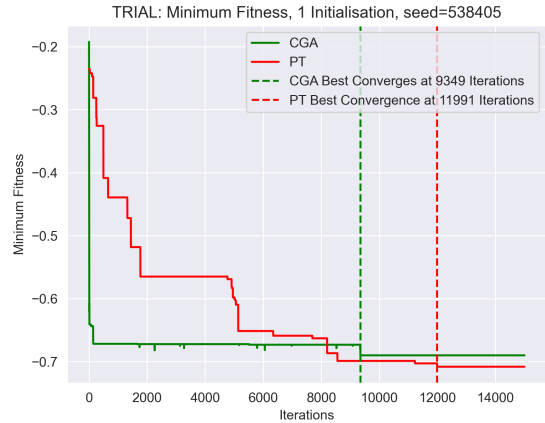
Although the PT algorithm's final solutions were only marginally superior to those of the CGA, it demanded significantly more CPU time to attain this outcome. Depending on the application and the consistency of this result, the CGA might be a more preferable algorithm. This observation is a valuable insight from the trial run, and emphasises the importance of considering expectations and identifying standard deviations in the final comparison.

Metric	CGA	PT
Final Avg. Fitness	-0.4971	-0.4646
Final Min. Fitness	-0.6901	-0.7081
Total Time Taken	299.2	592.7
Iters to Convergence	9349	11990
Optimal Solutions	3.104	3.111
	3.027	2.948
	1.624	3.071
	0.5872	1.643
	0.5363	0.3356
	0.6130	0.4636
	0.5294	0.4284
	0.4808	0.2457

Table 1: Trial comparison between CGA and PT, after 15,000 iterations with one shared initialisation, (random seed = 538405).



(a) Average fitnesses.



(b) Minimum fitnesses.

Figure 14: Evolution of fitness values over 15,000 iterations for the CGA and PT algorithms, using a single initialisation as a trial run, (random seed = 538405).

5.2 Final Comparison

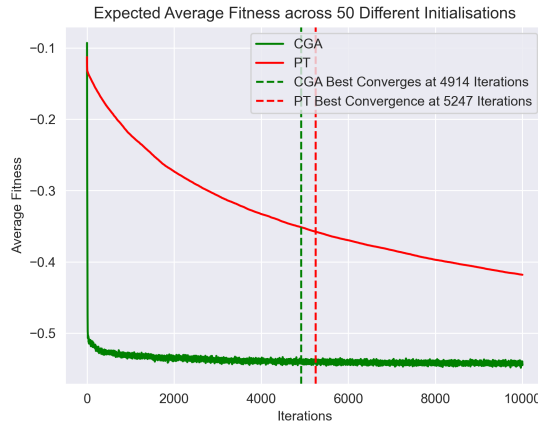
Having validated the experiment with a trial run, the final comparison was conducted using 50 different initialisations, generated with the following random seeds:

[588541, 776379, 146310, 178897, 630385, 455226, 75798, 763473, 295412, 733068, 521014, 926074, 667371, 58738, 543141, 263789, 572073, 46141, 360713, 247094, 379228, 395478, 102912, 110855, 602020, 673151, 903361, 138526, 750056, 969814, 998683, 433667, 885222, 414036, 401547, 862285, 671914, 26963, 764090, 99348, 794953, 642883, 292349, 168953, 736085, 528540, 558369, 41243, 168530, 285025]

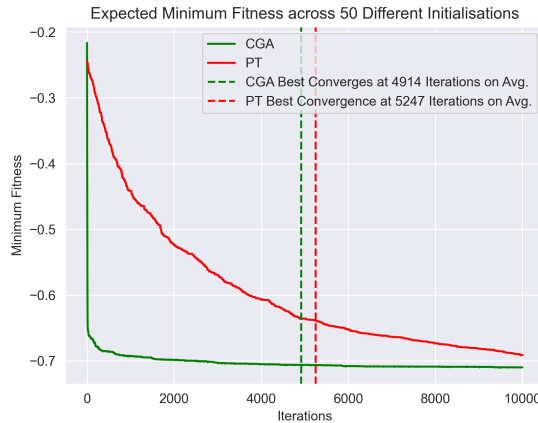
The results are presented in Table 2 and Fig. 15.

Metric	CGA	PT
Expected Final Avg. Fitness	-0.5431	-0.4181
Std. in Final Avg. Fitness	0.01658	0.01536
Expected Final Min. Fitness	-0.7099	-0.6913
Std. in Final Min. Fitness	0.02083	0.03044
Expected Total Time Taken	90.48	170.2
Std. in Time Taken	1.412	4.544
Mean Iters to Convergence	4915	5247

Table 2: Final comparison between CGA and PT, after 10,000 iterations with 50 shared initialisations.



(a) Average fitnesses.



(b) Minimum fitnesses.

Figure 15: Evolution of fitness values over 10,000 iterations for the CGA and PT algorithms, using 50 shared initialisations.

The contrast in the mean final fitness values of the optimal solutions amounted to a mere 0.0186, corresponding to a marginal 2.65% disparity. This slight difference implies that the CGA demonstrated only a modest advantage over the PT algorithm in terms of

solution quality after 10,000 iterations. This outcome suggests that the 10,000 iterations was not sufficient for the PT algorithm to thoroughly explore the search space and converge to the global maxima. Although it is challenging to definitively assert that more iterations would have led to global convergence, the observed trend in Fig. 15a strongly suggests that such convergence was likely, and that the PT algorithm would have eventually overtaken the CGA.

However, the contrast in the standard deviations of their final minimum fitness values paints a different picture, showcasing a discrepancy of 0.00961, equivalent to 37.5%. The PT algorithm demonstrated considerably greater stability in reaching its final optimal solution. Although its average fitness across the entire set of solutions displayed more variability, than the CGA this characteristic regarding its optimal solution could confer an advantage to the PT algorithm in specific applications where a slightly inferior objective function evaluation is acceptable in exchange for consistently reliable performance.

Furthermore, an additional noteworthy observation is that the PT algorithm demanded an additional 79.72 seconds to execute the entire 10,000 iterations, representing a substantial 61.2% increase above the CGA. On average, it also necessitated 332 more iterations for convergence, translating to roughly 44.83 seconds more than the CGA, marking a noteworthy 67.0% difference. This notable disparity can be ascribed to the inherently more exploratory approach of the PT algorithm, underscoring that the CGA is better suited for applications prioritising computational efficiency.

Another salient aspect to consider is that the CGA was comparatively simpler to implement than the PT algorithm. However, it is important to acknowledge the potential influence of personal bias, as Genetic Algorithms were covered in the lecture materials, while the PT algorithm was not. Similarly, accessing literature for inspiration and guidance was markedly easier for the CGA implementation and its adaptations than it was for the PT algorithm. This discrepancy in accessibility presents an additional practical advantage in favor of the CGA.

Overall, the PT algorithm holds promise in consistently attaining the global optimum with extended CPU time. Conversely, the CGA boasts efficiency, quicker convergence, and a simpler implementation process. The selection between these two algorithms hinges on the specific application and the relative significance assigned to these factors.

5.3 Limitations of this Study

While the preceding study was purposefully comprehensive and rigorous, there exist significant opportunities for enhancing the reliability and granularity of the results. The following points outline some notable limitations:

- **Limited Exploration of Hyperparameter Space:** The hyperparameter space was explored to a limited extent. The selection of specific parameters for fine-tuning was partly driven by educational objectives and the desire to solidify understanding of the algorithms. The approach, while educational, was not exhaustive, leaving room for further refinement of key parameters like population size, replica count, and chain count. There is potential to achieve improved and expedited results through more comprehensive hyperparameter tuning.
- **Dimensional Discrepancy:** Hyperparameter tuning was specifically carried out for the two-dimensional KBF. It is crucial to note that the optimal hyperparameters identified for the 2-dimensional KBF may not necessarily be optimal for its 8-dimensional counterpart. This discrepancy represents a limitation in the study, and there is a possibility that results could differ if hyperparameter tuning were conducted for the 8-dimensional KBF. However, the choice to focus on the 2-dimensional KBF for hyperparameter tuning was driven by considerations of computational efficiency and the facilitation of visualisation.
- **Parallelisation:** Despite temptation, parallelisation of the algorithms was deliberately avoided to maintain the reproducibility and comparability of the results. While parallelising the algorithms could have enhanced efficiency, particularly for the PT algorithm, known for its suitability for straightforward parallelisation, its implementation might introduce an unfair advantage and complicate the overall comparison.

References

- [1] Ben Calderhead and Mark Girolami. Estimating bayes factors via thermodynamic integration and population mcmc. *Computational Statistics & Data Analysis*, 53(12):4028–4045, 2009.
- [2] David J. Earl and Michael W. Deem. Parallel tempering: Theory, applications, and new perspectives. *Physical Chemistry Chemical Physics*, 7(23):3910, 2005.
- [3] M.A. El-Beltagy and A.J. Keane. A comparison of various optimization algorithms on a multilevel problem. *Engineering Applications of Artificial Intelligence*, 12(5):639–654, 1999.
- [4] Randy L. Haupt, S. E. Haupt, and Randy L. Haupt. *Practical Genetic Algorithms*, chapter 3: The Continuous Genetic Algorithm, pages 51–66. John Wiley & Sons, Ltd, 2003.
- [5] Frank Hoffmeister and Thomas Bäck. Genetic algorithms and evolution strategies: Similarities and differences. In Hans-Paul Schwefel and Reinhard Männer, editors, *Parallel Problem Solving from Nature*, pages 455–469, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.
- [6] Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs*. Springer, 2011.
- [7] Brad L. Miller and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Syst.*, 9, 1995.
- [8] Radford M. Neal. Sampling from multimodal distributions using tempered transitions. *Statistics and Computing*, 6(4):353–366, 1996.
- [9] Geoff Parks. Genetic algorithms: Lecture notes. Cambridge University, 4M17 Practical Optimisation Module, 2023. Unpublished.
- [10] Geoff Parks. Simulated annealing: Lecture notes. Cambridge University, 4M17 Practical Optimisation Module, 2023. Unpublished.
- [11] Geoffrey Thomas Parks. An intelligent stochastic optimization routine for nuclear fuel cycle design. *Nuclear Technology*, 89(2):233–246, 1990.
- [12] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017.

6 Appendix

6.1 Supplementary Results regarding CGA Tuning

Selection Method	Mating Procedure	Iterations	Final Avg. Fitness	Final Min. Fitness
Proportional	Crossover	10	-0.19890798	-0.247328018
		100	-0.212384457	-0.24867806
	Heuristic Crossover	10	-0.199296791	-0.258198948
		100	-0.22656183	-0.250977443
Tournament	Crossover	10	-0.309572335	-0.331996331
		100	-0.309235401	-0.342745604
	Heuristic Crossover	10	-0.241261775	-0.262876797
		100	-0.247574635	-0.262876811
SRS	Crossover	10	-0.189986008	-0.208434151
		100	-0.288766718	-0.319667279
	Heuristic Crossover	10	-0.177779432	-0.207148488
		100	-0.182101833	-0.338823558

Table 3: Raw results from an initial exploration of the selection method and mutation procedure hyperparameters within the CGA. Presented as the final fitness values of the CGA population after 10 and 100 iterations. Here, minimum fitness refers to the fitness of the best (feasible) individual within the population. Additionally, the mutation rate was set to 0.05, and the crossover probability was set to 0.7.

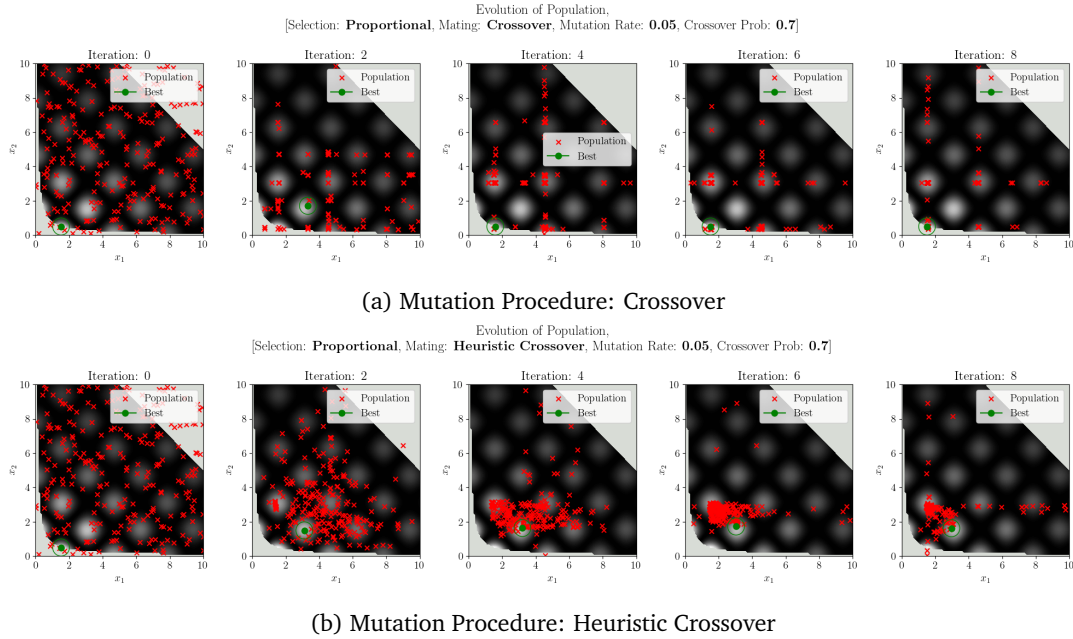
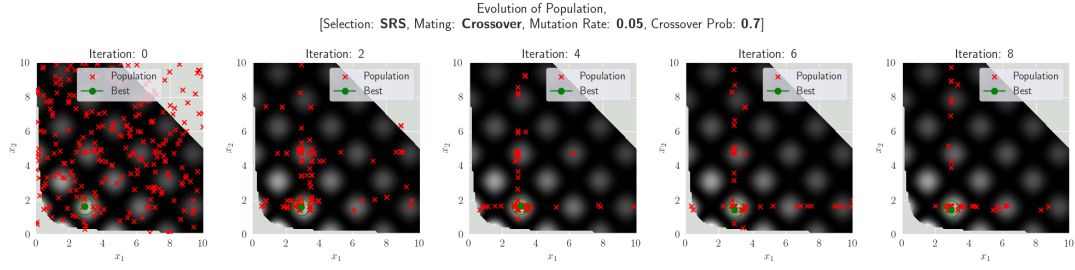
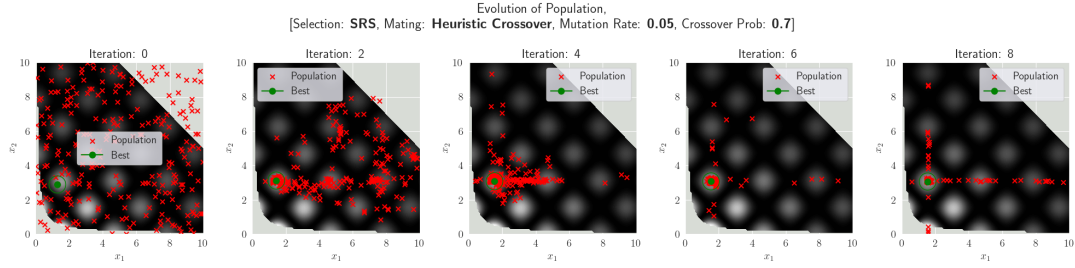


Figure 16: Evolution of the CGA population over 8 iterations using proportional selection. Proportional selection proved to be a somewhat effective selection method. However, it was not chosen over tournament selection as the primary selection method for the reasons outlined in Section 3.2.



(a) Mutation Procedure: Crossover



(b) Mutation Procedure: Heuristic Crossover

Figure 17: Evolution of the CGA population over 8 iterations using stochastic remainder selection without replacement (SRS). SRS proved to be the second most effective selection method, when compared to Proportional Selection and Tournament Selection, as discussed in 3.2.

6.2 Supplementary Results regarding PT Tuning

Exchange Procedure	Exchange Parameter	Power Term	Final Avg. Fitness	Final Min. Fitness
Periodic	0.1	1	-0.186415811	-0.34081297
		3	-0.186415811	-0.34081297
		5	-0.192142049	-0.34081297
	0.3	1	-0.183591253	-0.335207939
		3	-0.183591253	-0.335207939
		5	-0.178574074	-0.310814711
	0.5	1	-0.166370319	-0.355617525
		3	-0.166370319	-0.355617525
		5	-0.162769374	-0.31189813
Stochastic	0.1	1	-0.182496471	-0.327434397
		3	-0.182496471	-0.327434397
		5	-0.197887075	-0.328011329
	0.3	1	-0.191162383	-0.310872726
		3	-0.191162383	-0.310872726
		5	-0.196042151	-0.314329893
	0.5	1	-0.206621007	-0.347483314
		3	-0.206621007	-0.347483314
		5	-0.211224821	-0.349273739

Table 4: Raw results from an initial exploration of the exchange procedure, exchange parameter, and power term hyperparameters within PT. Presented as the final fitness values of the PT solutions after 100 iterations. Here, minimum fitness refers to the fitness of the best (feasible) solution across all chains.

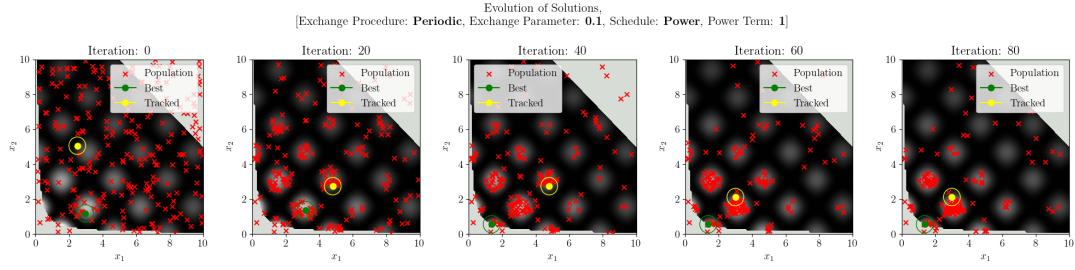


Figure 18: Evolution of the chain across all replicas over 100 iterations using periodic replica exchange and a uniform temperature scheduling, (power term of 1). Here, replica exchange occurs every 10% of the time, (exchange parameter set to 0.1). The 'tracked' solution is distinct from the 'best' solution. It is a random solution within the final replica that is observed to verify the MCMC evolution.

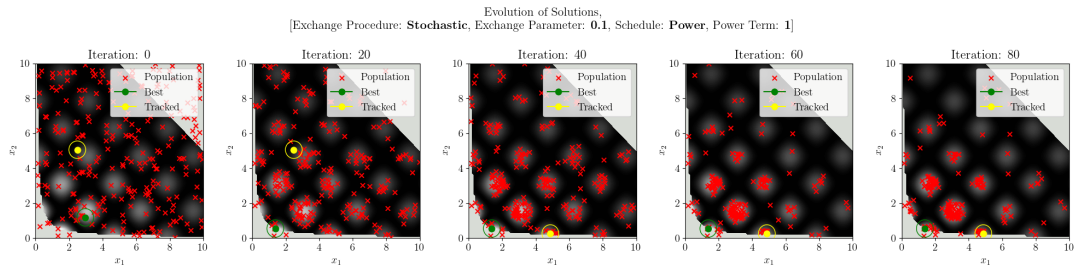


Figure 19: Evolution of the chain across all replicas over 100 iterations using stochastic replica exchange and a uniform temperature scheduling, (power term of 1). Here, replica exchange occurs with a probability of 10%. The 'tracked' solution is distinct from the 'best' solution. It is a random solution within the final replica that is observed to verify the MCMC evolution.

7 Code

To ease development, the codebase has been modularised in the manner presented below:

```
src/
  algorithms/
    CGA/
      CGA.py
      mating_functions.py
      selection_functions.py
    PT/
      PT.py
      replica_exchange_functions.py
      temp_prog_functions.py
  utils/
    helper_functions.py
    plotting_functions.py
  functions.py
FinalComparison.py
CGA_TuningExperiments.py
PT_TuningExperiments.py
```


7.1 algorithms

7.1.1 CGA

```
1 """
2 Candidate No : 5730E, Module: 4M17
3
4 Description :
5     This file contains the class for the continuous genetic algorithm.
6 """
7
8 import numpy as np
9 import sys; sys.path.append('.')
10
11 from src.algorithms.CGA.selection_functions import proportional_selection,
12     tournament_selection, SRS_selection
13 from src.algorithms.CGA.mating_functions import crossover, heuristic_crossover
14 from src.utils.helper_functions import satisfy_constraints
15
16 class ContinuousGeneticAlgorithm():
17     """
18     Class for continuous genetic algorithm.
19     """
20     def __init__(self, population_size, chromosome_length, num_parents, objective_function,
21         tournament_size, range=(0,10), mutation_rate=0.1, crossover_prob=0.8, selection_method='
22         Tournament', mating_procedure='Heuristic Crossover', constraints=True):
23         """
24         Constructor for continuous genetic algorithm.
25
26         Parameters:
27         - population_size (int): Number of individuals in population
28         - chromosome_length (int): Size of vector individual, (number of genes), i.e.
29         dimension of solution space
30         - num_parents (int): Number of parents to select for mating
31         - objective_function (function): Objective function to optimise
32         - tournament_size (int): Size of subset of population for tournament selection
33         - range (tuple): Range of values for genes, determined by constraints of problem
34         - mutation_rate (float): Mutation rate
35         - crossover_prob (float): Crossover rate
36         - selection_method (str): Selection method used for parent selection
37         - mating_procedure (str): Mating procedure used for reproduction
38         - constraints (bool): Whether to satisfy constraints with parent selection or not
39         """
40         self.population_size = population_size
41         self.chromosome_length = chromosome_length # n in R^n, dimension of the search space
42         self.num_parents = num_parents
43         self.func = objective_function
44         self.tournament_size = tournament_size
45         self.lb = range[0]
46         self.ub = range[1]
47         self.mutation_rate = mutation_rate
48         self.crossover_prob = crossover_prob
49         self.constraints = constraints
50
51         # Dictionaries to map string to function call. Function imported from directory files
52         selection_mapping = {'Proportional': proportional_selection,
53             'Tournament': tournament_selection,
54             'SRS': SRS_selection # SRS = Stochastic Remainder Selection
55             without Replacement
56         }
57
58         mating_mapping = {'Crossover': crossover,
59             'Heuristic Crossover': heuristic_crossover
60         }
61
62         # Check if selection method and mating procedure are valid
63         if selection_method not in ['Proportional', 'Tournament', 'SRS']:
64             raise ValueError(f"Invalid selection method: {selection_method}")
65         else:
66             self.selection_process = selection_mapping[selection_method]
```

```

62
63     if mating_procedure not in ['Crossover', 'Heuristic Crossover']:
64         raise ValueError(f"Invalid mating procedure: {mating_procedure}")
65     else:
66         self.mating_process = mating_mapping[mating_procedure]
67
68     # Initialise population
69     self.initialise_population()
70
71 def initialise_population(self):
72     """
73     Initialise population with random values between lb and ub.
74     """
75     self.population = np.random.uniform(low=self.lb,
76                                         high=self.ub,
77                                         size=(self.population_size, self.chromosome_length
78 )
79
80     self.fitness = np.zeros(self.population_size) # Initialise fitness of population
81     self.evaluate_fitness() # Update fitness of population with starting individual
82
83 def evaluate_fitness(self):
84     """
85     Evaluate fitness of population.
86
87     Parameters:
88     - fitness_function (function): Fitness function to evaluate fitness of population
89     """
90     for i in range(self.population_size):
91         self.fitness[i] = - self.func(self.population[i])
92
93     # Evaluate rankings of individuals in population by fitness
94     self.parent_rankings = np.argsort(self.fitness) # Indices of individuals in order of
95     fitness
96
97     # Update best individual and best fitness
98     self.best_individual = self.population[self.parent_rankings[0]]
99     self.min_fitness = self.fitness[self.parent_rankings[0]]
100
101     # Best individual must satisfy constraints, if not, find next best individual that
102     does
103     i = 1
104     while not satisfy_constraints(self.best_individual):
105         self.best_individual = self.population[self.parent_rankings[i]]
106         self.min_fitness = self.fitness[self.parent_rankings[i]]
107         i += 1
108
109 def select_parents(self):
110     """
111     Select parents for mating.
112
113     Returns:
114     - parents (np.array): Indices of parents selected for mating
115     """
116     return self.selection_process(self)
117
118 def mate(self):
119     """
120     Mate parents to produce offspring.
121
122     Returns:
123     - offspring (np.array): Offspring of parents
124     """
125     return self.mating_process(self)
126
127 def mutate(self):
128     """
129     Mutate offspring. Every gene has a mutation rate chance of being mutated.

```

```

128     """
129     for i in range(self.population_size):
130         for j in range(self.chromosome_length):
131             if np.random.rand() < self.mutation_rate:
132                 self.population[i][j] = np.random.uniform(low=self.lb, high=self.ub)
133
134 def evolve(self):
135     """
136     Evolve population for one generation.
137     """
138     offspring = self.mate()
139     self.population = offspring
140     self.mutate()
141     self.evaluate_fitness()

```

Listing 1: CGA.py

```

1  """
2  Candidate No : 5730E, Module: 4M17
3
4  This file contains the mating functions for the CGA algorithm.
5  """
6
7  import numpy as np
8
9  def crossover(CGA):
10     """
11     Crossover mating procedure.
12     """
13     # Select parents
14     selected_parents = CGA.select_parents()
15
16     # Reshape them into a set of potential parent pairs
17     parent_pairs = np.array(selected_parents).reshape(-1, 2)
18
19     # Initialise new offspring to replace population
20     offspring = np.zeros((CGA.population_size, CGA.chromosome_length))
21
22     # Iterate through population
23     for i in range(CGA.population_size):
24
25         # Assign random pair of parents from parent_pairs
26         parent1, parent2 = parent_pairs[np.random.randint(len(parent_pairs))]
27
28         # Assign random crossover point
29         p = np.random.randint(1, CGA.chromosome_length)
30
31         # Swap genes from parents to create offspring
32         if np.random.rand() < CGA.crossover_prob:
33             offspring[i][:p] = CGA.population[parent1][:p]
34             offspring[i][p:] = CGA.population[parent2][p:]
35         else:
36             offspring[i][:p] = CGA.population[parent2][:p]
37             offspring[i][p:] = CGA.population[parent1][p:]
38
39     return offspring
40
41 def heuristic_crossover(CGA):
42     """
43     Heuristic crossover mating procedure. Inspired by the relevant section in https://doi.org/10.1002/0471671746.ch3
44     """
45
46     # Select parents
47     selected_parents = CGA.select_parents()
48
49     # Reshape them into a set of potential parent pairs
50     parent_pairs = np.array(selected_parents).reshape(-1, 2)
51
52     # Initialise new offspring to replace population

```

```

53     offspring = np.zeros((CGA.population_size, CGA.chromosome_length))
54
55     # Iterate through population
56     for i in range(CGA.population_size):
57
58         # Assign random pair of parents from parent_pairs
59         parent1, parent2 = parent_pairs[np.random.randint(len(parent_pairs))]
60
61         # Iterate through all the genes of an individual/chromosome
62         for j in range(CGA.chromosome_length):
63
64             # Heuristic crossover with probability, CGA.crossover_prob
65             b = np.random.rand() # b is a random number between 0 and 1
66
67             if np.random.rand() < CGA.crossover_prob:
68                 # p_new = b * (p1 - p2) + p2
69                 offspring[i][j] = b * (CGA.population[parent1][j] - CGA.population[parent2]
70 ] [j]) + CGA.population[parent2][j]
71             else:
72                 # p_new = b * (p2 - p1) + p1
73                 offspring[i][j] = b * (CGA.population[parent2][j] - CGA.population[parent1]
74 ] [j]) + CGA.population[parent1][j]
75
76     return offspring

```

Listing 2: mating_functions.py

```

1  """
2  Candidate No : 5730E, Module: 4M17
3
4  This file contains the selection functions for the CGA algorithm.
5  """
6
7  import numpy as np
8  import sys; sys.path.append('.')
9  from src.utils.helper_functions import satisfy_constraints
10
11  def proportional_selection(GCA):
12      """
13      Proportional selection of parents.
14
15      Args:
16      - GCA (CGA): Continuous Genetic Algorithm object passed into this function using self.
17      select_parents(self)
18
19      Returns:
20      - selected_individuals (list): List of indices of selected individuals for mating,
21      length = GCA.num_parents
22      """
23      # Calculate probabilities
24      probabilities = GCA.fitness / np.sum(GCA.fitness)
25
26      # Select individuals based on probabilities
27      selected_individuals = list(np.random.choice(GCA.population_size, size=GCA.num_parents
28 , p=probabilities))
29
30      # Retry - reject parents that do not satisfy constraints
31      if GCA.constraints == True:
32          for i in range(GCA.num_parents):
33              while not satisfy_constraints(GCA.population[selected_individuals[i]]):
34                  selected_individuals[i] = np.random.choice(GCA.population_size, p=
35 probabilities)
36
37      return selected_individuals
38
39  def tournament_selection(GCA):
40      """
41      Tournament selection of parents.
42
43      Args:

```

```

40     - GCA (CGA): Continuous Genetic Algorithm object passed into this function using self.
      select_parents(self)
41
42     Returns:
43     - selected_individuals (list): List of indices of selected individuals for mating,
      length = GCA.num_parents
44     """
45
46     # Initialise list of selected individuals
47     selected_individuals = []
48
49     # Select top two parents for each tournament, so need 'GCA.num_parents // 2'
      tournaments
50     for i in range(GCA.num_parents//2):
51
52         # Take subset of population
53         subset = np.random.choice(GCA.population_size, size=GCA.tournament_size, replace=
      False)
54
55         # Take top two parents
56         parent1 = subset[np.argmin(GCA.fitness[subset])]
57         subset = np.delete(subset, np.argmin(GCA.fitness[subset]))
58         parent2 = subset[np.argmin(GCA.fitness[subset])]
59
60         # Retry, reject parents that do not satisfy constraints
61         if GCA.constraints == True:
62             while not satisfy_constraints(GCA.population[parent1]):
63                 subset = np.delete(subset, np.argmin(GCA.fitness[subset]))
64                 parent1 = subset[np.argmin(GCA.fitness[subset])]
65             while not satisfy_constraints(GCA.population[parent2]):
66                 subset = np.delete(subset, np.argmin(GCA.fitness[subset]))
67                 parent2 = subset[np.argmin(GCA.fitness[subset])]
68
69         # Add parents to list of selected individuals
70         selected_individuals += [parent1, parent2]
71
72     return selected_individuals
73
74 def SRS_selection(GCA):
75     """
76     Stochastic Remainder Selection without Replacement (SRS) of parents.
77
78     Args:
79     - GCA (CGA): Continuous Genetic Algorithm object passed into this function using self.
      select_parents(self)
80
81     Returns:
82     - selected_individuals (list): List of indices of selected individuals for mating,
      length = GCA.num_parents
83     """
84
85     # Calculate probabilities
86     probabilities = GCA.fitness / np.sum(GCA.fitness)
87
88     # Calculate expected number of copies of each individual
89     expected_num_copies = probabilities * GCA.num_parents
90
91     # Calculate integer number of copies of each individual
92     num_copies = np.floor(expected_num_copies)
93
94     # Calculate remainder, which later serves as the probability of further selection
95     remainder = expected_num_copies - num_copies
96
97     # Initialise list of selected individuals
98     selected_individuals = []
99
100    # Duplicate individuals "num_copies" times
101    for i in range(GCA.population_size):
102

```

```

103     # Add only feasible individuals
104     if satisfy_constraints(GCA.population[i]):
105         selected_individuals += [i] * int(num_copies[i]) # Add i to list num_copies[i]
106         times
107     # Remainder must satisfy sum(remainder) = 1, since it serves as the probabilities for
108     further selection
109     remainder = remainder / np.sum(remainder)
110
111     # Calculate remaining number of individuals that need to be selected
112     remaining_number = GCA.num_parents - len(selected_individuals)
113
114     # Cannot be negative
115     remaining_number = remaining_number if remaining_number > 0 else 0
116
117     # Select individuals using remainder probabilities
118     selected_individuals += list(np.random.choice(GCA.population_size, size=
119     remaining_number, p=remainder))
120
121     # Reject parents that do not satisfy constraints
122     if GCA.constraints == True:
123         for i in range(len(selected_individuals)):
124             while not satisfy_constraints(GCA.population[selected_individuals[i]]):
125                 selected_individuals[i] = np.random.choice(GCA.population_size, p=
126                 remainder)
127
128     return selected_individuals

```

Listing 3: selection_functions.py

7.1.2 PT

```

1     """
2     Candidate No : 5730E, Module: 4M17
3
4     Description :
5         This file contains the class for the parallel tempering algorithm.
6     """
7     import numpy as np
8
9     import sys; sys.path.append('.')
10    from src.algorithms.PT.temp_prog_functions import power_progression, geometric_progression
11    from src.algorithms.PT.replica_exchange_functions import period_exchange,
12    stochastic_exchange, always_exchange
13    from src.utils.helper_functions import satisfy_constraints
14
15    class ParallelTempering():
16        """
17        Class for parallel tempering algorithm.
18        """
19        def __init__(self, objective_function, x_dim, range=(0,10), alpha=0.1, omega=2.1,
20        num_replicas=10, num_chains=25, exchange_procedure='Periodic', exchange_param=0.2,
21        schedule_type='Power', power_term=1, total_iterations=100, constraints=True):
22            """
23            Constructor for parallel tempering algorithm.
24
25            Parameters:
26            - objective_function (function): Objective function to optimise
27            - x_dim (int): Dimension of solution space
28            - range (tuple): Range of values for x, determined by constraints of problem
29            - alpha (float): Dampening constant for max. allowable step size update
30            - omega (float): Weighting for max. allowable step size update
31            - num_replicas (int): Number of replicas
32            - num_chains (int): Number of solutions per replica
33            - exchange_procedure (str): Procedure for exchanging solutions between replicas
34            - exchange_param (float between 0 and 1):
35                if exchange_procedure = 'Periodic', this is the percentage of iterations after
36                which to exchange solutions
37                if exchange_procedure = 'Stochastic', this is the probability of exchanging
38                solutions between replicas during each iteration

```



```

34     - progression_type (str): Type of progression for temperature scheduling
35     - power_term (float > 1): Temperature progression power term, if using power
progression. Schedule is  $(i/N)^{\text{power\_term}}$ 
36     - total_iterations (int): Total number of iterations the algorithm will run for
37     - constraints (bool): Whether to satisfy constraints with Metropolis criterion or
not
38     """
39     self.func = objective_function
40     self.x_dim = x_dim
41     self.lb = range[0]
42     self.ub = range[1]
43     self.alpha = alpha
44     self.omega = omega
45     self.num_replicas = num_replicas
46     self.num_chains = num_chains
47     self.exchange_param = exchange_param
48     self.total_iterations = total_iterations
49     self.constraints = constraints
50
51     # The update step suggested by Parks et al. (1990) requires control variables to
be scaled to [0, 1]
52     # Therefore, we need to scale up solutions to original range when evaluating the
objective function
53     self.scale_up = lambda x: x * (self.ub - self.lb) + self.lb
54
55     # Energy difference =  $(f(x_{\text{new}}) - f(x)) / (k * d * T)$ , the term in the exponent of
the Metropolis criterion
56     k = 1.38064852e-23 # Boltzmann constant
57     self.deltaE = lambda x, x_new, d, T: (self.func(self.scale_up(x_new)) - self.func(
self.scale_up(x))) / (k * d * T)
58
59     # Check if temperature schedule type is valid
60     if schedule_type not in ['Geometric', 'Power']:
61         raise ValueError("Invalid progression type")
62
63     # Dictionaries to map string to function call. Function imported from directory
files
64     schedule_mapping = {'Geometric': geometric_progression,
65                         'Power': power_progression
66                         }
67
68     # Check if power term is valid, if it's needed
69     if schedule_type == 'Power':
70         if power_term is None or power_term < 1:
71             raise ValueError(f"Power term not specified correctly: {power_term}. Must
be greater than 1.")
72
73     # Generate temperature schedule
74     self.temperature_schedule = schedule_mapping[schedule_type](num_replicas,
power_term)
75
76     # Check if exchange procedure is valid
77     if exchange_procedure not in ['Periodic', 'Stochastic', 'Always']:
78         raise ValueError(f"Invalid exchange procedure: {exchange_procedure}.")
79
80     # Check if exchange parameter is valid, should be either a probability or a
percentage
81     if exchange_param < 0 or exchange_param > 1:
82         raise ValueError(f"Exchange parameter must be between 0 and 1: {exchange_param
}.".")
83
84     # Dictionaries to map string to function call. Function imported from directory
files
85     exchange_mapping = {'Periodic': period_exchange,
86                        'Stochastic': stochastic_exchange,
87                        'Always': always_exchange
88                        }
89
90     # Set exchange procedure

```

```

91     self.exchange_procedure = exchange_mapping[exchange_procedure]
92
93     # Initialise solutions
94     self.initialise_solutions()
95
96     def initialise_solutions(self):
97         """
98         Initialise solutions for each replica.
99         """
100         # Array of solutions for each replica, between 0 and 1 recommended by Parks et al.
101         (1990)
102         self.current_solutions = np.random.uniform(0, 1, (self.num_replicas, self.
103         num_chains, self.x_dim))
104
105         # Diagonal matrix of max. allowable step sizes for each solution
106         # Each item in the matrix pertains to the max step size for each dimension of the
107         solution
108         D = np.eye(self.x_dim)
109
110         # Copy matrix for each solution in each replica
111         # Each one will be updated individually, as the algorithm progresses
112         self.max_change = np.tile(D, (self.num_replicas, self.num_chains, 1, 1))
113
114     def get_best_solution(self, all_solutions=None):
115         """
116         Return best solution out of all replicas and solutions.
117
118         Parameter, all_solutions, is optional. All solutions are already at hand in the
119         fitness function,
120         so we can directly pass it into this to reduce overhead. If not passed in,
121         get_all_solutions() is called.
122         """
123         if all_solutions is None:
124             # Get a list of all solutions
125             all_solutions = self.get_all_solutions()
126
127         # Evaluate function at each solution
128         all_solutions_eval = np.array([self.func(x) for x in all_solutions])
129
130         # Find index of best solution
131         best_idx = np.argmax(all_solutions_eval)
132
133         # Only consider candidates from feasible region
134         if self.constraints:
135             while not satisfy_constraints(all_solutions[best_idx]):
136                 all_solutions_eval[best_idx] = -np.inf
137                 best_idx = np.argmax(all_solutions_eval)
138
139         # Return best solution
140         return all_solutions[best_idx]
141
142     def metropolis_criterion(self, x, x_new, T, T_new=None):
143         """
144         Metropolis-Hastings criterion for accepting new solution.
145         Acceptance probability as advised by Simulated Annealing lecture notes (Parks et
146         al.)
147
148         Despite efforts to avoid overflow, the small Boltzmann constant still causes
149         problems.
150         However, the algorithm needs to be sensitive, and only seems to work with the
151         inclusion of k.
152         It works completely fine, but it is not ideal from a performance perspective.
153
154         Parameters:
155         - x (np.array): Current solution
156         - x_new (np.array): New solution
157         - T (float): Temperature
158         - T_new (float): New temperature, if a replica exchange has occurred

```

```

152     Returns:
153     - bool: Whether to accept new solution or not
154     """
155     # If constraints are to be satisfied, check if new solution satisfies constraints
156     if self.constraints:
157         if not satisfy_constraints(self.scale_up(x_new)):
158             return False # Reject solution if it doesn't satisfy constraints
159
160     # Calculate the L2 norm of the step size
161     d = np.linalg.norm(x_new - x)
162
163     # Avoid division by 0, if denominator is too small, probability of acceptance is 1
164     if T * d < 1e-6:
165         return True
166
167     # If a replica exchange has occurred
168     elif T_new is not None:
169
170         # Avoid division by zero
171         if T < 1e-6:
172             return True
173         elif T_new < 1e-6:
174             return False
175         else:
176             # More likely to accept replica exchanges that see a small change in
177             temperature
178             # Small change in temp = larger acceptance probability, p = exp(-deltaE /
179             delta_T)
180             delta_T = ((1 / T) - (1 / T_new))*(-1)
181
182             # Avoid division by 0
183             if delta_T * d < 1e-6:
184                 return True
185
186             # Acceptance probability, change in temp is sent into denominator of
187             acceptance probability
188             return np.random.uniform() < min(1, np.exp(self.deltaE(x, x_new, d, delta_T)))
189
190     else:
191
192         # Acceptance probability with no replica exchange
193         return np.random.uniform() < min(1, np.exp(self.deltaE(x, x_new, d, T)))
194
195     def update_max_change(self, x, x_new, i, j):
196         """
197         Function to update max. allowable step size whenever a solution is accepted.
198         See Parks et al. (1990) for more details.
199         """
200         # Absolute difference between new and current solution
201         R = np.diag(np.abs(x_new - x))
202
203         # Update max. allowable step size
204         self.max_change[i][j] = (1-self.alpha) * self.max_change[i][j] + self.alpha * self
205         .omega * R
206
207     def update_chains(self):
208         """
209         One step of the algorithm. Update solutions for each replica. Easily
210         parallelisable.
211         """
212         # Loop through each replica, i.e. each temperature
213         for i in range(self.num_replicas):
214
215             # Loop through each solution in replica
216             for j in range(self.num_chains):
217
218                 # Generate new solution, [ x_new = x + D * U(-1, 1) ], where D is max.
219                 allowable step size
220                 x_new = self.current_solutions[i, j] + self.max_change[i][j] @ np.random.

```

```

uniform(-1, 1, self.x_dim)
215
216         # Metropolis-Hastings criterion
217         if self.metropolis_criterion(self.current_solutions[i, j], x_new, self.
temperature_schedule[i]):
218
219             # Update current solution if new solution is accepted
220             self.current_solutions[i, j] = x_new
221
222             # Update max. allowable step size if new solution is accepted
223             self.update_max_change(self.current_solutions[i, j], x_new, i, j)
224
225
226     def replica_exchange(self, iter):
227         """
228         Function to exchange solutions between replicas.
229         """
230         # Call exchange procedure
231         self.exchange_procedure(self, iter)
232
233     def get_fitness(self):
234         """
235         Function to calculate average and min fitness of population. Fitness is negative
of objective function.
236         """
237         all_solutions = self.get_all_solutions()
238
239         avg = np.mean([-self.func(x) for x in all_solutions])
240         best_solution = self.get_best_solution(all_solutions)
241         min = -self.func(best_solution)
242
243         return avg, min
244
245     def get_all_solutions(self):
246         """
247         Return all solutions reshaped into a n-dim array, scaled up to original range of
problem.
248         """
249         return self.scale_up(self.current_solutions.flatten().reshape(-1, self.x_dim))

```

Listing 4: PT.py

```

1  """
2  Candidate No : 5730E, Module: 4M17
3
4  Description :
5      This file contains the functions for exchanging solutions between replicas.
6  """
7
8  import numpy as np
9
10 def swap(PT):
11     """
12     Swap solutions between adjacent replicas, (subject to Metropolis criterion).
13     """
14     # Loop through each replica
15     for i in range(PT.num_replicas - 1):
16
17         # Temperatures of adjacent replicas to swap
18         T_1 = PT.temperature_schedule[i]
19         T_2 = PT.temperature_schedule[i + 1]
20
21         # Loop through each solution in replica
22         for j in range(PT.num_chains):
23
24             # If solutions are the same, no need to swap
25             if np.array_equal(PT.current_solutions[i, j], PT.current_solutions[i + 1, j]):
26                 continue
27
28             # Check Metropolis criterion for both directions.

```

```

29         # Acceptance is now dependent on temp difference, so now both temps are sent
    in as args
30         check_criterion = [
31             PT.metropolis_criterion(PT.current_solutions[i, j], PT.current_solutions[i
+ 1, j], T_1, T_2),
32             PT.metropolis_criterion(PT.current_solutions[i+1, j], PT.current_solutions
[i, j], T_2, T_1)
33         ]
34
35         # Only swap solutions if both directions satisfy Metropolis criterion
36         if all(check_criterion):
37
38             # Swap solutions
39             PT.current_solutions[i, j], PT.current_solutions[i + 1, j] = PT.
current_solutions[i + 1, j], PT.current_solutions[i, j]
40
41             # Update max. allowable step size
42             PT.update_max_change(PT.current_solutions[i, j], PT.current_solutions[i +
1, j], i, j)
43
44
45     def period_exchange(PT, iter):
46         """
47         Periodic exchange of solutions between replicas, swaps solutions every PT.
exchange_param*NUM_ITERS.
48         """
49         # Check if it is time to swap solutions between replicas
50         if iter % (PT.exchange_param * PT.total_iterations) == 0:
51             swap(PT)
52
53     def stochastic_exchange(PT, iter):
54         """
55         Random exchange of solutions between replicas, with probability PT.exchange_param.
56         """
57         # Chance to swap solutions between replicas
58         if np.random.uniform() < PT.exchange_param:
59             swap(PT)
60
61     def always_exchange(PT, iter):
62         """
63         Always exchange solutions between replicas.
64         """
65         swap(PT)

```

Listing 5: replica.exchange.functions.py

```

1     """
2     Candidate No : 5730E, Module: 4M17
3
4     Description : This file contains temperature scheduling functions for parallel tempering.
5                   Each temperature scheduling function returns a list of temperatures ranging
6                   from 0 to 1.
7     """
8     import numpy as np
9
10    def power_progression(num_replicas, p=1):
11        """
12        Uniform progression for temperature scheduling.
13        Returns (i / num_replicas)^p for i in [0, num_replicas].
14
15        Parameters:
16        - num_replicas (int): Number of replicas
17
18        Returns:
19        - schedule (list): List of temperatures
20        """
21        return np.linspace(0, 1, num_replicas)**p
22
23    def geometric_progression(num_replicas, p=1):
24        """

```

```

24     Geometric progression for temperature scheduling.
25     Returns  $2^i / 2^{\text{num\_replicas}}$  for  $i$  in  $[0, \text{num\_replicas}]$ .
26     (Was not used, because power progression allows for more flexibility).
27
28     Parameters:
29     - num_replicas (int): Number of replicas
30
31     Returns:
32     - schedule (list): List of temperatures
33     """
34     return np.logspace(-2, 0, num_replicas)

```

Listing 6: temp_prog_functions.py

7.2 utils

```

1     """
2     Candidate No : 5730E, Module: 4M17
3
4     Description :
5         This file contains some helper functions for the project.
6     """
7
8     import numpy as np
9     import os
10    from src.functions import KBF_function
11
12    def satisfy_constraints(x):
13        """
14        Function to check if a given vector x satisfies the constraints of the problem.
15
16        Args:
17        - x (np.ndarray): Vector to check.
18
19        Returns:
20        - bool: True if x satisfies constraints, False otherwise.
21        """
22
23        # List of boolean values for each constraint satisfied
24        constraints = [
25            np.all(x >= 0) and np.all(x <= 10),
26            np.prod(x) > 0.75,
27            np.sum(x) < 15 * x.shape[0] / 2,
28        ]
29
30        # Return True if all constraints are satisfied, False otherwise
31        return all(constraints)
32
33    def evaluate_2D(func, x_range=(0,10), constraints=False):
34        """
35        Function for generating a meshgrid and evaluating a function in  $R^2$ .
36        """
37
38        # Create a meshgrid
39        x1 = np.linspace(x_range[0], x_range[1], 100)
40        x2 = np.linspace(x_range[0], x_range[1], 100)
41        X1, X2 = np.meshgrid(x1, x2)
42
43        # Compute the function values
44        f = np.zeros_like(X1)
45        for i in range(X1.shape[0]):
46            for j in range(X1.shape[1]):
47                f[i, j] = func(np.array([X1[i, j], X2[i, j]]))
48
49        # If constraints are enabled, set f to nan if constraints are not satisfied
50        if constraints == True:
51            if not satisfy_constraints(np.array([X1[i, j], X2[i, j]])):
52                f[i, j] = np.nan
53
54        return X1, X2, f

```



```

55
56 def create_figure_directories_CGA(name, selection_methods, mating_procedures, iters_list):
57     """
58     Function for creating directories for figures generated in my simulations.
59     """
60     # Create parent directory
61     if not os.path.exists('figures'):
62         os.makedirs('figures')
63
64     # Create directory for specific function
65     function_dir = os.path.join('figures', name)
66     if not os.path.exists(function_dir):
67         os.makedirs(function_dir)
68
69     # Create directory for each number of iterations
70     for iters in iters_list:
71         iters_dir = os.path.join(function_dir, f'{iters}_iters')
72         if not os.path.exists(iters_dir):
73             os.makedirs(iters_dir)
74
75     # Create directories for each selection method and mating procedure
76     for selection_method in selection_methods:
77         for mating_procedure in mating_procedures:
78             for iters in iters_list:
79                 selection_dir = os.path.join(function_dir, f'{str(iters)}_iters',
selection_method)
80                 if not os.path.exists(selection_dir):
81                     os.makedirs(selection_dir)
82
83                 mating_dir = os.path.join(selection_dir, mating_procedure)
84                 if not os.path.exists(mating_dir):
85                     os.makedirs(mating_dir)
86
87 def create_figure_directories_PT(name, exchange_procedures, schedule_types, iters_list):
88     """
89     Function for creating directories for figures generated in my simulations.
90     """
91     # Create parent directory
92     if not os.path.exists('figures'):
93         os.makedirs('figures')
94
95     # Create directory for specific function
96     function_dir = os.path.join('figures', name)
97     if not os.path.exists(function_dir):
98         os.makedirs(function_dir)
99
100    # Create directory for each number of iterations
101    for iters in iters_list:
102        iters_dir = os.path.join(function_dir, f'{iters}_iters')
103        if not os.path.exists(iters_dir):
104            os.makedirs(iters_dir)
105
106    # Create directories for each exchange procedure and schedule type
107    for exchange_procedure in exchange_procedures:
108        for schedule_type in schedule_types:
109            for iters in iters_list:
110                exchange_dir = os.path.join(function_dir, f'{str(iters)}_iters',
exchange_procedure)
111                if not os.path.exists(exchange_dir):
112                    os.makedirs(exchange_dir)
113
114                schedule_dir = os.path.join(exchange_dir, schedule_type)
115                if not os.path.exists(schedule_dir):
116                    os.makedirs(schedule_dir)
117
118 def generate_initial(x_dim=8, pop_size=250):
119     """
120     Function for generating a collection of populations for comparison section.
121     Each population is generated using a different random seed.

```

```

122     Only initialisations which do not contain a solution close to the optimum are kept.
123
124     Args:
125     - x_dim (int): Dimension of the vectors.
126     - pop_size (int): Size of each collection of initial solutions.
127
128     Returns:
129     - initialisations (list): List of initial populations.
130     - seeds (list): List of seeds used for each initialisation.
131     """
132
133     # Initialise variables
134     seeds = []
135     initialisations = []
136
137     # Generate 50 initialisations
138     i = 0
139     while i < 50:
140         # Generate a random number seed
141         seed = np.random.randint(0, 1000000)
142
143         # Set the seed
144         np.random.seed(seed)
145
146         # Generate a random initialisation
147         initialisation = np.random.uniform(0, 10, (pop_size, x_dim))
148
149         # Make sure no solution is close to the optimum
150         f_x = np.array([KBF_function(x) for x in initialisation])
151
152         if np.max(f_x) < 0.3:
153             initialisations.append(initialisation)
154             seeds.append(seed)
155             i += 1
156
157     return initialisations, seeds

```

Listing 7: helper_functions.py

```

1     """
2     Candidate No : 5730E, Module: 4M17
3
4     Description :
5         This file contains various functions used for plotting figures.
6     """
7
8     import matplotlib.pyplot as plt
9     from mpl_toolkits.mplot3d import Axes3D
10    from matplotlib import rc
11    import seaborn as sns
12    from src.utils.helper_functions import evaluate_2D
13    from src.algorithms.PT.temp_prog_functions import power_progression
14
15    # Set LaTeX font
16    rc('font', **{'family': 'serif', 'serif': ['Computer Modern']}, size=14)
17    rc('text', usetex=True)
18
19    def plot_2D(X1, X2, f, name, constraints=False):
20        """
21        Function for visualising a function in  $R^2$ .
22        Creates both a contour plot and a 3D surface plot.
23
24        Args:
25        - X1 (np.ndarray): Meshgrid of x1 values.
26        - X2 (np.ndarray): Meshgrid of x2 values.
27        - f (np.ndarray): Function values.
28        - name (str): Name of function.
29        - constraints (bool): Whether to plot with carved out feasible region or not.
30        """
31

```

```

32 # Change visualisation depending on whether we're plotting the carved out feasible
region or not
33 if constraints == True:
34     name_png = f'{name} Feasible'
35     angle = -10
36     elevation = 30
37
38 else:
39     name_png = name
40     angle = 30
41     elevation = 20
42
43 # Plot contour
44 plt.figure()
45 plt.gca().set_facecolor('xkcd:light grey')
46 plt.contourf(X1, X2, f, 100, cmap='jet')
47 plt.xlabel(r'$x_1$')
48 plt.ylabel(r'$x_2$')
49 plt.title(f'{name_png} Contour Plot')
50 cbar = plt.colorbar()
51 cbar.set_label(r'$f(x_1, x_2)$')
52 plt.savefig(f'figures/{name}/{name_png}_contour.png')
53
54 # Plot 3D surface
55 fig = plt.figure()
56 ax = fig.add_subplot(111, projection='3d')
57 ax.plot_surface(X1, X2, f, cmap='rainbow', edgecolor='k')
58 ax.set_xlabel(r'$x_1$')
59 ax.set_ylabel(r'$x_2$')
60 ax.zaxis.set_rotate_label(False)
61 ax.set_zlabel(r'$f(x_1, x_2)$', rotation=90)
62 ax.set_title(f'{name_png} 3D Plot')
63 ax.view_init(elev=elevation, azimuth=angle) # Set the view angle
64 plt.savefig(f'figures/{name}/{name_png}_surf.png')
65
66 def plot_population(population, plot, best=None, last=None):
67     """
68     Function for overlaying a particular generation from the CGA's optimisation on a
69     contour plot in R^2.
70
71     Args:
72     - population (np.ndarray): Population of individuals.
73     - plot (matplotlib.pyplot): Plot to overlay on.
74     - best (np.ndarray): Best individual.
75     - last (np.ndarray): Last individual in collection, (for highlighting to visualise
76     MCMC moves).
77     """
78
79     # Plot population
80     plot.scatter(population[:,0], population[:,1], marker='x', label='Population', color='
red')
81
82     # Plot circle around best individual
83     if best is not None:
84         plot.plot(best[0], best[1], marker='o', markersize=8, label='Best', color='green')
85         plot.add_patch(plt.Circle((best[0], best[1]), 0.5, color='green', fill=False))
86
87     # Plot circle around last individual
88     if last is not None:
89         plot.plot(last[0], last[1], marker='o', markersize=8, label='Tracked', color='
yellow')
90         plot.add_patch(plt.Circle((last[0], last[1]), 0.5, color='yellow', fill=False))
91
92     plot.legend()
93
94 def plot_sub_contour(X1, X2, f, plot, x_range=(0,10), colour='gray'):
95     """
96     Function for plotting the grey contour plot which will be overlayed with the
97     population during optimisation.

```

```

95
96     Args:
97     - X1 (np.ndarray): Meshgrid of x1 values.
98     - X2 (np.ndarray): Meshgrid of x2 values.
99     - f (np.ndarray): Function values.
100     - plot (matplotlib.pyplot): A subplot within the grid of contours to plot the contour
    on.
101     - x_range (tuple): Range of x values.
102     """
103     plot.contourf(X1, X2, f, 100, cmap=colour)
104     plot.set_facecolor('xkcd:light grey')
105     plot.set_xlabel(r'$x_1$')
106     plot.set_ylabel(r'$x_2$')
107     plot.set_xlim(x_range)
108     plot.set_ylim(x_range)
109
110 def plot_fitness(avg_fitness, min_fitness, type, PT=False):
111     """
112     Function for plotting the evolution of the average and minimum fitness of a population
113     .
114     Args:
115     - avg_fitness (np.ndarray): Array of average fitness values.
116     - min_fitness (np.ndarray): Array of minimum fitness values.
117     - name (str): Name of function.
118     - type (list): List of parameters used for the optimisation.
119     - PT (bool): Whether the optimisation was performed using parallel tempering or not.
120     """
121     plt.figure()
122     sns.set_style('darkgrid')
123     plt.plot(avg_fitness, label='Average Fitness')
124     plt.plot(min_fitness, label='Minimum Fitness')
125     plt.xlabel('Iteration')
126     plt.ylabel(r'Fitness = $-f(x_1, x_2)$')
127
128     # Set naming based on algorithm used
129     if PT:
130         hyperparams = ['Exchange Procedure', 'Schedule', 'Exchange Param', 'Power Term']
131
132     else:
133         hyperparams = ['Selection', 'Mating', 'Mutation Rate', 'Crossover Prob']
134
135     plt.title("Evolution of Fitness to " + type[0] + " Function, \n"
136             + f"[{hyperparams[0]}]: " + r"\textbf{" + type[2]
137             + r"}", " + hyperparams[1] + r": \textbf{" + type[3]
138             + r"}", " + hyperparams[2] + r": \textbf{" + str(type[4])
139             + r"}", " + hyperparams[3] + r": \textbf{" + str(type[5])
140             + r"}]", fontsize=12)
141
142     plt.legend()
143     plt.savefig(f'figures/{type[0]}/{str(type[1])}_iters/{type[2]}/{type[3]}/{type[4]}_{
type[5]}_Fitness.png')

```

Listing 8: plotting_functions.py

7.3 functions

```

1     """
2     Candidate No : 5730E, Module: 4M17
3
4     Description :
5         This file contains the implementation of the Keane's Bump Function and the Rosenbrock
        function.
6     """
7
8
9     import numpy as np
10
11     def KBF_function(x, eps=1e-8):
12         """

```

```

13     Function implements Keane's Bump Function for a given input vector x of shape n x 1.
14
15     Args:
16         x (np.ndarray): Input vector of shape n x 1.
17         eps (float): Small value to avoid division by zero.
18
19     Returns:
20         f (float): Function value at x.
21     """
22
23     # Get the number of dimensions
24     n = x.shape[0]
25
26     # Compute the numerator
27     num = np.sum(np.cos(x)**4) - 2*np.prod(np.cos(x)**2)
28
29     # Compute the denominator
30     den = np.sqrt(np.sum(np.arange(1, n+1)*x**2))
31
32     # Avoid division by zero
33     if den == 0:
34         den = eps
35
36     # Compute the function value
37     f = np.abs(num/den)
38
39     return f
40
41 def Rosenbrock_function(x):
42     """
43     Function implements the Rosenbrock function for a given input vector x of shape n x 1.
44     Used for testing optimisation algorithms.
45
46     Args:
47         x (np.ndarray): Input vector of shape n x 1.
48
49     Returns:
50         f (float): Function value at x.
51     """
52
53     # Get the number of dimensions
54     n = x.shape[0]
55
56     # Compute the function value
57     f = 0
58     for i in range(n-1):
59         f += 100*(x[i+1] - x[i]**2)**2 + (1 - x[i])**2
60
61     return f

```

Listing 9: functions.py

7.4 Experiments

7.4.1 CGA Tuning Experiments

```

1     """
2     Candidate No : 5730E, Module: 4M17
3
4     Description :
5         This file serves as a platform to run multiple simulations of the CGA algorithm.
6         Used to generate the results for the table and figures in Section 3.2 of the report.
7     """
8
9     import sys; sys.path.append('.')
10    from src.utils.helper_functions import evaluate_2D, create_figure_directories_CGA
11    from src.utils.plotting_functions import plot_2D, plot_sub_contour, plot_population,
12    plot_fitness
13    from src.functions import KBF_function, Rosenbrock_function

```

```

13 from src.algorithms.CGA.CGA import ContinuousGeneticAlgorithm
14
15 import numpy as np
16 import matplotlib.pyplot as plt
17 from multiprocessing import Pool
18 import pandas as pd
19 import seaborn as sns
20
21 # Hyperparameters
22 X_RANGE=(0,10)
23 FUNCTION = KBF_function
24 POPULATION_SIZE = 250
25 CHROMOSOME_LENGTH = 2
26 NUM_PARENTS = POPULATION_SIZE // 4 # 25% of population size
27 MUTATION_RATE_LIST = [0.05, 0.1]
28 CROSSOVER_PROB_LIST = [0.7, 0.65]
29 SELECTION_METHOD_LIST = ['Proportional', 'Tournament', 'SRS']
30 MATING_PROCEDURE_LIST = ['Crossover', 'Heuristic Crossover']
31 ITERS_LIST = [5, 10, 100]
32 TOURNAMENT_SIZE = POPULATION_SIZE // 4 # 25% of population size
33
34 NAME = 'Rosenbrock' if FUNCTION == Rosenbrock_function else 'KBF'
35
36 # Make sure NUM_PARENTS is a multiple of 2
37 if NUM_PARENTS % 2 != 0:
38     NUM_PARENTS += 1
39
40 # Create directories for figures
41 create_figure_directories_CGA(NAME, SELECTION_METHOD_LIST, MATING_PROCEDURE_LIST,
42                                ITERS_LIST)
43
44 # 2D Visualisation of function
45 X1, X2, f = evaluate_2D(FUNCTION, x_range=X_RANGE)
46 plot_2D(X1, X2, f, name=NAME)
47
48 # Visualise with carved out feasible region
49 X1, X2, f_feasible = evaluate_2D(FUNCTION, x_range=X_RANGE, constraints=True)
50 plot_2D(X1, X2, f_feasible, name=NAME, constraints=True)
51
52 ### Function used to generate results for table and figures in Section 3.2/Appendix of the
53 report ###
54 def selection_mating_tuning(params):
55     """
56     Parallelisable function to run multiple simulations of the CGA algorithm and
57     assess the impact of different hyperparameters on the algorithm's performance.
58     The results are saved to a csv file and figures are generated to visualise the results
59     .
60     """
61
62     # Parse parameters
63     SELECTION_METHOD, MATING_PROCEDURE, MUTATION_RATE, CROSSOVER_PROB, NUM_ITERS = params
64
65     # Instantiate CGA
66     CGA = ContinuousGeneticAlgorithm(
67         population_size=POPULATION_SIZE,
68         chromosome_length=CHROMOSOME_LENGTH,
69         num_parents=NUM_PARENTS,
70         objective_function=FUNCTION,
71         tournament_size=TOURNAMENT_SIZE,
72         range=X_RANGE,
73         mutation_rate=MUTATION_RATE,
74         crossover_prob=CROSSOVER_PROB,
75         selection_method=SELECTION_METHOD,
76         mating_procedure=MATING_PROCEDURE,
77     )
78
79     # Evaluate initial population
80     CGA.evaluate_fitness()

```

```

79 # Make a grid of 2D plots to show evolution of population
80 PLOT_EVERY = NUM_ITERS // 5
81 num_plots = (NUM_ITERS // PLOT_EVERY)
82 fig, axs = plt.subplots(1, num_plots, figsize=(20, 5))
83 fig.suptitle("Evolution of Population, \n"
84             + r"[Selection: \textbf{" + SELECTION_METHOD
85             + r"}], Mating: \textbf{" + MATING_PROCEDURE
86             + r"}], Mutation Rate: \textbf{" + str(MUTATION_RATE)
87             + r"}], Crossover Prob: \textbf{" + str(CROSSOVER_PROB)
88             + r"}]", fontsize=18)
89
90 # Plot grey contour plots of function on each subplot in grid
91 # This will be overlayed with populations at different iterations
92 for idx in range(num_plots):
93     plot_sub_contour(X1, X2, f_feasible, plot=axs[idx], x_range=X_RANGE)
94
95 # Initialise arrays to store fitness values
96 avg_fitness = np.zeros(NUM_ITERS)
97 min_fitness = np.zeros(NUM_ITERS)
98
99 for iter in range(NUM_ITERS):
100
101     # Overlay population on grey contour every "PLOT_EVERY" iterations. (Periodic
102     break to allow for visualisation).
103     if iter % PLOT_EVERY == 0:
104         plot_num = (iter // PLOT_EVERY)
105         idx = plot_num % num_plots
106         axs[idx].set_title(f'Iteration: {iter}')
107         plot_population(CGA.population, plot=axs[idx], best=CGA.best_individual)
108
109     # Evolve population
110     CGA.evolve()
111
112     # Update fitness arrays
113     avg_fitness[iter] = np.mean(CGA.fitness)
114     min_fitness[iter] = CGA.min_fitness
115
116     plt.tight_layout()
117     plt.savefig(f'figures/{NAME}/{str(NUM_ITERS)}_iters/{SELECTION_METHOD}/{
118     MATING_PROCEDURE}/{MUTATION_RATE}_{CROSSOVER_PROB}_Population.png')
119
120     # Plot fitness evolution with iteration
121     plot_fitness(avg_fitness, min_fitness, type=(NAME, NUM_ITERS, SELECTION_METHOD,
122     MATING_PROCEDURE, MUTATION_RATE, CROSSOVER_PROB))
123
124     # Return results to dataframe
125     return {
126         'Selection Method': SELECTION_METHOD,
127         'Mating Procedure': MATING_PROCEDURE,
128         'Mutation Rate': MUTATION_RATE,
129         'Crossover Rate': CROSSOVER_PROB,
130         'Iterations': NUM_ITERS,
131         'Final Avg Fitness': avg_fitness[-1],
132         'Final Min Fitness': min_fitness[-1]
133     }
134
135 ### Function used to generate fitness evolutions in Section 3.2 of the report ###
136 def plot_fitnesses():
137     """
138     Function to plot fitness evolution for each selection method.
139     """
140     for MATING in MATING_PROCEDURE_LIST:
141
142         # Make a plot for each mating procedure to show fitness evolution for the three
143         selection method
144         plt.figure(figsize=(14, 10))
145         sns.set_style('darkgrid')
146         plt.title(f'Average Fitness Evolution for each Selection Method on {NAME} Function
147         \n'

```



```

143         + r"Mating Procedure held as \textbf{" + MATING + r"}", fontsize=24)
144     plt.xlabel('Iteration', fontsize=20)
145     plt.ylabel('Average Fitness', fontsize=20)
146
147     for SELECTION_METHOD in SELECTION_METHOD_LIST:
148         # Instantiate CGA
149         CGA = ContinuousGeneticAlgorithm(
150             population_size=POPULATION_SIZE,
151             chromosome_length=CHROMOSOME_LENGTH,
152             num_parents=NUM_PARENTS,
153             objective_function=FUNCTION,
154             tournament_size=TOURNAMENT_SIZE,
155             range=X_RANGE,
156             mutation_rate=0.05,
157             crossover_prob=0.7,
158             selection_method=SELECTION_METHOD,
159             mating_procedure=MATING,
160             )
161
162         # Evaluate initial population
163         CGA.evaluate_fitness()
164
165         # Initialise arrays to store fitness values
166         avg_fitness = np.zeros(100)
167         min_fitness = np.zeros(100)
168
169         for iter in range(100):
170             # Evolve population
171             CGA.evolve()
172
173             # Update fitness arrays
174             avg_fitness[iter] = np.mean(CGA.fitness)
175             min_fitness[iter] = CGA.min_fitness
176
177         # Plot fitness evolution with iteration
178         sns.lineplot(x=range(100), y=avg_fitness, label=SELECTION_METHOD)
179
180         plt.legend(fontsize=16)
181         plt.savefig(f'figures/{NAME}/Fitness_Evolution_{MATING}.png')
182
183     ### Function used to generate heat maps in Section 3.2 of the report ###
184     def rate_prob_mesh(params):
185         """
186         Parallelisable function to create mesh grids for heat map plots to assess
187         the impact of mutation rate and crossover probability on the
188         algorithm's performance.
189         """
190
191         # Parse parameters
192         i, j, MUTATION_RATE, CROSSOVER_PROB = params
193
194         # Instantiate CGA
195         CGA = ContinuousGeneticAlgorithm(
196             population_size=POPULATION_SIZE,
197             chromosome_length=CHROMOSOME_LENGTH,
198             num_parents=NUM_PARENTS,
199             objective_function=FUNCTION,
200             tournament_size=TOURNAMENT_SIZE,
201             range=X_RANGE,
202             mutation_rate=MUTATION_RATE,
203             crossover_prob=CROSSOVER_PROB,
204             selection_method=SELECTION_METHOD,
205             mating_procedure=MATING_PROCEDURE,
206             )
207
208         # Evaluate initial population
209         CGA.evaluate_fitness()
210
211         # Evolve population

```

```

212         for iter in range(100):
213             CGA.evolve()
214
215         # Return the results
216         return i, j, np.mean(CGA.fitness), CGA.min_fitness
217
218     ### Now run the above three functions ###
219
220     # Create a list of parameter combinations
221     params_list = [(SELECTION_METHOD, MATING_PROCEDURE, MUTATION_RATE, CROSSOVER_PROB,
222                     NUM_ITERS)
223                    for SELECTION_METHOD in SELECTION_METHOD_LIST
224                    for MATING_PROCEDURE in MATING_PROCEDURE_LIST
225                    for MUTATION_RATE in MUTATION_RATE_LIST
226                    for CROSSOVER_PROB in CROSSOVER_PROB_LIST
227                    for NUM_ITERS in ITERS_LIST]
228
229     print('Starting simulations for table and figure generation...')
230
231     # Create a pool of worker processes
232     pool = Pool()
233
234     # Create a dataframe to store results for CSV table
235     results = pd.DataFrame(columns=['Selection Method',
236                                   'Mating Procedure',
237                                   'Mutation Rate',
238                                   'Crossover Rate',
239                                   'Iterations',
240                                   'Final Avg Fitness',
241                                   'Final Min Fitness'])
242
243     # Run the simulations in parallel
244     for result in pool.map(selection_mating_tuning, params_list):
245         results.loc[len(results)] = result # Append result to dataframe
246
247     # Close the pool
248     pool.close()
249     pool.join()
250
251     # Save results to csv file
252     results.to_csv(f'figures/{NAME}/CGAresults.csv')
253
254     print('Done!')
255
256     print('Starting simulations for fitness plots generation')
257
258     # Plot fitness evolution for each selection method
259     plot_fitnesses()
260
261     print('Done!')
262
263     print('Starting simulations for contour plot generation')
264
265     # Optimal selection and mating chosen as Tournament and Heuristic Crossover respectively
266     # as discussed in Section 3.2 of the report
267     SELECTION_METHOD = 'Tournament'
268     MATING_PROCEDURE = 'Heuristic Crossover'
269
270     # Particular mutation rates and crossover probabilities I want to test
271     MUTATION_RATE_LIST = [0.01, 0.05, 0.075, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7]
272     CROSSOVER_PROB_LIST = [0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95]
273
274     # Initialise meshgrids of variables, X = mutation rate, Y = crossover probability
275     X, Y = np.meshgrid(MUTATION_RATE_LIST, CROSSOVER_PROB_LIST)
276
277     # Initialise array to store final avg and min fitnesses, used as Z in heat maps
278     AVG = np.zeros((len(MUTATION_RATE_LIST), len(CROSSOVER_PROB_LIST)))
279     MIN = np.zeros((len(MUTATION_RATE_LIST), len(CROSSOVER_PROB_LIST)))

```

```

280 # Create a pool of workers
281 pool = Pool()
282
283 # Create a list of parameters for parallel execution
284 params_list = [(i, j, MUTATION_RATE, CROSSOVER_PROB)
285                 for i, MUTATION_RATE in enumerate(MUTATION_RATE_LIST)
286                 for j, CROSSOVER_PROB in enumerate(CROSSOVER_PROB_LIST)]
287
288 # Run the simulations in parallel
289 results = pool.map(rate_prob_mesh, params_list)
290
291 # Update fitness arrays
292 for i, j, fitness, min_fitness in results:
293     AVG[i, j] = fitness
294     MIN[i, j] = min_fitness
295
296 # Close the pool
297 pool.close()
298 pool.join()
299
300 # Plot average fitness heat map
301 plt.figure(figsize=(14, 10))
302 plt.title(f'Average Final Fitness with varying Mutation Rate and Crossover Probability on
303 {NAME} Function \n'
304           + r"[Selection Method: \textbf{" + SELECTION_METHOD + r"}], "
305           + r"Mating Procedure: \textbf{" + MATING_PROCEDURE + r"}]", fontsize=18)
306 sns.heatmap(AVG, annot=True, xticklabels=MUTATION_RATE_LIST, yticklabels=
307             CROSSOVER_PROB_LIST)
308 plt.xlabel('Mutation Rate', fontsize=14)
309 plt.ylabel('Crossover Probability', fontsize=14)
310 plt.savefig(f'figures/{NAME}/AVGContour_{SELECTION_METHOD}_{MATING_PROCEDURE}.png')
311
312 # Plot minimum fitness heat map
313 plt.figure(figsize=(14, 10))
314 plt.title(f'Minimum Final Fitness with varying Mutation Rate and Crossover Probability on
315 {NAME} Function \n'
316           + r"[Selection Method: \textbf{" + SELECTION_METHOD + r"}], "
317           + r"Mating Procedure: \textbf{" + MATING_PROCEDURE + r"}]", fontsize=18)
318 sns.heatmap(MIN, annot=True, xticklabels=MUTATION_RATE_LIST, yticklabels=
319             CROSSOVER_PROB_LIST)
320 plt.xlabel('Mutation Rate', fontsize=14)
321 plt.ylabel('Crossover Probability', fontsize=14)
322 plt.savefig(f'figures/{NAME}/MINContour_{SELECTION_METHOD}_{MATING_PROCEDURE}.png')
323
324 print('Done!')

```

Listing 10: CGA_TuningExperiments.py

7.4.2 PT_TuningExperiments

```

1 """
2 Candidate No : 5730E, Module: 4M17
3
4 Description :
5     This file serves as a platform to run multiple simulations of the PT algorithm.
6     Used to generate the results for the table and figures in Section 4.2 of the report.
7 """
8
9 import matplotlib.pyplot as plt
10 import numpy as np
11 from multiprocessing import Pool
12 import pandas as pd
13 import seaborn as sns
14
15 from src.algorithms.PT.PT import ParallelTempering
16 from src.functions import KBF_function, Rosenbrock_function
17 from src.utils.helper_functions import evaluate_2D, create_figure_directories_PT
18 from src.utils.plotting_functions import plot_sub_contour, plot_population, plot_fitness,
19 plot_temp_progressions

```

```

20 # Hyperparameters
21 X_RANGE=(0,10)
22 FUNCTION = KBF_function
23 X_DIM = 2
24 NUM_REPLICAS = 10
25 NUM_SOL_PER_REPLICA = 250 // NUM_REPLICAS # 250 solutions overall, same as CGA population
26 EXCHANGE_PROCEDURE_LIST = ['Periodic', 'Stochastic']
27 EXCHANGE_PARAM_LIST = [0.1, 0.3, 0.5, 0.01]
28 TEMP_TYPE = 'Power'
29 PROGRESSION_POWER_LIST = [1, 3, 5] # 1 is uniform
30 NUM_ITERS_LIST = [100]
31
32 FUNC_NAME = 'Rosenbrock' if FUNCTION == Rosenbrock_function else 'KBF'
33
34 # Create figure directories
35 create_figure_directories_PT(FUNC_NAME, EXCHANGE_PROCEDURE_LIST, [TEMP_TYPE],
36 NUM_ITERS_LIST)
37
38 ### Function used to generate results for table and figures in Section 4.2/Appendix of the
39 report ###
40 def PT_initial_tuning(params):
41     """
42     Parallelisable function to run multiple simulations of the PT algorithm and
43     assess the impact of different hyperparameters on the algorithm's performance.
44     The results are saved to a csv file and figures are generated to visualise the results
45     .
46     """
47
48     # Parse parameters
49     EXCHANGE_PROCEDURE, EXCHANGE_PARAM, PROGRESSION_POWER, NUM_ITERS = params
50
51     # Set schedule name for files and plots
52     SCHEDULE_NAME = TEMP_TYPE + " " + str(PROGRESSION_POWER) if TEMP_TYPE == 'Power' else
53     TEMP_TYPE
54
55     # Instantiate PT object
56     PT = ParallelTempering(
57         objective_function=FUNCTION,
58         x_dim=X_DIM,
59         range=X_RANGE,
60         num_replicas=NUM_REPLICAS,
61         num_chains=NUM_SOL_PER_REPLICA,
62         exchange_procedure=EXCHANGE_PROCEDURE,
63         exchange_param=EXCHANGE_PARAM,
64         schedule_type=TEMP_TYPE,
65         power_term=PROGRESSION_POWER
66     )
67
68     # Make a grid of 5 2D plots to show evolution of population
69     PLOT_EVERY = NUM_ITERS // 5
70     num_plots = (NUM_ITERS // PLOT_EVERY)
71     fig, axs = plt.subplots(1, num_plots, figsize=(20, 5))
72     plt.suptitle(f"Evolution of Solutions, \n"
73                 + r"[Exchange Procedure: \textbf{" + EXCHANGE_PROCEDURE
74                 + r"}], Exchange Parameter: \textbf{" + str(EXCHANGE_PARAM)
75                 + r"}], Schedule: \textbf{" + TEMP_TYPE
76                 + r"}], Power Term: \textbf{" + str(PROGRESSION_POWER)
77                 + r"}]", fontsize=16)
78
79     # Plot grey contour plots of function on each subplot in grid
80     # This will be overlayed with populations at different iterations
81     X1, X2, f_feasible = evaluate_2D(FUNCTION, x_range=X_RANGE, constraints=True)
82     for idx in range(num_plots):
83         plot_sub_contour(X1, X2, f_feasible, plot=axs[idx], x_range=X_RANGE)
84
85     # Initialise arrays to store fitness values
86     avg_fitness = np.zeros(NUM_ITERS)
87     min_fitness = np.zeros(NUM_ITERS)

```

```

85     # Initial avg. and min. fitness
86     avg_fitness[0], min_fitness[0] = PT.get_fitness()
87
88     for iter in range(NUM_ITERS):
89         # Overlay population on grey contour every "PLOT EVERY" iterations. (Periodic
90         # break to allow for visualisation).
91         if iter % PLOT_EVERY == 0:
92             plot_num = (iter // PLOT_EVERY)
93             idx = plot_num % num_plots
94             axs[idx].set_title(f'Iteration: {iter}')
95             final_replica = PT.get_all_solutions()
96             plot_population(final_replica, axs[idx], best=PT.get_best_solution(), last=
97             final_replica[-1])
98
99             # Algorithm update
100             PT.update_chains()
101             PT.replica_exchange(iter)
102
103             # Update fitness arrays
104             avg_fitness[iter], min_fitness[iter] = PT.get_fitness()
105
106         plt.tight_layout()
107         plt.savefig(f'figures/{FUNC_NAME}/{str(NUM_ITERS)}_iters/{EXCHANGE_PROCEDURE}/{
108         TEMP_TYPE}/{EXCHANGE_PARAM}_{PROGRESSION_POWER}_Solutions.png')
109
110         plot_fitness(avg_fitness, min_fitness, [FUNC_NAME,
111         NUM_ITERS,
112         EXCHANGE_PROCEDURE,
113         TEMP_TYPE,
114         EXCHANGE_PARAM,
115         PROGRESSION_POWER], PT=True)
116
117     # Return results to dataframe
118     return {
119         'Exchange Procedure': EXCHANGE_PROCEDURE,
120         'Exchange Parameter': EXCHANGE_PARAM,
121         'Schedule': TEMP_TYPE,
122         'Power Term': PROGRESSION_POWER,
123         'Iterations': NUM_ITERS,
124         'Final Avg. Fitness': avg_fitness[-1],
125         'Final Min. Fitness': min_fitness[-1],
126         'Avg. Fitness Progression': avg_fitness
127     }
128
129 ### Function used to generate heatmaps in Section 4.2/Appendix of the report ###
130 def power_exchange_mesh(params):
131     """
132     Parallelisable function to create mesh grids for heat map plots to assess
133     the impact of exchange rate and temperature scheduling on the
134     algorithm's performance.
135     """
136
137     # Parse parameters
138     i, j, EXCHANGE_PROCEDURE, EXCHANGE_PARAM, PROGRESSION_POWER = params
139
140     # Instantiate PT object
141     PT = ParallelTempering(
142         objective_function=FUNCTION,
143         x_dim=X_DIM,
144         range=X_RANGE,
145         num_replicas=NUM_REPLICAS,
146         num_chains=NUM_SOL_PER_REPLICA,
147         exchange_procedure=EXCHANGE_PROCEDURE,
148         exchange_param=EXCHANGE_PARAM,
149         schedule_type='Power',
150         power_term=PROGRESSION_POWER
151     )
152
153     # Run algorithm for 100 iterations

```

```

151     for iter in range(100):
152         PT.update_chains()
153         PT.replica_exchange(i)
154
155     # Get final fitness
156     final_avg_fitness, final_min_fitness = PT.get_fitness()
157
158     return i, j, final_avg_fitness, final_min_fitness
159
160 # Create a list of all combinations of parameters
161 params_list = [(EXCHANGE_PROCEDURE, EXCHANGE_PARAM, PROGRESSION_POWER, NUM_ITERS)
162                for EXCHANGE_PROCEDURE in EXCHANGE_PROCEDURE_LIST
163                for EXCHANGE_PARAM in EXCHANGE_PARAM_LIST
164                for PROGRESSION_POWER in PROGRESSION_POWER_LIST
165                for NUM_ITERS in NUM_ITERS_LIST]
166
167 print('Starting simulations for table and figure generation...')
168
169 # Create a pool of worker processes
170 pool = Pool()
171
172 results = pd.DataFrame(columns=['Exchange Procedure',
173                                'Exchange Parameter',
174                                'Schedule',
175                                'Power Term',
176                                'Iterations',
177                                'Final Avg. Fitness',
178                                'Final Min. Fitness',
179                                'Avg. Fitness Progression'])
180
181 # Run the simulations in parallel
182 for result in pool.map(PT_initial_tuning, params_list):
183     results.loc[len(results)] = result # Append result to dataframe
184
185 # Save results to csv
186 results.to_csv(f'figures/{FUNC_NAME}/PT_initial_tuning.csv', index=False)
187
188 # Close the pool of workers
189 pool.close()
190 pool.join()
191 print('Done!')
192
193 print('Generating fitness figures...')
194
195 ### Used to generate fitness figures in Section 4.2 of the report ###
196 # Plot fitness progression for each exchange procedure with varying power term
197 for EXCHANGE_PROCEDURE in EXCHANGE_PROCEDURE_LIST:
198     plt.figure(figsize=(10, 8))
199     sns.set_style('darkgrid')
200
201     # Get results pertaining to that exchange procedure
202     periodic_results = results[results['Exchange Procedure'] == EXCHANGE_PROCEDURE]
203
204     # Get results corresponding to uniform schedule
205     periodic_results_1 = periodic_results[periodic_results['Power Term'] == 1]
206
207     # Get results corresponding to 'Exchange Parameter' = 0.1, 0.3, 0.5
208     periodic_results_1_01 = periodic_results_1[periodic_results_1['Exchange Parameter'] ==
209     0.1]
209     periodic_results_1_03 = periodic_results_1[periodic_results_1['Exchange Parameter'] ==
210     0.3]
210     periodic_results_1_05 = periodic_results_1[periodic_results_1['Exchange Parameter'] ==
211     0.5]
211
212     # Plot fitness progression for each 'Exchange Parameter'
213     plt.plot(periodic_results_1_01['Avg. Fitness Progression'].values[0], label='Exchange
214     Parameter = 0.1')
214     plt.plot(periodic_results_1_03['Avg. Fitness Progression'].values[0], label='Exchange
215     Parameter = 0.3')

```

```

215 plt.plot(periodic_results_1_05['Avg. Fitness Progression'].values[0], label='Exchange
Parameter = 0.5')
216
217 plt.title(f'Average Fitness Progression for {EXCHANGE_PROCEDURE} Exchange, \n'
218           + r"[Temp Schedule: \textbf{Uniform}]")
219 plt.xlabel('Iteration')
220 plt.ylabel('Average Fitness')
221 plt.legend()
222 plt.tight_layout()
223
224 plt.savefig(f'figures/{FUNC_NAME}/100_iters/{EXCHANGE_PROCEDURE}/
PT_Avg_Fitness_Evolution.png')
225
226 print('Done!')
227
228 print('Starting heatmap generation...')
229
230 EXCHANGE_PROCEDURE = 'Stochastic'
231 EXCHANGE_PARAM_LIST = [0.01, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5]
232 PROGRESSION_POWER_LIST = range(1, 10)
233
234 # Create mesh grid for exchange parameters and power terms
235 X, Y = np.meshgrid(EXCHANGE_PARAM_LIST, PROGRESSION_POWER_LIST)
236
237 # Initialise arrays to store fitness values
238 AVG = np.zeros((len(EXCHANGE_PARAM_LIST), len(PROGRESSION_POWER_LIST)))
239 MIN = np.zeros((len(EXCHANGE_PARAM_LIST), len(PROGRESSION_POWER_LIST)))
240
241 params_list = [(i, j, EXCHANGE_PROCEDURE, EXCHANGE_PARAM, PROGRESSION_POWER)
242                for i, EXCHANGE_PARAM in enumerate(EXCHANGE_PARAM_LIST)
243                for j, PROGRESSION_POWER in enumerate(PROGRESSION_POWER_LIST)]
244
245 # Create a pool of worker processes
246 pool = Pool()
247
248 results = pool.map(power_exchange_mesh, params_list)
249
250 for i, j, final_avg_fitness, final_min_fitness in results:
251     AVG[i, j] = final_avg_fitness
252     MIN[i, j] = final_min_fitness
253
254 # Close the pool
255 pool.close()
256 pool.join()
257
258 # Plot average fitness heat map
259 plt.figure(figsize=(14, 10))
260 plt.title(f'Average Final Fitness with varying Power Param and Exchange Param on {
FUNC_NAME} Function \n'
261           + r"[Exchange Procedure: \textbf{" + EXCHANGE_PROCEDURE + r"}]", fontsize=18)
262 sns.heatmap(AVG, annot=True, fmt='.2f', xticklabels=EXCHANGE_PARAM_LIST, yticklabels=
PROGRESSION_POWER_LIST, cmap='Blues')
263 plt.xlabel('Exchange Parameter', fontsize=16)
264 plt.ylabel('Power Parameter, p', fontsize=16)
265 plt.tight_layout()
266 plt.savefig(f'figures/{FUNC_NAME}/PT_Avg_Fitness_Heatmap_{EXCHANGE_PROCEDURE}.png')
267
268 # Plot minimum fitness heat map
269 plt.figure(figsize=(14, 10))
270 plt.title(f'Minimum Final Fitness with varying Power Param and Exchange Param on {
FUNC_NAME} Function \n'
271           + r"[Exchange Procedure: \textbf{" + EXCHANGE_PROCEDURE + r"}]", fontsize=18)
272 sns.heatmap(MIN, annot=True, fmt='.2f', xticklabels=EXCHANGE_PARAM_LIST, yticklabels=
PROGRESSION_POWER_LIST, cmap='Blues')
273 plt.xlabel('Exchange Parameter', fontsize=16)
274 plt.ylabel('Power Parameter, p', fontsize=16)
275 plt.tight_layout()
276 plt.savefig(f'figures/{FUNC_NAME}/PT_Min_Fitness_Heatmap_{EXCHANGE_PROCEDURE}.png')
277

```



```

278 print('Done!')
279
280 print('Generating optimal PT evolution and temperature schedule plot...')
281
282 ### The rest of the code is used to plot the final plot in 4.2, showing the evolution of
283 the solutions for the optimal PT ###
284 # Optimally-tuned PT
285 PT = ParallelTempering(
286     objective_function=FUNCTION,
287     x_dim=X_DIM,
288     range=X_RANGE,
289     num_replicas=NUM_REPLICAS,
290     num_chains=NUM_SOL_PER_REPLICA,
291     exchange_procedure='Always',
292     exchange_param=0.5,
293     schedule_type='Power',
294     power_term=1
295 )
296
297 # Make a grid of 10 2D plots to show evolution of population
298 PLOT_EVERY = 100 // 5
299 num_plots = 100 // PLOT_EVERY
300
301 # Two rows, one for evolution with iterations, one for solutions for each replica/
302 temperature
303 fig, axs = plt.subplots(2, num_plots, figsize=(22, 10))
304 plt.suptitle(f"Optimall-Tuned PT; Evolution with Iteration and Final Solutions per Replica
305 \n"
306             + r"[Exchange Procedure: \textbf{Always}, Schedule: \textbf{Uniform}]"
307             + r", Power Term: \textbf{" + str(1)
308             + r"}]", fontsize=16)
309
310 # On first row, plot grey contour plots of function on each subplot in grid
311 X1, X2, f_feasible = evaluate_2D(FUNCTION, x_range=X_RANGE, constraints=True)
312 for idx in range(num_plots):
313     plot_sub_contour(X1, X2, f_feasible, plot=axs[0][idx], x_range=X_RANGE)
314
315 # Make a new array of every t_index'th temperature, (excluding idx 0)
316 t_index = len(PT.temperature_schedule) // num_plots
317 temps = PT.temperature_schedule[1::t_index]
318
319 for idx in range(num_plots):
320     # Raise function to power of temperature!!! Good for visualising how the temperature
321     affects the function
322     f_temp = f_feasible ** temps[idx]
323
324     plot_sub_contour(X1, X2, f_temp, axs[1][idx], x_range=X_RANGE, colour='Greys')
325     axs[1][idx].set_title(f'Temperature = {temps[idx]:.2f}')
326     plt.colorbar(axs[1][idx].contourf(X1, X2, f_temp, 100, cmap='inferno'), ax=axs[1][idx
327 ])
328
329 # Plot evolution for 100 iterations
330 for iter in range(100):
331     if iter % PLOT_EVERY == 0:
332         plot_num = (iter // PLOT_EVERY)
333         idx = plot_num % num_plots
334         axs[0][idx].set_title(f'Iteration: {iter}')
335         final_replica = PT.get_all_solutions()
336         plot_population(final_replica, axs[0][idx], best=PT.get_best_solution(), last=
337 final_replica[-1])
338
339         PT.update_chains()
340         PT.replica_exchange(iter)
341
342 replica_solutions = PT.current_solutions[1::t_index]
343
344 # Overlay replica solutions on second row
345 for idx in range(num_plots):
346     solutions = PT.scale_up(replica_solutions[idx])

```

```

341     axs[1][idx].scatter(solutions[:, 0], solutions[:, 1], c='lime', s=15, label='Replica
    Solutions')
342
343 plt.tight_layout()
344 plt.savefig(f'figures/{FUNC_NAME}/PT_Optimal_Tuning.png')

```

Listing 11: PT.TuningExperiments.py

7.4.3 FinalComparison

```

1  """
2  Candidate No : 5730E, Module: 4M17
3
4  Description :
5      This file serves to compare the optimally-tuned CGA and PT algorithms in section 5 of
    the report.
6  """
7
8  import sys; sys.path.append('.')
9  from src.utils.helper_functions import generate_initial
10 from src.algorithms.CGA.CGA import ContinuousGeneticAlgorithm
11 from src.algorithms.PT.PT import ParallelTempering
12 from src.functions import KBF_function
13
14 import numpy as np
15 import matplotlib.pyplot as plt
16 import seaborn as sns
17 from time import time
18 import pandas as pd
19
20 # Generate initial solutions for both functions.
21 # No solution has a function value > 0.3, (away from global optimum)
22 initial_pop_list, seeds = generate_initial(x_dim=8, pop_size=250)
23
24 # Max number of iterations, as required by assignment handout
25 MAX_NUM_ITERS = 10000
26
27 # Convergence criteria
28 eps = 0.00025
29 conv_iters = 1300
30
31 ### CGA Gather Results ###
32 CGA_AvgFit_ALL = np.zeros((50, MAX_NUM_ITERS))
33 CGA_MinFit_ALL = np.zeros((50, MAX_NUM_ITERS))
34 CGA_times_ALL = np.zeros(50)
35 CGA_best_Xs = np.zeros((50, 8))
36 CGA_converg_iters = np.zeros(50)
37
38 # Run CGA 50 times with different initialisations
39 for run_iter, initialisation in enumerate(initial_pop_list):
40
41     # Instantiate optimally-tuned CGA
42     CGA = ContinuousGeneticAlgorithm(population_size = 250,
43                                     chromosome_length = 8,
44                                     num_parents = 62,
45                                     objective_function = KBF_function,
46                                     tournament_size = 62,
47                                     range=(0,10),
48                                     mutation_rate=0.1,
49                                     crossover_prob=0.65,
50                                     selection_method='Tournament',
51                                     mating_procedure='Heuristic Crossover',
52                                     constraints=True)
53
54     # Now forcibly reset population to the initialisation
55     CGA.population = initialisation
56
57     # Initialise arrays for storing fitness
58     avg_fitness = np.zeros(MAX_NUM_ITERS)
59     min_fitness = np.zeros(MAX_NUM_ITERS)

```

```

60
61     # Max number of iterations = MAX_NUM_ITERS
62     conv_count = 0
63     CGA_tic = time()
64     for i in range(MAX_NUM_ITERS):
65
66         # Evolve population
67         CGA.evolve()
68
69         # Update fitness arrays
70         avg_fitness[i] = np.mean(CGA.fitness)
71         min_fitness[i] = CGA.min_fitness
72
73         # Check if the algorithm has converged,  $|f(x) - f(x_{prev})| < \epsilon$  for 'conv_iters'
74         iterations
75         if i != 0 and np.linalg.norm(min_fitness[i] - min_fitness[i-1]) < eps:
76             conv_count += 1
77         else:
78             conv_count = 0
79
80         # If the algorithm has converged, store the iteration at which it converged
81         if conv_count == conv_iters:
82             CGA_converg_iters[run_iter] = i - conv_iters
83
84         # Time taken to run algorithm
85         CGA_toc = time()
86         CGA_times_ALL[run_iter] = CGA_toc - CGA_tic
87
88         # Update arrays
89         CGA_AvgFit_ALL[run_iter, :] = avg_fitness
90         CGA_MinFit_ALL[run_iter, :] = min_fitness
91         CGA_best_Xs[run_iter, :] = CGA.best_individual
92
93     ### PT Gather Results ###
94     PT_AvgFit_ALL = np.zeros((50, MAX_NUM_ITERS))
95     PT_MinFit_ALL = np.zeros((50, MAX_NUM_ITERS))
96     PT_times_ALL = np.zeros(50)
97     PT_best_Xs = np.zeros((50, 8))
98     PT_converg_iters = np.zeros(50)
99
100     # Run PT 50 times with different initialisations
101     for run_iter, initialisation in enumerate(initial_pop_list):
102
103         # Instantiate optimally-tuned PT
104         PT = ParallelTempering(objective_function=KBF_function,
105                                x_dim=8,
106                                range=(0,10),
107                                alpha=0.1,
108                                omega=2.1,
109                                num_replicas=10,
110                                num_chains=25,
111                                exchange_procedure='Always',
112                                power_term=1,
113                                constraints=True)
114
115         # Forcibly reset the PT solutions
116         initialisation = initialisation / 10 # Scale down to the range of the PT algorithm
117         (0-1)
118         initialisation = initialisation.reshape(10, 25, 8) # Share solutions between replicas,
119         i.e. from shape: (250, 8) to (10, 25, 8)
120         PT.current_solutions = initialisation
121
122         # Initialise arrays for storing fitness
123         avg_fitness = np.zeros(MAX_NUM_ITERS)
124         min_fitness = np.zeros(MAX_NUM_ITERS)
125
126         # Max number of iterations = MAX_NUM_ITERS
127         conv_count = 0
128         PT_tic = time()

```

```

126     for i in range(MAX_NUM_ITERS):
127
128         # Algorithm update
129         PT.update_chains()
130         PT.replica_exchange(iter)
131
132         # Update fitness arrays
133         avg_fitness[i], min_fitness[i] = PT.get_fitness()
134
135         # Check if the algorithm has converged,  $|f(x) - f(x_{prev})| < \epsilon$  for 'conv_iters'
iterations
136         if i != 0 and np.linalg.norm(min_fitness[i] - min_fitness[i-1]) < eps:
137             conv_count += 1
138         else:
139             conv_count = 0
140
141         # If the algorithm has converged, store the iteration at which it converged
142         if conv_count == conv_iters:
143             PT_converg_iters[run_iter] = i - conv_iters
144
145         # Time taken to run algorithm
146         PT_toc = time()
147         PT_times_ALL[run_iter] = PT_toc - PT_tic
148
149         # Update arrays
150         PT_AvgFit_ALL[run_iter, :] = avg_fitness
151         PT_MinFit_ALL[run_iter, :] = min_fitness
152         PT_best_Xs[run_iter, :] = PT.get_best_solution()
153
154     # Find the expectation of the average and min fitness across all 50 initialisations
155     CGA_AvgFit_mean = np.mean(CGA_AvgFit_ALL, axis=0)
156     PT_AvgFit_mean = np.mean(PT_AvgFit_ALL, axis=0)
157     CGA_MinFit_mean = np.mean(CGA_MinFit_ALL, axis=0)
158     PT_MinFit_mean = np.mean(PT_MinFit_ALL, axis=0)
159
160     # Find the expected iteration at which CGA and PT converge
161     CGA_Avg_i = np.mean(CGA_converg_iters)
162     PT_Avg_i = np.mean(PT_converg_iters)
163
164     # Plot expected average fitness values for CGA and PT
165     plt.figure()
166     sns.set_style('darkgrid')
167     plt.plot(CGA_AvgFit_mean, label='CGA', color='green')
168     plt.plot(PT_AvgFit_mean, label='PT', color='red')
169     plt.axvline(CGA_Avg_i, color='green', linestyle='--', label=f'CGA Best Converges at {int(
CGA_Avg_i)} Iterations')
170     plt.axvline(PT_Avg_i, color='red', linestyle='--', label=f'PT Best Convergence at {int(
PT_Avg_i)} Iterations')
171     plt.xlabel('Iterations')
172     plt.ylabel('Average Fitness')
173     plt.title('Expected Average Fitness across 50 Different Initialisations')
174     plt.legend()
175     plt.savefig('figures/Final Comparison/CGA vs PT Average Fitness.png', dpi=300)
176
177     # Plot expected min fitness values for CGA and PT
178     plt.figure()
179     sns.set_style('darkgrid')
180     plt.plot(CGA_MinFit_mean, label='CGA', color='green')
181     plt.plot(PT_MinFit_mean, label='PT', color='red')
182     plt.axvline(CGA_Avg_i, color='green', linestyle='--', label=f'CGA Best Converges at {int(
CGA_Avg_i)} Iterations on Avg.')
183     plt.axvline(PT_Avg_i, color='red', linestyle='--', label=f'PT Best Convergence at {int(
PT_Avg_i)} Iterations on Avg.')
184     plt.xlabel('Iterations')
185     plt.ylabel('Minimum Fitness')
186     plt.title('Expected Minimum Fitness across 50 Different Initialisations')
187     plt.legend()
188     plt.savefig('figures/Final Comparison/CGA vs PT Minimum Fitness.png', dpi=300)
189

```

```

190 # Save final results to csv
191 CGA_final_avg = CGA_AvgFit_mean[-1]
192 CGA_final_std = np.std(CGA_AvgFit_ALL[:, -1])
193 CGA_final_min = CGA_MinFit_mean[-1]
194 CGA_final_min_std = np.std(CGA_MinFit_ALL[:, -1])
195 CGA_final_time = np.mean(CGA_times_ALL)
196 CGA_final_time_std = np.std(CGA_times_ALL)
197 CGA_AVG_best_X = np.mean(CGA_best_Xs, axis=0)
198
199 PT_final_avg = PT_AvgFit_mean[-1]
200 PT_final_std = np.std(PT_AvgFit_ALL[:, -1])
201 PT_final_min = PT_MinFit_mean[-1]
202 PT_final_min_std = np.std(PT_MinFit_ALL[:, -1])
203 PT_final_time = np.mean(PT_times_ALL)
204 PT_final_time_std = np.std(PT_times_ALL)
205 PT_AVG_best_X = np.mean(PT_best_Xs, axis=0)
206
207 data = {'CGA': [CGA_final_avg, CGA_final_std, CGA_final_min, CGA_final_min_std,
208               CGA_final_time, CGA_final_time_std, CGA_Avg_i, CGA_AVG_best_X],
209         'PT': [PT_final_avg, PT_final_std, PT_final_min, PT_final_min_std, PT_final_time,
210               PT_final_time_std, PT_Avg_i, PT_AVG_best_X]}
211 df = pd.DataFrame(data, index=['Final Avg. Fitness', 'Final Avg. Fitness Std', 'Final Min.
212                               Fitness', 'Final Min. Fitness Std', 'Total Time Taken', 'Time Taken Std', 'Mean Iters to
                               Convergence', 'Mean Best Solution'])
df.to_csv('figures/Final Comparison/CGA vs PT Final Results.csv')

print(seeds) # These are the random seeds used to generate the initial populations

```

Listing 12: FinalComparison.py